Doctoral Dissertation
Doctoral Program in Energy Engineering (34^{th}cycle)

Towards Autonomous Computer Networks in Support of Critical Systems

By

Alessio Sacco
******

Supervisor:
Prof. Guido Marchetto

Doctoral Examination Committee:

Politecnico di Torino
2022

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Alessio Sacco
2022

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my wife, without whom this thesis would have been impossible. Just thank you, for all.

Also, my loving parents, which helped me during the ups and during the downs. For my family, in particular, my sister, who was always present when needed.

All my friends, which facilitated my journey.

I would also like to acknowledge and express my gratitude to my advisor Guido Marchetto, for his important support in all the activities I was involved in during my Ph.D., steering me in the right direction when necessary.

Despite not being officially my supervisor, a special thank goes to Flavio Esposito, which helped me as an actual supervisor, supporting me multiple times at Saint Louis University, USA. Its enthusiasm towards scientific research has been tangible and contributed to enjoying my doctorate program.

Lastly, all the people I met during this adventure, and all I haven't cited because of the limited space. You were precious.

Abstract

The soaring complexity of computer networks has opened up new opportunities, but it also raised new problems. For example, edge computing, which extends the cloud paradigm and moves it closer to the data source (i.e., to the edge of the network), can pave the way for new interactive applications, which are associated with more stringent requirements. This new paradigm can be used, for example, in life-saving applications such as telemedicine and remote surgery, or first responder support after a natural or human-made disaster scenario. Due to their mission and nature, these services are often referred to as critical systems. However, since these systems are not only limited to real-time video streaming but pose requirements for extremely low round-trip delay, it is desirable that the management and orchestration of networks would consider new methodologies to address the new data flow.

A recent trend dictating this evolution is constituted by the softwarization and virtualization of networks, which have drastically simplified the deployment and real-time reconfiguration of network functions, allowing them to continuously adapt and to deal with dynamic demands in an automated way. Alongside, recent management and orchestration approaches for softwarized networks employ Artificial Intelligence (AI) and Machine Learning (ML) to further reduce reaction time and improve the accuracy of decisions, where the network operations can be automated to the point of realizing autonomous driving networks. However, while automating operations can improve the overall system (it is acknowledged that 70% of network faults are caused by manual errors), AI/ML methods are not the panaceas, and we are still far from having a fully operating and efficient automated architecture.

In this thesis, we present a novel class of software network solutions that share the goal of enabling intelligent and autonomous computer networks, exploring how to exploit the power of AI/ML to handle the growing complexity of the critical systems. We start with routing and network planning, proposing two separate solutions,

Rope and Mystique. The former steers the traffic over paths that are predicted to be unloaded in order to mitigate network congestion. The latter is a network management schema that auto-scales (virtual) network resources to accommodate the traffic demand and reacts to possible failures. Then, acting on TCP congestion control, we propose an ML-based solution aiming to solve the problem of an adequate congestion window update strategy, and that can increase the throughput and fairness while reducing the number of packets lost and delay.

Finally, we consider a specific edge computing problem, namely task offloading, in which a task can be fully or partially offloaded to the close multi-access edge computing in order to speed up the computation. Focusing on task planning management for Unmanned Aerial Vehicles (UAVs), we present two ML-based offloading decision schemes that can be deployed and used depending on the nature of the constraints of devices under consideration. First, we focus on a collaborative offloading decision strategy proposing the use of Multi-Agent Reinforcement Learning (MARL) to jointly improve the energy efficiency and task completion time of edge computing-enabled UAVs. Second, we design a solution less hungry for computation and memory resources, and that, by predicting future device load, supports the UAV decide whether to offload incoming tasks.

In conclusion, we believe that the proposed solutions, and their combination, can lay the foundation for automated systems that better suit modern edge environments and cellular networks by providing unprecedented flexibility and adaptation to even unseen and unknown network conditions.

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**

1.1 Summary of Contributions 4

2 Motivating Applications **7**

2.1 Real-Time Telepathology 8

2.2 Edge-Assisted Unmanned Aerial Vehicles 10

2.3 Tactile Internet 12

3 Data-Driven Routing **13**

3.1 Introduction 13

3.2 Related Work 15

3.2.1 Network Traffic Prediction 15

3.2.2 Adaptive Routing and Traffic Engineering 16

3.3 Rope Overview 17

3.3.1 Measurements Collection 18

3.3.2 Cost Function 19

3.4 Prediction Algorithms Analysis 21

3.4.1 Time-Series Models 21
3.4.2 Machine Learning Algorithms 23
3.5 Prediction Algorithms Evaluation 25
3.5.1 Data set 26
3.5.2 Algorithms Analysis 27
3.6 Routing Evaluation 30
3.6.1 Automate the choice of predictor 31
3.6.2 Routing Performance Improvement 34
3.6.3 Real-case Environment on GENI 35
3.7 Conclusion 37

4 Automatic Network Planning Decisions 39

4.1 Introduction 39
4.2 Related Work 41
4.3 System Design 43
4.3.1 Edge Network Scenario 43
4.3.2 System Components 44
4.4 Auto-Scaling System Model 47
4.4.1 Network Model and Preliminary Definitions 48
4.4.2 Optimizing Quality of Experience, Costs, and Fairness . . . 50
4.5 The Mystique Solution 52
4.5.1 Reinforcement Learning States 53
4.5.2 Actions 53
4.5.3 Reward 53
4.5.4 Multi-Agent Reinforcement Learning Framework 54
4.5.5 Mystique Overall Algorithm 55
4.6 Experimental Settings 56

4.6.1 Implementation 56
4.6.2 Experimental Setup 57
4.7 Performance Evaluation 59
4.7.1 Evaluation Metrics 59
4.7.2 Centralized versus Multi-Agent Reinforcement Learning . . 60
4.7.3 State-of-the-art Comparison 61
4.8 Conclusion 62

5 High-Performance Transport Protocol 63
5.1 Related Work 65
5.2 Congestion Control via Reinforcement Learning 67
5.2.1 Stability Analysis 69
5.2.2 Owl Protocol Design 71
5.3 Owl Prototype Implementation 73
5.4 Protocol Evaluation 74
5.4.1 Trace-Driven Emulation Results 74
5.4.2 Evaluation over the GENI Testbed 77
5.4.3 Partial Network Knowledge Impact 77
5.5 Conclusion 79

6 Task Offloading in UAV Networks via Multi-Agent Reinforcement Learning 80
6.1 Collaborative RL for task offloading 81
6.2 Related Work 82
6.3 Model and Problem Definition 83
6.3.1 Background on Actor-Critic and Multi-agent Reinforcement Learning 83
6.3.2 System Model 87

6.3.3 MARL Framework Formulation . . . 92
6.3.4 Problem Formulation . . . 94
6.4 Our Algorithm . . . 95
6.4.1 MARL system optimization . . . 96
6.4.2 Local Updates and Consensus-based phase . . . 96
6.5 Evaluation Results . . . 98
6.5.1 Experimental Setup . . . 98
6.5.2 Trace-Driven Emulation Results . . . 100
6.6 Conclusion . . . 104

7 A Self-Learning Strategy for Task Offloading in UAV Networks 106
7.1 System Model . . . 107
7.1.1 Local Execution . . . 109
7.1.2 Edge Cloud Offloading . . . 109
7.1.3 Problem Formulation . . . 111
7.2 Regression Prediction Methods . . . 112
7.2.1 Time-Series Analysis with VARMA . . . 112
7.2.2 ML Regression with RFR . . . 113
7.2.3 State Variables in Our Solution . . . 114
7.3 Agent's Decision Process . . . 115
7.3.1 Follow the Perturbed Leader . . . 115
7.3.2 Our Algorithm . . . 117
7.4 Results . . . 118
7.4.1 Experimental Setup . . . 118
7.4.2 Evaluation Metrics . . . 119
7.4.3 Predictor Accuracy . . . 121
7.4.4 Solution Performance Analysis . . . 122

7.4.5 Comparison with State-of-the-art 123
7.5 Conclusion 125

8 Conclusions 126

References 129

Appendix A List of Publications 148

List of Figures

1.1 Research Focus. My research is located at the intersection between network management, edge computing, and machine learning. . . . 4

3.1 Architecture and main RoPE's functionalities. 17

3.2 Training time and error (MAPE) for different training set sizes. . . . 29

3.3 The cost function for the tested algorithms. RFR is the best for ML algorithm, while Holt for TS methods. 32

3.4 (a) Comparison of different classes of algorithms for different use cases and (b) algorithms performance among different topologies with increasing connectivity. 33

3.5 Comparison for different routing strategies. (a) Trade-off between latency and throughput. Our solution provides an higher throughput and a lower latency. (b)-(c)-(d) The effectiveness of the three routing strategies among the three use cases, Disaster, Telepathology and Tactile respectively. 34

3.6 Comparison for different routing strategies over the GENI testbed. Our solution outperforms the related work, as confirmed by (a)-(b)-(c), where the cost function with the proper coefficients is computed for the three use cases, Disaster Scenario, Telepathology and Tactile Internet, respectively. 35

4.1 The goal of our Mystique system is to learn via reinforcement learning how to adapt to network demand fluctuations by creating new virtual switches and split traffic (from t=0 to t=1), and to scale-down nodes and network resources when demands decrease (from t=2 to t=3). 45

4.2 System overview. The Software-Defined Network controller receives as input traffic statistics and outputs new flow routes and power on/off commands. 46

4.3 A sample of emulated network topology used during the experiment campaign. The design reflects the desire of allowing a variety of test conditions and a multitude of available paths. 58

4.4 Reward function for different traffic patterns: (a) uniform, (b) moderate increase, (c) sharp increase, (d) synthetic generation. Four available strategies are compared, highlighting differences between centralized versus decentralized (MARL) model. 60

4.5 Performance comparison with other benchmark algorithms in terms of (a) energy saved with respect to original setting, (b) mean application throughput achieved, (c) fairness index for the flows, (d) system reward. 62

5.1 Owl Overview: reinforcement learning sender's agent interaction with the network. 71

5.2 **(a)-(b) LTE Trace-driven emulation.** Owl vs. previous schemes (using RL or not) tested over two cellular network traces (top-right are better). In both cases, Owl outperforms our benchmark, and has the highest fairness, on average, in both our tested use cases (Section 5.4.1). **(c) GENI testbed evaluation.** Throughput-loss rate trade-off for kernel-level solutions over real networks. Owl optimizes the two quantities simultaneously (Section 5.4.2). 75

5.3 **Our protocol best follows the available bandwidth.** (a) A 60-seconds throughput's evolution compared to the actual link capacity. Owl fits best the Verizon LTE trace; while, especially for Cubic, overshoots in throughput lead to large standing queues. The curves shown have been selected for visual clarity. (b) A 120-seconds utility's evolution. Owl guarantees an adaptive response to the network dynamic changes. 76

5.4 **Network Knowledge Impact on Performance.** (a) Throughput and (b) RTT of Owl protocol for increasing percentage of known network. Somehow surprisingly, the highest performance gaps with respect to other algorithms are obtained when the percentage of network knowledge is either low or very high. (c) Throughput performance with or without network knowledge averaged over different network topologies and increasing number of informing switches. 78

6.1 System Overview: mobile devices, e.g., UAVs, interaction with the edge cloud via cellular and Wi-Fi network. 87

6.2 System performance in terms of (a) task completion time and (b) utility for a varying number of agents and (c) node-antenna distance. The results are compared with similar solutions whose aim is analogous to ours. 101

6.3 System performance evaluation. (a) Application completion time for increasing average computing workload. (b) Comparison in terms of computation rate for various offloading solutions. 102

6.4 (a) CDF of the task completion time for the compared solutions. (b) Utility evolution for different RL-based algorithms. Our model can shorten convergence time compared to the alternatives. 103

7.1 System Overview. The mobile devices, e.g., UAVs, interacts with the edge cloud asking help for the processing of the collected data. . 107

7.2 (a) MAPE error and (b) mean absolute deviation (MAD) for different algorithms. (c) Convergence time comparison, i.e., loss evolution, for FPL and RFR methods. 121

7.3 (a) Queue agents length at varying the task arrival rate. Both experiments consider an increasing fleet size. (b) Task completion time of our FPL-based approach compared to a more complex solution as MAB. Our FPL outperforms this alternative. 123

7.4 CDF of queue agents length, and (c) energy consumption for various offloading solutions. Despite not as an objective of our algorithm, our solution can also limit the energy consumed by UAVs. 124

7.5 Memory resources and (d) CPU consumed during the execution of our considered algorithms. 124

List of Tables

1.1 Summary of contributions and thesis organization. 6

3.1 Time-Series results when the function is fitted for each new observation. 27

3.2 Hyperparameters set in our methods. 27

3.3 Comparison of error for ML algorithms. 28

3.4 Performance evaluation in the context of the Disaster Response application running on the GENI virtual network testbed. Even in these real settings RoPE outperforms related solutions. 36

3.5 Application requirements and the satisfiability achieved by using RoPE. 37

5.1 The network statistics gathered for estimating the upcoming performance. 67

6.1 The contextual metrics gathered for building the state space. 92

7.1 Symbols and notations. 108

7.2 Parameters setting. 119

Chapter 1

Introduction

In recent years, communication networks have witnessed a radical evolution towards more programmability and flexibility. Among the main leading factors for such evolution, we can cite the so-called “network softwarization". Such paradigm was enabled by technologies such as Software Defined Networking (SDN), Network Virtualization (NV), and Network Function Virtualization (NFV), which have also reached wireless networks with concepts including Software Defined Radio Access Networks [1].

The separation of control and data plane, as suggested in SDN, has attracted interest from both academia and industry. In particular, the combination of such network softwarization and new architectures opened up new opportunities. For example, edge computing, a novel paradigm in which resources, instead of residing in the cloud, are moved closer to the sender, i.e., at the edge of the network, can facilitate the rise of new interactive systems, promising simultaneously (ultra) low-latency, high-bandwidth, and reliable telecommunications. Together, the edge computing paradigm and the programmability of the data plane with novel network programming languages such as P4 [2] or the Deep Programmable Data Plane Kit (DPDK) [3] are showing promising business use cases, supporting several applications [4]. Among all examples of these networked systems and applications, in this thesis the focus is on the so-called “critical systems", deployed for life improvement and sometimes even life-saving services. Consider, e.g., a remote surgery operation, where the system should lead to an improvement in the accuracy and dexterity of a surgeon while minimizing trauma to the patient [5]. Similarly, a telepathology

session, in which histological imagery is transmitted over delay and bandwidth-sensitive path to be processed and shared with a remote medical doctor for real-time diagnosis or pre-computation of digital pathology [6, 7]. As another example of critical networked application, we considered a response to a natural or human-made disaster that requires real-time imagery generated by first-responders with the incident commander to recognize faces of disaster victims [8], or the detection of children in an attempt to reunite them with their families [9, 10].

On the one hand, these novel paradigms are opening new applications and business opportunities. On the other hand, however, they are opening new network and application management problems. As highlighted further in each chapter of this thesis, there is a need for new effective and efficient management of softwarized networks and services to cope with the unfolding plethora of opportunities provided by softwarization. Such flexibility does not necessarily have to be addressed by selecting a configuration once, but systems can be adapted continuously and be able to deal with dynamic demands in an automated way [11]. In this perspective, we argue that Artificial Intelligence (AI) and Machine Learning (ML) can play a central role in managing and orchestrating softwarized networks for the following reasons.

First, the tools that network operators use to gather data from the network have not changed appreciably in decades, even as both demands on the network and traffic volumes have increased. The network data collection, the analysis of such data, and the decision of whether and how to adapt the network's configuration in response to changing network conditions (e.g., a shift in traffic demand, an attack), still remain three decoupled steps. Operators perform network management and network optimization tasks on several timescales, often relying on operators' experience and cumbersome adjustments.

Second, new security and performance requirements create a growing need for new approaches to real-time network management that exploit the growing capabilities in programmable networks and systems to support the analysis of real-time streaming data. Despite recent advances in algorithmic and system aspects of streaming applications, the set of queries that network management requires is significantly more extensive than current methods can handle.

Third and lastly, due to the increasing popularity of Internet-connected devices and the various applications that run over the network, the expectations for network reliability and performance are greater than ever. To achieve these goals, network

operators must continuously collect and analyze the various data streams from the network, possibly with lightweight yet accurate methods.

Therefore, we believe that networks can be equipped with AI/ML to reach autonomous run-time decision-making capabilities, as also suggested by recent trends [12–15]. Given the fact that it is almost impossible for human operators to render network management in real-time, it is likely that future networks will apply AI/ML to *autonomously* identify and locate congestion or malfunctions in the network, and opportunely react. For example, to accurately configure and manage itself, the network needs to pinpoint the malfunction, collect and analyze measurements in a stream way. Once metrics are collected, the network reacts to address the sub-optimal behavior via network programmability.

In addition, there is an application push to make networks more reactive and able to accommodate the more stringent requirements. While solutions for real-time communications have been proposed for video streaming [16–19], supporting round-trip time on communication necessary for critical systems remains a major challenge [5]. The challenges introduced by interactive, reliable, and low-latency communications go beyond the current advances in video streaming and are yet to be solved. Thus, despite the proliferation of adaptive application-layer algorithms, a proper network infrastructure handling the traffic of applications with strict requirements cannot be an afterthought in network management.

Our contribution. In this dissertation, we present a novel class of software network solutions that share the goal of enabling intelligent and autonomous networks. We summarize in Fig. 1.1 the broad research area that has driven the research in this thesis. The research question driving the work can be summarized as: *how these new algorithms (such as machine learning) and the new technologies (such as edge computing and SDN) can be used to effectively solve (traditional and not) network management problems*. While AI/ML techniques are promising tools to design methodologies and algorithms for the automation of communication networks, the increasingly powerful computational capabilities of emerging network devices are driving AI/ML solutions closer to the edge of the network (as opposed to a traditionally more computationally powerful network core). Such a shift is promising to “democratize AI services" enabling a wide variety of new AI usages.

In particular, the research contribution of this thesis covers solutions that exploit the power of AI/ML to handle the growing complexity of communication networks

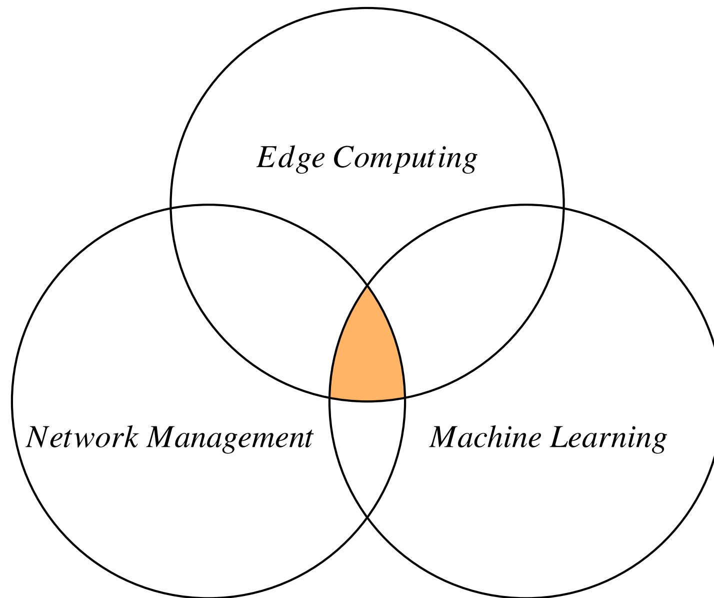


Fig. 1.1 Research Focus. My research is located at the intersection between network management, edge computing, and machine learning.

in the context of resource-constrained, dynamic, and mission-critical environments of modern networks. In this dissertation we focus on the design and deployment of learning algorithms for some specific network operations, ranging from the offloading of tasks to the close edge cloud to the management of virtualized and softwarized networks.

1.1 Summary of Contributions

Starting with the definition of a new mechanism to route packets in an SDN-enabled network, we demonstrated how machine learning-based models, such as time-series forecast and regressors, can help in dynamically avoiding over-congested paths [20]. Our proposed system, RoPE, contributes to mitigating congestion and steering traffic over paths that are (predicted to be) less likely to be congested. In that context, we present and analyze multiple techniques used to enhance routing in SDN edge networks (Chapter 3).

A similar problem regards the planning of network resources. To optimize (virtual) network resource allocation, we proposed Mystique, a system that mitigates congestion and route packets over paths that are predicted to be unloaded, learning

from the load on links to establish the minimal set of active network resources [21]. Our network management schema, using Multi-Agent Reinforcement Learning (MARL), aims at auto-scaling the underlying network topology to accommodate the traffic demand and reacts to possible failures. On the one hand, our solution unburdens network nodes that are over-congested with traffic, to preserve the high bandwidth and high availability of the applications. On the other hand, our solution leverages healing strategies to repair failing nodes and links. As traffic demands ebb and flow, our adaptive and self-driving solution can scale up and scale down the virtual network resources (links and nodes) and also react to failures in a fully automated, flexible, and efficient manner (Chapter 4).

With a similar aim of mitigating network congestion, we worked on Owl, a new TCP congestion-control algorithm based on the Reinforcement Learning (RL) framework [22]. Different from the aforementioned work, in this solution the modification occurs at the end-hosts rather than on the network infrastructure. At the same time, in contrast to other Machine Learning-based approaches for transport protocols, we conduct training at the source and decide the next value of the congestion window, also using an in-network mechanism when such information is accessible from the sender. Using these metrics, our solution can increase the throughput and fairness while reducing the number of packets lost and delay in a variety of contexts, with notable improvements in cellular networks. We also showed how our solution converges to a fair resource allocation after the learning overhead (Chapter 5).

The last contribution set considers problems originated by the advent of edge computing and highly dynamic networked systems: computation offloading. In particular, we propose an edge cloud-assisted architecture for distributed and adaptive task planning management for Unmanned Aerial Vehicles (UAVs) via task offloading.

First, we consider a set of distributed offloading decision strategies, and we propose the use of Multi-Agent Reinforcement Learning (MARL) to jointly improve the energy efficiency and task completion time of edge computing-enabled UAVs swarms [23]. However, in the case of task offloading, the model also decides the Radio Access Technology (RAT) to consume, i.e., Wi-Fi or cellular, to transmit the task from the device to the edge cloud. The optimization problem can run in real-time by combining information coming from other devices and enables the training of the model in a collaborative way (Chapter 6).

Table 1.1 Summary of contributions and thesis organization.

Networking Problem	Chapter	ML Method
Routing	Ch. 3	Time-series, ML regressors
Auto-scaling	Ch. 4	MARL
TCP Congestion Control	Ch. 5	RL
Task Offloading	Ch. 6, Ch. 7	MARL, Time-series, ML regressors

Second, we presented a solution that is less hungry for computation and memory resources, and that supports the UAV during the decision of offloading incoming tasks. Such a decision is taken on the basis of the predicted behavior of the agent, suggesting whether edge cloud is beneficial or not to the incoming tasks. Two alternative methods are designed to perform such a prediction about future device load: a model belonging to time-series class and a model belonging to the class of ML regressors. In such a way, not only the agent learns how to forecast future values of enqueued tasks, but it can also learn online what type of model is more accurate, leading to a self-learning approach. In particular, having chosen two different ways in treating the input metrics, this approach also provides flexibility and adaptability, resulting in a learning agent that can select which predictor best fits a particular environment [24]. While other RL-based models can be computationally expensive to run onboard of constrained resources devices, this formulation simplifies the decision process and makes the solution suited for constrained scenarios (Chapter 7). Studying both RL-based (Chapter 6) and regression models (Chapter 7), we showed how the offloading decisions could be taken at run-time, adapting the network conditions.

We summarize the contributions in Table 1.1, where for each specific addressed problem, we report the ML method used and the corresponding chapter in this manuscript. We argue that these solutions, and their combination, can lay the foundation for automated systems that better suit modern edge environments and cellular networks.

Chapter 2

Motivating Applications

The advent of new technologies, *e.g.*, SDN, 5G, and edge computing, has been designed to support the development of applications with very strict requirements, *e.g.*, very low latency and high throughput [25]. Among them, *(i)* Industry 4.0, in which robotic machinery is controlled remotely, *(ii)* the Tactile Internet which can enable haptic interaction with visual feedback, providing the illusion of remote touch, *(iii)* VR/AR and Holographic-Type Communications (HTC) in which the application is supposed to rapidly adapt streamed contents based on changes in user position or viewing angle, *(iv)* telemedicine, where medical devices, or simply medical information, are accessed remotely during a teleoperated session, *(v)* assisted or connected vehicles, such as drones and cars, which involve communication and data exchange between vehicles and traffic infrastructure are pivotal to realize the vision of smart cities and intelligent transportation.

In this chapter, we specifically focus on three applications that would require network management optimizations and that have been considered in the presented solutions. We start from a medical scenario, namely telepathology, in which it is common to transfer (very large) medical images for real-time diagnosis. Guaranteeing both high throughput and low latency is the mission of new networking systems.

We then continue with recent applications employing Unmanned Aerial Vehicles (UAVs) to assist humans in accomplishing tasks that would be impossible without them. UAV-based systems have experienced a constantly increasing popularity in the last years, mainly thanks to their maneuverability, flexibility, and limited deployment costs. However, since the challenge of keeping an acceptable quality of service with

stringent delay constraints for these UAVs networks grows with the drone-based applications, the design of efficient techniques to re-assign tasks is key.

Lastly, the literature refers to the new internet network that combines ultra-low latency with extremely high availability, reliability, and security as Tactile Internet [26]. To achieve this promise, there is a need to combine multiple technologies at the network and application level, and utilize the edge computing paradigm to move data closer to the user.

2.1 Real-Time Telepathology

The field of medical pathology is concerned with the causal study of disease, whether caused by pathogens or non-infectious physiological disorders. A significant part of the job of pathologists is characterized by visualizing histological images via a multi-lens microscope. Often they analyze histological images on a glass slide when the patient is still under a tumor removal surgery. In such situations, a quick and correct pathology assessment is crucial as it defines vital next steps for the surgeon team and the right follow-up treatment for the patient. In the vast majority of non-trivial pathology cases, to minimize the time to response to the surgeon team and the probability of incorrect assessments, pathologists ask for second opinions to nearby experts (if available) by physically carrying privacy protected glass specimens. When not enough local experts are available, a *telepathology* system can be used to transmit high-resolution images of specimens to remote doctors.

Telepathology is the practice of digitizing histological or macroscopic tissue images based on a glass slide for transmission along telecommunication pathways for diagnosis, consultation, or medical education. Pathologists seek second opinions from local experts either intraoperatively (rapid frozen section assessment of margins during tumor excision, for example) or during routine sign-out of difficult or unusual entities; in most settings, this involves the transport of physical glass slides. Hence, every pathologist should ideally be able to consult an expert via telepathology, either intraoperatively or for routine sign-out.

Today, however, telepathology is often grouped in expensive software packages that most local hospitals cannot afford, and it is practically unused for the applications that would need it the most: fast and reliable consultations, as well as multi-student,

live teaching sessions; pathology is nowadays mostly taught via offline methods or via one-to-one mentor-student specimen analysis.

Our work has been focused on enhancing the currently booming field of Digital Pathology (DP), which can be seen as an attempt to adopt digital technologies to reduce the time and improve the accuracy of a diagnosis, for example by reducing the number of physical slide requests [27, 28]. However, developing technologies to efficiently digitalize image samples and building solutions that pathologists can easily access is a challenge.

Pathologists often analyze histological samples on a glass slide while the patient is still in the operating room. Without a telepathology solution, pathologists usually ask for second opinions from nearby experts (if available) by physically carrying glass slides, in order to minimize the time to diagnosis and report to the surgeon. When not enough local experts are available, the presence of a telepathology system could be a critical factor in providing the best care for the patient. Moreover, a telepathology system can be used to transmit high-resolution images of specimens to connected experts in order to speed up the diagnosis of more routine cases in more rural treatment locations. Finally, it might also mitigate discordance between pathologists [29]. In summary, telepathology has the potential to have a positive impact on the delivery of expert care patients regardless of the location of their surgical treatments.

Telepathology solutions can be used not only to connect rare experts with patients, but also for the rapid diagnosis of standard cases in locations that have patients without having schools of medicine. Telepathology is often enclosed in the telemedicine field, but it differs both in the subject and the aim of such practice. This difference leads to different requirements that the underlying network has to guarantee [30].

However, current telepathology solutions are limited by the technology, the scale, and the (best-effort) performance of the underlying telecommunication media on which they rely on, i.e., the Internet or, at best, a virtual private network for in-hospital offline, i.e., non-real-time, consultations.

An attempt to further reduce latency and enable a fast image processing, has been presented in [31], where a novel telepathology system has been designed to employ a close edge computing for partial or full processing offloading. This collaborative image analyzer supports high-speed data transfer with low-latencies, but aims to

provide services for real-time consultations in both education and surgery scenarios using edge computing.

While it is clear that emerging technologies can drastically improve service interactivity, these solutions still require excessive overhead in essential network operations. Learning-based approaches can help towards automating actions, but it is not sufficient. Moreover, although the utilization of an edge cloud can lead to low latency, other application-layer and/or network-layer optimizations are advised.

2.2 Edge-Assisted Unmanned Aerial Vehicles

Unmanned Aerial Vehicles (UAVs) are often used for collecting data and sending them to the edge/fog, e.g., for data-intensive visual computing. At network edges, indeed, there may be present more resources that can speed-up the processing. In particular, data-intensive visual computing requires seamless processing of imagery/video at the network-edge and resilient performance to guarantee adequate user Quality of Experience (QoE) expectations. This is particularly critical, e.g., in (natural or human-made) disaster scenarios, due to the poor bandwidth availability and the highly variable conditions. These applications should be able to provide rapid awareness through videos or audios collected at salient incident scenes in order to plan a proper response that can minimize disaster impact and/or save lives [32].

To meet such network-edge data-intensive computations and local storage requirements, *edge computing* is a valuable solution [33], by providing on-demand network, storage, and computational resources that compensate (scaling up and scaling down on demand) the insufficient local processing capabilities within a geographical area of interest. Edge computing extends the notion of cloud, but it is placed closer to the location of users and data sensors, reducing latency and enabling real-time decision making. A few examples of how edge computing could be of help in the above described scenarios are reported in the following.

Reconnaissance to save lives. A (very) large fleet of camera-equipped UAVs collect visible (or infrared) imagery, e.g., to recognize body temperatures or identify bodies under ruins or massive avalanches. In such environments, image processing is key, to first enhance the image, e.g., dehaze, stabilize, compress inputs for lower level image processing, and then apply computationally intensive computer vision algorithms.

Transmitting such data to a remote cloud is thus unfeasible, given the poor connection bandwidth which dramatically increases the data transfer time.

Reuniting lost citizens and families. Online face recognition software runs at the edge and acts on imagery snapped from cameras onboard the UAVs. Face image feature extraction processing performed at the network edge would attempt to match against a database of missing people without encountering poor network or processing performance. The face detection and identification can gain great benefit from utilizing deep neural networks-based models, that are rapidly improving their performance in this field.

Property Surveillance. Alarms or other actuators may be triggered if the continuous monitoring performed by UAVs detects activities of concern, such as a fire, a human intrusion, or a broken window. A first video analytic pre-scanning phase is recommended to run at the edge, and only upon the completion data could be sent to the cloud core, where a more in-depth analysis can occur and the video can be shared with law enforcement for further investigation.

to enable immediate feedback in these scenarios, crucial especially for survivors' rescue, IoT devices today could benefit from the mobile edge computing paradigm [34]. In particular, one of the most important mechanisms in edge computing is cyber foraging: processes from mobile resources delegate computations or code to execute to servers within the edge computing infrastructure [35]. A particular case of cyber foraging is also known as offloading, where the cyber foraging mechanism is orchestrated by mobile devices. A solution optimizing the offloading decisions, i.e., what and when to offload, would speed up the transfer of media-rich visual information.

Moreover, since traffic needs to be handled dynamically and with low-latency constraints, also routing is a crucial infrastructure management orchestration mechanism. Although geographic routing-based approaches are generally suitable for these applications, there is a lack of providing sustainable high-speed data delivery to the edge cloud gateway [36].

2.3 Tactile Internet

The Tactile Internet is the evolution of the mobile Internet and enables real-time control of the Internet of Things (IoT) for very latency-sensitive applications. It adds a new dimension to human-to-machine interaction by enabling tactile and sensations, and at the same time revolutionizes the interaction of machines. The Tactile Internet enables haptic interaction with visual feedback. The term haptic relates to the sense of touch, in particular, the perception and manipulation of objects using touch. The visual feedback will encompass not just audiovisual interaction, but also robotic systems that can be controlled in real-time as well as actuating robots, *i.e.*, those that can activate a motion.

Nowadays, data rates increased in the orders of magnitude, as well as the data capacity [5], but there is another frontier to be considered: the reduction in the end-to-end latency of interaction has not dropped below the requirement for telephony. Long-term evolution (LTE) achieves a typical end-to-end latency close to 100ms [37], exceeding the order of 1-ms requirement needed to enable Tactile Internet applications [5]. At the same time, fifth generation (5G) mobile communications systems underpin this emerging Internet at the wireless edge [38]. A recent trend is the use of Mobile Edge Computing (MEC) as a solution to reduce the delay and provide a way for offloading computation from the cellular network [39]. However, the latency reduction is still an open problem due to an intrinsic lack of the available infrastructures. The SDN paradigm is shown to be helpful for these applications [40], but real support for very low latency communications is an urgent need to enable the still unexpressed haptic applications.

Starting from next chapter, we consider these applications for the design of the solutions and for evaluation.

Chapter 3

Data-Driven Routing

Traditional solutions operate on the routing or flow path decisions, by typically using source routing and policy-based routing for the path control. However, to adapt these decisions to the current network load, the algorithm must be able to collect information about network traffic, which can be achieved via NetFlow [41], SNMP, or custom protocols. A common scenario, due to the ease of use of SDN in prototyping, considers OVS switches [42] featuring OpenFlow in combination with an SDN controller to meet the above requirements. These features are key for dynamic and performance-aware routing solutions.

3.1 Introduction

Delivering simultaneously low-latency and high-bandwidth reliable telecommunications is a challenge, especially when the underlying infrastructure is unstable and applications impose tight constraints. Solutions for real-time communications have been proposed when the application is bound to video streaming [43, 44, 16, 45, 46].

Many of them are based on sound design and target bit rate adaptation. Aside from ignoring other end-to-end performance improvement techniques such as traffic compression, these solutions perform poorly within edge computing use cases, where the underlying network needs also to be optimized, in response to offloading requests [47, 33, 48].

The work presented in this chapter has been partially published in [20].

Most of these solutions, however, train their learning system on specific datasets, without the ability to adapt. While complicated machine learning techniques such as transfer learning exist [14], such techniques could be applied to overcome the dataset-tailored limitation. In this solution, we take a more humble approach and we show its effectiveness. In particular, we introduce RoPE, a Software-Defined Networking (SDN)-based architecture whose goal is to select the best (physical or virtual) route by applying the most appropriate bandwidth prediction algorithm, chosen adaptively, on the basis of the amount of data collected and the response time deadline. RoPE leverages the availability of multiple paths and relies on the idea that the bottleneck for delay-sensitive applications is at the edge [49, 50].

Our design is based on the observation that, in recent years, the field of prediction has achieved excellent results when enough data are available. When insufficient data are available instead, other classes of prediction algorithms may be a better fit. In this context, many forecast-based or data-driven solutions have been proposed [14]. The question we propose to answer instead in this work is: *which bandwidth prediction algorithm works best, based on the variance of our network measurements and on application constraints?*

To address this question, we prototype and evaluate RoPE with both numerical, event-driven simulations, and with scalability tests over the large-scale GENI testbed. In particular, we make the following contributions.

We design and implement a novel architecture that integrates QoE estimation and bandwidth prediction directly into an edge-based application. The prediction phase is used for selecting the best routes on the basis of global traffic condition information gathered from an SDN controller. Hence, we defined a new strategy for the route selection while the prediction continues during system operation, with consequent possible traffic re-routing.

To adapt to different edge-based applications and evaluate its performance, we define a new cost function that embraces the most common evaluation parameters.The collection of our results evaluating three separate uses cases are a mixture of expected and surprising results.

RoPE stands for Routing Prediction at the Edge.

3.2 Related Work

Our approach is based on the prediction of traffic conditions to modify routing for edge-based applications. In this section, we analyze the literature related to the main components of the solution: (i) the recent prediction algorithms for networking, and (ii) the existing routing solutions that rely on machine learning methods to improve traditional strategies.

3.2.1 Network Traffic Prediction

The prediction of traffic conditions is crucial in network operations and management for today's increasingly complex and diverse networks. It entails forecasting future traffic and traditionally it has been addressed via Time Series (TS) algorithms. The main goal in TS is to construct a regression model capable of drawing an accurate correlation between future traffic volume and previously observed traffic volumes. Existing TS models can be broadly decomposed into statistical analysis models and supervised ML models. Statistical analysis models are typically built upon the generalized Autoregressive Integrated Moving Average (ARIMA) method, while the majority of learning for traffic prediction is achieved via supervised Neural Networks (NNs). However, with the rapid growth of networks and the increasing complexity of network traffic, traditional statistical models are seemingly compromised, giving rise to more advanced ML models [14]. More recently, efforts have been made to reduce overhead introduced by the prediction process or improve accuracy in traffic prediction by employing features from flows, other than traffic volume. Prior work focused on NNs and showed how this approach outperforms TS [51]. However, the use of NNs implies an offline training phase and a huge quantity of training data [52], which is unfeasible for some applications where traffic demand is volatile, as in a disaster response scenario. In our scenario we do not have such a quantity of data, therefore we focus on lighter approaches that enable an online training phase. These models are then compared against Machine Learning methods where the training is performed offline. Furthermore, for edge-based applications, there are no databases available online as for traffic traces provided by ISPs or inter and intra DCs [53].

For this reason, in our work we focus on other ML algorithms that also do not necessitate a long training phase. Many techniques have been developed to measure path properties as summarized by CAIDA [54]. In particular, several studies [55–57]

focused on the measurement of the available bandwidth, needed for data collection in our predictor. By available bandwidth, we mean the minimum unused capacity on a given end-to-end path. These measurements are usually collected with probe packets. In this work, we do not actively probe but we rely on packets sent from switches to the controller. In this way, packets containing network statistics do not affect the user traffic, since the communication with the controller is separated from the data plane [58, 59].

Finally, machine learning techniques have been widely applied to network measurement. As suggested by [60] intradomain traffic engineering can benefit from the application of ML algorithms or a a Reinforcement Learning (RL)-based approach. For example, there are applications in the network intrusion detection field (*e.g.*, [61]) and for round-trip time prediction [62]. In contrast to NNs-based algorithms, Support Vector Machine (SVM) has low computational overhead and is more robust to parameter variations, *e.g.*, time scale, number of samples. However, this approach is usually applied to classification rather than regression. Bermolen *et al.* [63] applied SVR (the regression variant of SVM) for link load forecasting. Furthermore, He *et al.* [64] extensively studied history-based predictors using three different time series forecasts. Other approaches for TCP throughput prediction employ "bandwidth probes", small TCP file transfers, *e.g.*, 64kB, to collect the measures [65, 66].

3.2.2 Adaptive Routing and Traffic Engineering

Even though much work has been conducted to improve the quality of prediction over network traffic, only a few solutions exploited these results to develop new routing strategies [67, 68].

Instead, other prediction-driven routing approaches have been based on Reinforcement Learning (RL), with the aim of coping and scaling to complex and dynamic network topologies [69, 70]. Even though RL would be a viable solution, we used a time-series approach as it offers the possibility of predicting a specific parameter. Such a parameter can then in turn be used to assess if a given traffic flow fits a physical path. If the flow does not fit the path, a better route is chosen looking at other available paths.

The same problem can be clearly addressed by means of traffic engineering solutions, *e.g.*, [71–73]. In particular, COYOTE [73] aims to minimize link over-

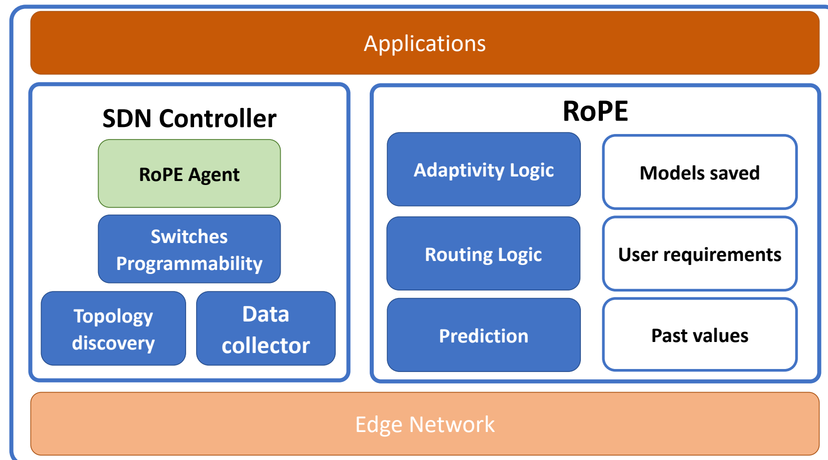


Fig. 3.1 Architecture and main RoPE's functionalities.

utilization by engineering the traffic generated with optimal traffic splitting ratios. Given the limited knowledge of traffic demands, this method strategically advertises fake links and nodes to adjust the splitting ratios resulting from the traditional ECMP mechanism. We share with this solution the idea of adapting the routing to address a link utilization problem; however, our focus is to better support for edge-based applications without reserving resources for tasks that could be rarely executed.

3.3 Rope Overview

Our proposal is to use bandwidth prediction on links to drive routing operations so that the best available path is selected. Given a large number of available prediction algorithms and the differences in requirements to satisfy each application, we also introduce a cost function that captures the policy programmability of the proper algorithm for each specific context. The design goal of such policy knobs is to extract the invariances in the routing prediction mechanism. Network management application programmers then may tune this utility based on their needs and constraints.

Our architecture implementation includes a Ryu SDN controller that collects data from the switches and communicates them to the RoPE agent (Figure 3.1) running on it. This component allows the necessary information sharing between the controller and RoPE. In RoPE, the most important component is responsible for predicting on the basis of the data received and prior information. The data-driven engine selects the best path combining user requirements and the future available bandwidth on

a link. To select the best path, knowledge about the topology of the network is necessary, and this information is obtained and transmitted by the controller. The routing process combines the output of the prediction with the topology information and changes the flow rules of the switches to select the path consistently among all the devices.

RoPE saves the collected data as recent history, which is in turn used by the prediction algorithms. Notice that not all the algorithms need online training (see Section 3.4). For some algorithms, the training phase must be performed offline because it requires a long time, as illustrated in Section 3.6.1. For these algorithms, the SDN Controller can make use of the saved model to predict the next bandwidth value. In essence, the prediction might be based on models saved and past values, as shown in Figure 3.1. The selection of all the parameters is based on the data analysis performed beforehand and described in the following sections.

3.3.1 Measurements Collection

Each managed switch is connected to the Ryu controller, which periodically collects information on their state. In particular, we collect network state statistics of ports (incoming or outgoing packets), flows, and the switch connectivity status.

Since paths do not change very frequently, it is unnecessary to acquire statistics from switches with very high granularity. In our implementation, we use a collection period of 5-seconds, as in [43]. This value is result of a preliminary empirical analysis. In the rest of the chapter, we refer to such interval as an *epoch*.

Data acquired are grouped per switch ID and in chronological order. This is implemented on the controller by logging the received information in a file for every switch. Each row in this file corresponds to an observation per epoch and is formatted as follow:

$$[timestamp, bandwidth, bytes, packets, packets_port],$$

where $timestamp$ denotes the time at which the record is obtained, $bandwidth$ is the the measured bandwidth, $bytes$ refers to the number of bytes sent and received by the switch, $packets$ expresses the total number of packets sent and received by the switch, and, lastly $packets_port$ indicates the amount of packets sent and received

in the selected port. Note how the timestamp is essential to apply TS analysis, while it is not used by ML algorithms. RoPE uses the statistics collected to fit the model. With a period r of 20 s (selected to avoid network instability) we predict the future available bandwidth and decide when to steer the route.

Algorithm 1 Prediction-based routing

1: Let t be the epoch, and r the prediction period
2: Let A and B be the target source and destination
3: $P \leftarrow$ all the paths between A and B
4: $P_s \leftarrow$ the best s paths in P
5: $U \leftarrow$ best path
6: **for** every epoch t **do**
7: Monitor the paths in P_s
8: **if** r has elapsed since last prediction **then**
9: $FL_s \leftarrow$ future predicted load on the s paths in P_s
10: $FL_U \leftarrow$ future predicted load on U
11: **if** $FL_U > Threshold$ **then**
12: $U \leftarrow P[min(FL_s)]$.
13: **close**;

Overall Procedure. The objective of the algorithm is to optimize the available bandwidth between the source A and the destination B, which affects the application transmission time. In the telemedicine example described before (Section 2), A is the machine connected to the microscope, while B is the edge server. The controller first detects all the paths available for the pair (A, B) and stores them into the set P. Then, it selects the best s paths according to the current load on all links composing the path. The parameter s is used to avoid looking for all the paths if this number is significant. Every epoch, the controller obtains the statistics and saves them for the prediction, which occurs every period r. In this phase, we estimate the future value of traffic load for the paths in P_s, and the path whose prediction is the minimum, *i.e.*, "argmin", is set as the default one.

3.3.2 Cost Function

RoPE predicts the bandwidth on links and selects the best path on the basis of this value. However, different applications have different requirements in terms of throughput, latency, jitter, and different prediction algorithms may have different effects on these parameters.

To evaluate the fitness of such an algorithm to the use case, we define a cost function $C_{K,I}(D)$ that takes into account the above aspects of communication. While the metric is inspired by similar studies [74] in our case we are not limited to video streaming scenarios. The cost function $C_{K,I}(D)$ of a sent file which requires D bytes, I packets and K time intervals, is made up of the following terms:

1. *Average Throughput*: the average throughput per time interval k: $\frac{1}{K}\sum_{k=1}^{K} T_k$, where T_k denotes the throughput at interval k
2. *Average latency*: the average latency per packet i:$\frac{1}{I}\sum_{i=1}^{I} L_i$, where L_i is the latency for packet i
3. *Average jitter*: the average jitter between two consecutive packets: $\frac{1}{I-1}\sum_{i=2}^{I}|L_i - L_{i-1}| = \frac{1}{I-1}\sum_{i=2}^{I} J_i$, where J_i indicates the jitter for packet i
4. *Average jitter variation*: the average difference of jitter among two consecutive jitter measurements $\frac{1}{I-2}\sum_{i=3}^{I}|J_i - J_{i-1}| = \frac{1}{I-2}\sum_{i=3}^{I}\Delta J_i$ where ΔJ_i refers to the jitter variation for packet i

Notice that, besides the standard performance metrics of throughput, latency, and jitter, it is worth also considering the jitter variation since for interactive systems, this element affects the user experience, as demonstrated in [75, 76]. Users and application programmers may have different preferences on which of the four components is more important, so we define a tunable objective function as a weighted sum of the aforementioned components:

$$C_{K,I}(D) = \alpha D \frac{K}{\sum_{k=1}^{K} T_k} + \mu \frac{1}{I}\sum_{i=1}^{I} L_i + \lambda \frac{1}{I-1}\sum_{i=2}^{I} J_i + \gamma \frac{1}{I-2}\sum_{i=3}^{I}\Delta J_i \tag{3.1}$$

Here $\alpha, \mu, \lambda, \gamma$ are non-negative weighting parameters corresponding to average throughput, average latency, average jitter and average jitter variation respectively. A relatively small α indicates that the user is not particularly concerned about a very high bitrate; the large γ is, the more effort is made to achieve smoother changes of video quality. A large μ, relatively to the other parameters, indicates that a user is deeply concerned about low latency communication.

In summary, this definition of $C_{K,I}(D)$ is quite general as it allows us to model varying user preferences on different contributing factors. The goal of our routing

strategy is to minimize (3.1) in order to guarantee the optimal user experience. In fact, a higher throughput, along with lower values of latency and jitter, leads to a lower value for the function. Therefore, we need to select the proper prediction method in order to obtain the best routing strategy that minimizes (3.1).

3.4 Prediction Algorithms Analysis

The task of bandwidth prediction can be formulated as a regression problem, *i.e.*, predicting a real-valued number based on single or multiple real-valued input features. For the sake of clarity we classify the applied algorithms in 2 categories, *(i)* Time-Series (TS) algorithms, *(ii)* Machine Learning (ML) algorithms. The following subsections reflect this classification and each one describes in-depth the structure of our algorithms.

The idea is to predict the bandwidth, in such a way the controller can check whether the desired application fits the network load. For instance, if the application is sending a video streaming of 300kb/s and the predicted available bandwidth of the current path is 500kb/s, this means the path complies with the requirements. If the available bandwidth is 200kb/s, the controller enforces a new path.

3.4.1 Time-Series Models

These solutions are based on traditional regression algorithms that predict the future values using the history and the evolution of such value in the past. The history used is made up of past values associated with the timestamp. The presence of the tuple $< timestamp, value >$ leads to the name Time-Series.

Simple Exponential Smoothing. Simple Exponential Smoothing (SES) is a good choice for data with no clear trend or seasonality. Let y_t be the bandwidth on a link at time t. We compute a k-steps ahead prediction. Formally, we forecast the value of the bandwidth at time $t+k$, y_{t+k}, where k is also called horizon.

$$y_{t+k} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + ..., \tag{3.2}$$

where α is the smoothing parameter for $0 \leq \alpha \leq 1$. If α is large (close to 1), more weight is given to more recent observations. The quantity y_{t+k} represents the predicted value and is used to decide whether or not a congestion will occur.

Holt-Winters. The prediction is composed of three submodels that fit a time series: an average value, a slope (or trend) over time and a cyclical repeating pattern (seasonality) [77]. These three aspects of the time series behavior are expressed as three types of exponential smoothing. The model requires several parameters: one for each smoothing (α, β, γ), the length of a prediction season, and the number of periods in a season. Here below we report how the Holt-Winters seasonal method includes the forecast equation and three smoothing equations: one for the level $\mathrm{L_t}$, one for the trend $\mathrm{b_t}$ and one for the seasonal component denoted by $\mathrm{S_t}$, with smoothing parameters α, β and γ:

$$\begin{aligned} level\ L_t = &\quad \alpha(y_t - S_{t-s}) + (1-\alpha)(L_{t-1} + b_{t-1}), \\ trend\ b_t = &\quad \beta(L_t - LL_{t-1}) + (1-\beta)b_{t-1}, \\ seasonal\ S_t = &\quad \gamma(y_t - L_t) + (1-\gamma)S_{t-s}, \\ forecast\ y_{t+k} = &\quad Lt + kb_t + S_{t+k-s}, \end{aligned}$$

where s is the length of the seasonal cycle, for $0 \leq \alpha \leq 1$, $0 \leq \beta \leq 1$ and $0 \leq \gamma \leq 1$.

ARIMA. ARIMA is a class of models typically used for analyzing and forecasting time series (*e.g.*, financial market data). A standard notation for this method is ARIMA(p,d,q), where the parameters account for seasonality, trend, and noise in datasets. In particular, p captures the auto-regressive component *i.e.*, the number of lag observations included in the model, also called the "lag order"; d captures the integrated part of the model, it is the number of times that the raw observations are differenced, also called the degree of differencing; q captures the moving average part of the model and represents the size of the moving average window, also called the order of moving average. The ARIMA overall model is given by the following equation:

$$\left(1 - \sum_{i=1}^{p} \alpha_i L^i\right) \left(1-L\right)^d y_t = \left(1 + \sum_{i=1}^{q} \theta_i L^i\right) \varepsilon_t, \tag{3.3}$$

where L is the lag operator — the number of past samples considered during the prediction — and α_i are the parameters of the autoregressive part of the model; the θ_i are the parameters of the moving average while ε_t are error terms. Such error terms ε_t are generally assumed to be independent and identically distributed (i.i.d.) variables sampled from a normal distribution with zero mean, which is what we did.

SARIMA. To deal with seasonal effects, we make use of the seasonal ARIMA (SARIMA), which is denoted as ARIMA$(p,d,q)(P,D,Q)s$. Here, (p,d,q) are the non-seasonal parameters described above, while (P,D,Q) follow the same definition but correspond to the seasonal components of the time series. The term s is the periodicity of the model (4 for quarterly periods, 12 for yearly periods, etc.).

The Ryu controller is in charge of collecting all bandwidth values and save them in a time series $Y = \{y_t, y_{t-1}, ..\}$. The sequence is then used to fit the model and find the aforementioned parameters. Once the model is built, it is used to forecast the y_{t+k} value, which is then used to avoid congested paths in a telepathology session.

3.4.2 Machine Learning Algorithms

Machine Learning has received great attention in recent years, due to the ease of use and the wide range of applications that can benefits. In this section we define a model for the most popular algorithms, providing a brief explanation of the advantages and disadvantages of applying for each of them. In our model the set of features used is represented by [timestamp, bandwidth, bytes, packets, packets_port], however, for ML methods only a subset is considered:

1. Δ *Packets:* the number of packets received and transmitted by the switch in the time interval;
2. Δ *Bytes:* the number of bytes received and transmitted by the switch in the time interval;
3. Δ *Packets port:* the number of packets received and transmitted by the switch on a certain port in the time interval.

Our problem lies in the Regression procedure since the aim is to predict a continuous value, as opposed to other well-known problems such as classification and clustering. A real number is more effective than a class value as in the Classification problem because it can be used to check if a streaming video will be delayed or not, as described in Section 3.6. By computing the available bandwidth on a path, we are able to verify whether the bit-rate of communication fits the path or not, and in case move to another path. Hence, the output variable is the bandwidth of the links

connected to the switch. The predicted value is the same as the TS models, while in ML models the input set is based on more features than just the past bandwidth.

Linear. The simplest machine learning model is to build a linear regression model, where there is a linear relationship between the dependent (y) variable and the set of independent (x) variables.

Polynomial. Polynomial regression is a special case of linear regression. But in this case, higher order powers (2nd, 3rd or higher) of an independent variable are included.

Support Vector Regression. Support Vector Machines (SVMs) are supervised learning models [78], that aim to analyze data and recognize patterns, used for classification tasks. Support Vector Regression (SVR), is the regression version of the popular SVM and a state-of-the-art machine learning tool for multivariate regression.

Gradient Boosting Regression. Gradient boosting is a machine learning technique used both for regression and classification problems. Like other boosting methods, it builds the model in a stage-wise fashion, and it generalizes them by allowing optimization of an arbitrary differentiable loss function. The intuition behind the gradient boosting algorithm is to repetitively leverage the patterns in residuals and strengthen a model with weak predictions and improve it. Once a stage that do not have any pattern that could be modeled is reached, residuals modeling can be stopped (otherwise it might lead to overfitting). In other words, for Gradient Boosting Regression (GBR) a regression tree is fit on the negative gradient of the given loss function.

Partial Least Squares Regression. Partial least squares regression (PLSR) is a statistical method similar to other regressors; instead of finding hyperplanes of maximum variance between the dependent and independent variables, it finds a linear regression model by projecting the predicted variables and the input variables to a new space [79]. PLSR is used to find the fundamental relations between the two matrices X and Y, i.e. a latent variable approach to model the covariance structures in these two spaces. PLSR is particularly suited when there is multicollinearity among X values. Conversely, standard regression will fail in these cases (unless it is regularized).

Decision Tree Regression. A decision tree has a flow-chart-like structure, where each internal (non-leaf) node denotes a test on an attribute. Each branch represents the outcome of a test, and each leaf node holds a class label. The topmost node in a tree is the root node. The general approach of deriving predictions from a few simple if-then conditions can be applied to regression problems as well. Unlike linear models, Decision Tree Regression (DTR) is able to capture non-linear interaction between the features and the output value [80].

Random Forest Regression. The random forest model for regression (RFR) is a type of additive model that predicts by combining decisions from a sequence of base models. More formally this class of models can be written as:

$$g(x) = f_0(x) + f_1(x) + f_2(x) + ...,$$

where the final model g is the sum of simple base models f_i. Here, each base classifier f_i is a simple decision tree. This broad technique of using multiple models to obtain better predictive performance is also known as model ensembling. In RFR, all the base models are constructed independently using a different subsample of the data.

As a matter of fact, classical and ML methods are not that different from each other but distinguished by whether the models are more simple and interpretable or more complex and flexible. Hence, classical statistical algorithms tend to be much quicker and easier-to-use.

3.5 Prediction Algorithms Evaluation

This section exposes the logic of the methods and the errors in the prediction of the future available bandwidth on a single path. The path consists of a certain number of link, where the assumption is that the SDN controller knows the topology of the network. Collected data are split into three sets: Training set, Validation set, and Test Set. Training Set is used to decide the parameters of the algorithm and Validation Set to compare the performance of a single family of algorithms with different settings. Finally, we use the Test Set to assess the quality of implemented algorithms.

In this section, the algorithms are compared on the basis of the accuracy of predicting. Even though ML algorithms rely on features to predict, while TS algorithms

on history, we can compare the quality using standard error measures. We compute the Mean Absolute Percentage Error (MAPE) which is given by:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n} 100 \times \left| \frac{y_t - \overline{y_t}}{y_t} \right|, \tag{3.4}$$

where y_t and $\overline{y_t}$ are the real and the predicted observations.

Furthermore, for each model we compute the Root Mean Square Error (RMSE) and the Maximum Prediction Error (MAXE) to obtain information about the mean and the maximum error of the prediction. A direct comparison of the benefits for the user by applying each of these algorithms is performed in Sec.3.6, where we compute the cost function defined (Eq. 3.1). In this section, a comparison among the algorithms on the basis of the standard errors is shown.

3.5.1 Data set

The data used in this section are collected via the Mininet network emulator. In particular, a communication between a device and a server occurs in the emulated environment to reproduce the critical traffic in the edge network. To realistically represent the emulated loads over physical links, we set our parameters using real publicly available Internet traces [81]. Using the *TCP iperf3* tool, we replicate the link load over the paths while the source and destination run the interested application. The Mininet topology we adopted consists of 10 switches and 20 hosts where we randomly created links between two switches with the probability of 0.3 [82]. Despite the limitations of an emulated testbed as Mininet, such a trace-based approach allows us to reproduce a quite realistic environment.

We collected a dataset made of more than $50,000$ historical samples. We then split it into training (80%), validation (10%) and test (10%) set, and the error is computed on the test only. The bottom line, however, is that we cannot know for sure which approach results in the best QoE and so it becomes necessary to compare model performance and extensively study methods properties. The framework choose which model to use in light of these findings.

Table 3.1 Time-Series results when the function is fitted for each new observation.

Algorithm	MAPE	RMSE	MAXE
SES	3.12	36.45	678.69
Holt-Winters	2.87	33.17	110.41
ARIMA	2.67	30.73	597.05
SARIMA	3.70	42.83	626.22
LS	3.69	43.39	937.84

Table 3.2 Hyperparameters set in our methods.

Method	Hyperparameters
Linear	–
SVR	cost=1.0, kernel=rbf, epsilon=0.1
Polynomial	degree=4
GBR	n_estimators=500, max_depth=4, learning_rate=0.01
PLSR	n_components=1
DTR	random_state = 1
RFR	n_estimators=70, random_state=2

3.5.2 Algorithms Analysis

We implemented the algorithms exposed in the previous section (Section 3.4) and assess the performance for each one of them. A good predictor should at least outperform a simple algorithm in which the next value is a replica of the Last Sample (LS). This is not considered as a statistical algorithm due to the simplicity of the method, but it is a recommended baseline to compare the quality of the implemented method.

We compare the most popular algorithms in the Machine Learning field, where all experiments are performed using Python implementations of the presented algorithms [83]. In addition, regarding the forecasting horizon, every model has been designed for forecasting with this horizon, since the most common usage scenario is the one-step-ahead prediction.

We define the parameters grid for each method to be searched. At the end of the process, the algorithm is tuned using the optimal set of parameters returned by this

optimization process. For RFR, we define the number of estimators = [10, 50, 70, 100, 200] and random state = [0, 1, 2, None]. The same set of random states is used for DTR as well. Regarding PLSR, the number of components is set to 1, after a study performed on [0, 1, 2, 5] set. For the Polynomial model, the degree refers to the maximum exponents in the function, and we evaluated all the numbers between 2 and 7. The SVR algorithm has more parameters to be set, and we chose cost between 0.7 and 1.0, and epsilon = [0.01, 0.1, 0.5, 1.0]. The kernel value is a string, evaluated among = [rbf, poly, linear]. Finally,for GBR we set the n_estimators the same as for RFR, and learning_rate = [0.01, 0.05, 0.1, 0.5] and max_depth = [2, 3, 4, 5].

To choose the most suitable parameter combination for each method, we perform an initial study of the performance on a validation set. For each method, the parameter combination yielding the higher accuracy is chosen. The resulting parameters for ML algorithms are summarized in Table 3.2.

To choose the best methods to address the user specification, the framework relies on the data shown in Table 3.3 and Table 3.1. The tables summarize the main details about errors and performance. MAPE is used to select the best algorithms, while MAXE to compare the maximum error, useful to understand the routing achievements in Section 3.6. Algorithms like Holt-Winters and DTR do not have the lowest error (MAPE and RMSE) but have a low MAXE. This means they are on average correct and are not far off the real value, even though the predicted value is not too close to the actual one. Routing based on this class of algorithms can achieve excellent results because they can reduce the number of false positive (wrong peak), but it can be hard to detect a true positive (real peak).

Table 3.3 Comparison of error for ML algorithms.

Predictor	MAPE	RMSE	MAXE
LINEAR	2.70	31.15	599.55
POLYNOMIAL	2.66	30.96	590.51
SVR	2.65	30.54	585.61
GBR	2.66	30.68	586.98
PLSR	3.14	37.40	885.59
DTR	3.36	41.16	539.34
RFR	2.91	33.50	580.91

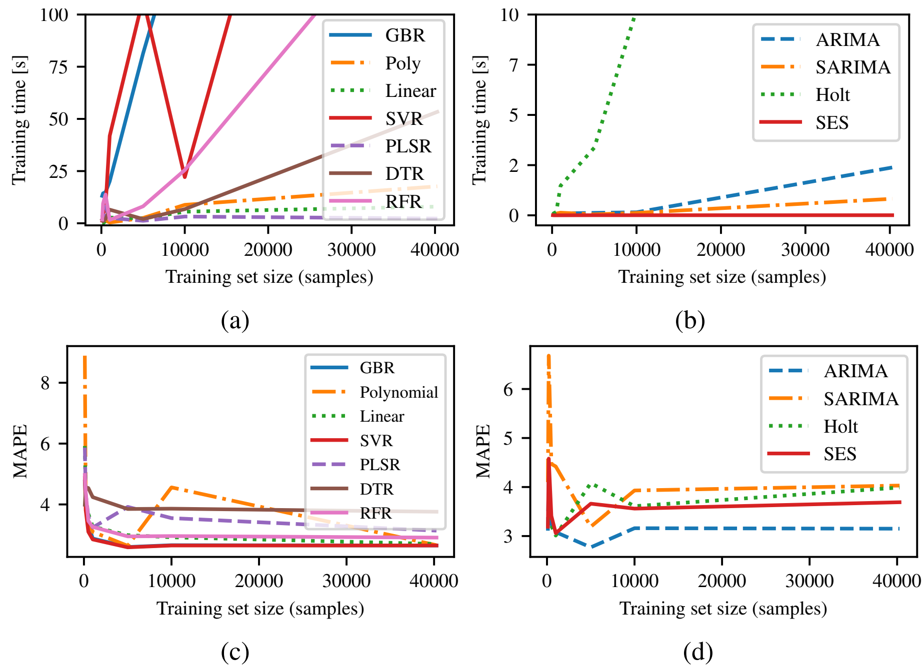


Fig. 3.2 Training time and error (MAPE) for different training set sizes.

Another aspect to be considered is the available time to predict and to train, therefore we study the behavior of the methods for different training set sizes. Figures 3.2a-b show the training time for ML and TS algorithms respectively. As can be noticed, excepted Holt-Winters, TS algorithms take less time to train data. Furthermore, ML training time increases for large datasets of big sizes, with the expectation of the SVR model, whose behaviour is of a considerable training time also for small dataset sizes.

At the same time, training time must be combined with error in the prediction for a comprehensive analysis of the algorithms. Figures 3.2c-d shows MAPE for both the TS and ML models. Clearly the more trained data the lower error, however, it is worth noting that for TS methods the error after a minimum around the size of $1,000$, tends to slightly increase. This result suggests using a small training set for this class. On the other hand, for ML algorithms a general decreasing in the error holds.

These results confirm our hypothesis of training offline ML algorithms on a large data set, and train the TS methods online Holt-Winters is trained online on a small data-set, with no reduction in the error as proved in Figure 3.2d.

For this reason, in RoPE the ML models are trained off-line and then used on-line for predicting. The classical model does not need to be trained off-line, and it is better to use more recent data to predict. In this case, there are two major approaches: the sliding window and the expanding window. In the sliding window approach, one uses a fixed size window for training. On the other hand, the expanding window uses more and more training data, while keeping the testing window size fixed. This approach is particularly useful if there is a limited amount of data to work with. Our choice regarding TS is to marry the two methods: start with the expanding window method and, when the window grows sufficiently large, switch to the sliding window method.

3.6 Routing Evaluation

The goal, as mentioned, is to adapt the routing behavior to better cope with the predicted links conditions. Nowadays many SDN controllers, e.g., Onos, Ryu, OpenDayLight, can obtain a logical view of the network topology. In our testbed Ryu is chosen as SDN controller technology due to its usability and a lighter approach as a python framework for SDN application development: thus, a faster response on flow installation was expected, as confirmed in previous work [84]. In addition, since it is developed in Python, it has many predictors and machine learning libraries readily available.

Firstly, we need to enumerate the cost function weights used in Eq. 3.1 to take into account specific requirements of different scenarios. Considering in particular our three use cases mentioned in Chapter 2, we can observe how throughput is really crucial for a Telepathology session, while it is not so relevant for a Disaster response. For Tactile Internet instead, the latency is the predominant factor. For this reason, for the Telepathology application we used ($\alpha = 10^6$, $\mu = 5 \times 10^{-5}$, $\lambda = 10^{-3}$, $\gamma = 2 \times 10^{-9}$), in the Disaster-response use case we used ($\alpha = 10^6$, $\mu = 10^{-6}$, $\lambda = 10^{-5}$, $\gamma = 10^{-12}$), and to emulate Tactile Internet scenarios we used ($\alpha = 10^3$, $\mu = 10^{-4}$, $\lambda = 10^{-3}$, $\gamma = 10^{-10}$). For the disaster response scenario, we limit the link probability of our Mininet topology to 0.2. In these three applications the client and server run the corresponding programs.

In order to evaluate the feasibility of applications deployed during a disaster, we implemented a program that continuously sends the recorded audio to a server that

processes it and provides useful information such as the presence of humans and the corresponding location.

For the Telepathology use case, where we focus on the achieved bitrate and latency, we deployed an application that sends the video captured by the emulated microscope and sends it to a program responsible for performing video processing [31]. The client sends the video at a maximum bitrate through *ffmpeg*, and we measure whether or not the network can provide the adequate throughput.

Finally, for Tactile applications, we modified the telepathology solution and allowed remote haptic control of machinery instead of the microscope. This robotic plugin is emulated programmatically.

3.6.1 Automate the choice of predictor

As demonstrated, a prediction method can provide optimal results in a number of cases, but might not work properly in other situations. For this reason, we try to automatically choose the algorithm to apply, in order to guarantee the best possible performance. Choosing the right forecasting method for a given use case is a function of many factors, starting from how much historical data are available, and if exogenous variables (*e.g.*, weather, concerts) play a big role. Moreover, we can consider business needs, whether or not the model needs to be understandable. We imagine this is not always necessary, but we may use a classical method to achieve this requirement.

In the context of our Telepathology use case, the choice of the predictors affects the routing performance (Figure 3.3). In particular, the TS and ML methods are considered in Figure 3.3a and Figure 3.3b, respectively. Our results show that RFR achieves a cost of 5.93, the minimum for the MLs, and Holt-Winters a cost of 5.51, the best for both classes. While our results show that the online training phase has a lower cost than the offline counterpart, this is valid for the considered use case but, in other circumstances, the training offline may result as a valuable strategy.

Figure 3.4 demonstrates how the approach is general and can handle different use cases and increasing sizes of the network. In particular, Figure 3.4a shows the cost function value for the three use cases, considering the best TS and ML algorithms for each one. We can see how in a disaster response network, a prediction made by TS algorithms achieves a better transmission quality. This holds because in this

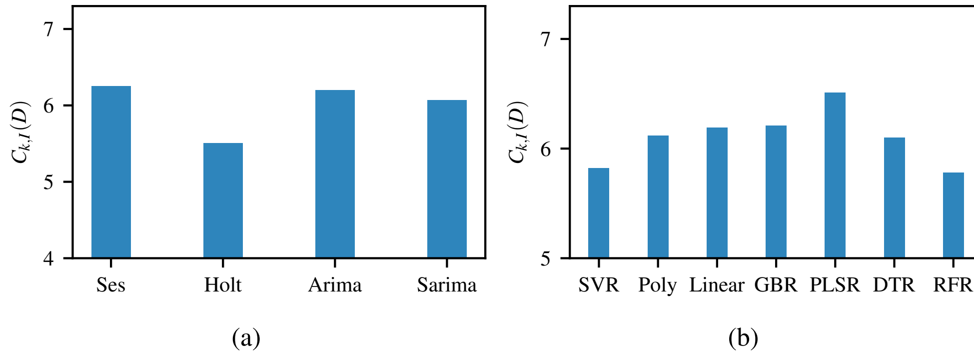


Fig. 3.3 The cost function for the tested algorithms. RFR is the best for ML algorithm, while Holt for TS methods.

scenario fresh data (even if in a small quantity) are more reliable than a huge dataset trained offline, as in ML methods. Conversely, the offline training phase is desirable for tactile Internet applications, where patterns can be discovered in advance and exploited to predict future traffic. This means that, according to user requirements, a class of methods can be preferred to tackle the problem. RoPE is able to detect which class of algorithms to apply and switch among them according to user needs.

In order to generalize our findings, we deployed a more random topology where links among switches and hosts are randomly generated. The number of links between the switches is a parameter in the generation phase and it affects the density of the network. We define the network density, a, as:

$$a = \frac{num_links}{\frac{N \times (N-1)}{2}}, \tag{3.5}$$

where the denominator represents the number of potential links, N is the number of switches and *num_links* is the actual number of links. This value is changed to evaluate scalability and test the performance of the framework. Results in the Telepathology case are depicted in Figure 3.4b, for a different number of links in the network.

On the basis of these findings, the choice of the predictor comprises many factors: use case, expressed as preferences by the user, seasonality of data, frequency in the adaptation of routing, and, consequently, frequency of data collection. Our framework can adequately choose which algorithm to apply, based on the user preferences, for an autonomous network management system. In detail, the choice of

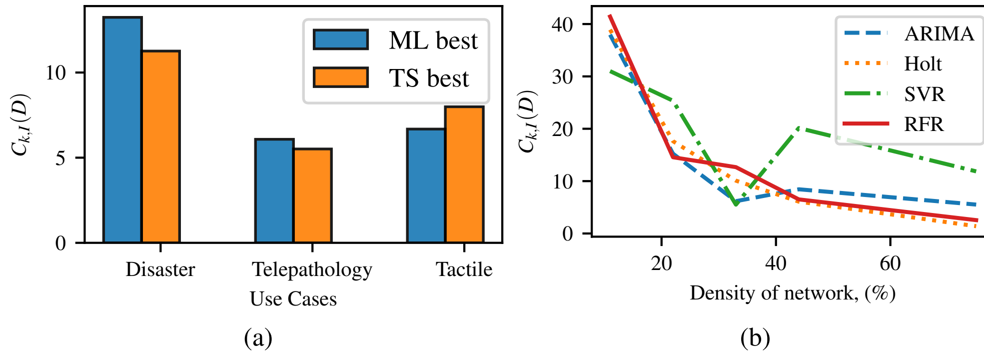


Fig. 3.4 (a) Comparison of different classes of algorithms for different use cases and (b) algorithms performance among different topologies with increasing connectivity.

the best predictor first selects the class (ML or TS) by evaluating the user needs. TS is used by default for its ability to be trained online and providing an understandable model. Instead, in case the application exhibits patterns that a schema can discover offline, ML is preferred. For example, among the three use cases that we evaluated, ML class provides the best results for Tactile Internet, while TS for Telepathology and Disaster Response. However, other cases can be considered as well. Thanks to the generality of the approach, they can be studied by leveraging the general cost function in order to better identify the proper class. The further comparison is distinct for the two classes as follows. *(i)* For the *TS methods*: on one hand, if marked seasonality is denoted, the system selects ARIMA for the best MAPE (Table 3.1). In fact, ARIMA provides a lower MAPE compared to SARIMA and a comparable training time. For both the algorithms we set the training window to $5,000$ values, since MAPE achieves the minimum at this size for the two methods (Figure 3.2d). On the other hand, if there is no seasonality, we then investigate the value of r, and if greater than the default value ($20s$), we select Holt-Winters with the training set size of $1,000$ samples as default predictor. In such a way, we select the more accurate method w.r.t SES, but we limit the training set to reduce the training time to a reasonable value (Figure 3.2b) that can also achieve the best MAPE for this method (Figure 3.2d). When r is lower than the default, we set SES as the preferred option for its lower training time (Figure 3.2b) in order to satisfy the more frequent routing updates. *(ii)* For the *ML methods*: our system sets SVR as the predefined predictor method for its lowest MAPE (Figure 3.2c and Table 3.3). In this case, the size of training data partially affects the accuracy, and, for this reason, we use as much data as available, since SVR minimizes the MAPE on almost any size of the training set.

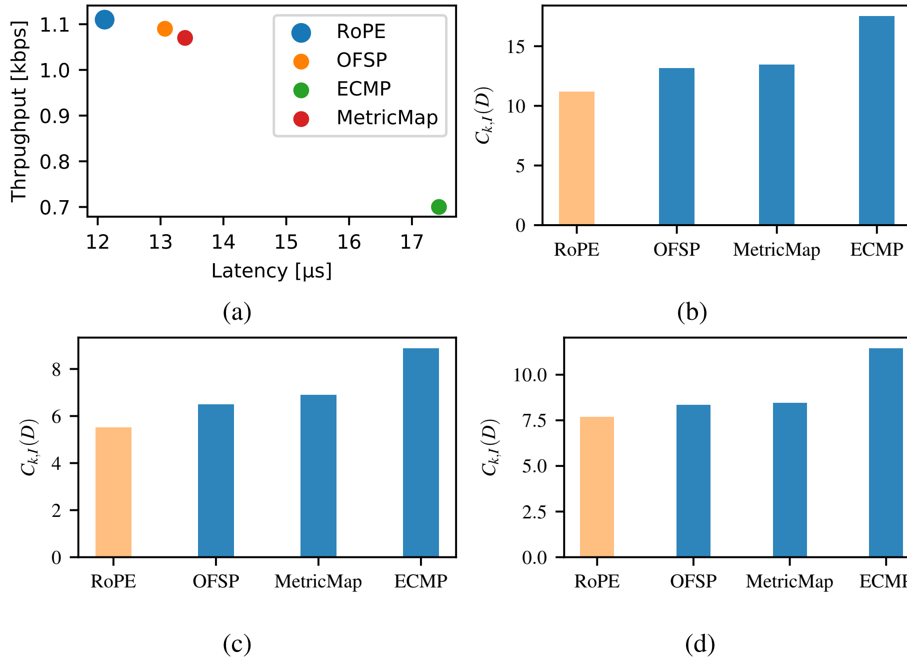


Fig. 3.5 Comparison for different routing strategies. (a) Trade-off between latency and throughput. Our solution provides an higher throughput and a lower latency. (b)-(c)-(d) The effectiveness of the three routing strategies among the three use cases, Disaster, Telepathology and Tactile respectively.

3.6.2 Routing Performance Improvement

In this experimental setup we evaluate the quality improvement by comparing our solution against other currently deployed algorithms (Figure 3.5).

In particular, we compare RoPE against the Equal-Cost-Multi-Path (ECMP), Online Flow Size Prediction (OFSP) [67] and agasint MetricMap [68]. *ECMP*, a well-known algorithm, is used a the baseline. In *OFSP*, authors detect elephant flows by means of the GPR algorithm; hence, the least congested path to route such flows is selected while the ECMP protocol is used to route mice flows. *MetricMap* uses the Very Fast Decision Tree (VFDT) online algorithm [85] to learn and classify traffic. The routing protocol is atop MintRoute and specified for Wireless Networks, but can be generalized.

First, we compare the achieved latency and throughput by using the RFR prediction algorithm for RoPE in Figure 3.5a. From this result, we can state that RoPE reduces the latency while increasing the application throughput, with respect to the other solutions. The result also points out the flaws of a simple yet deployed

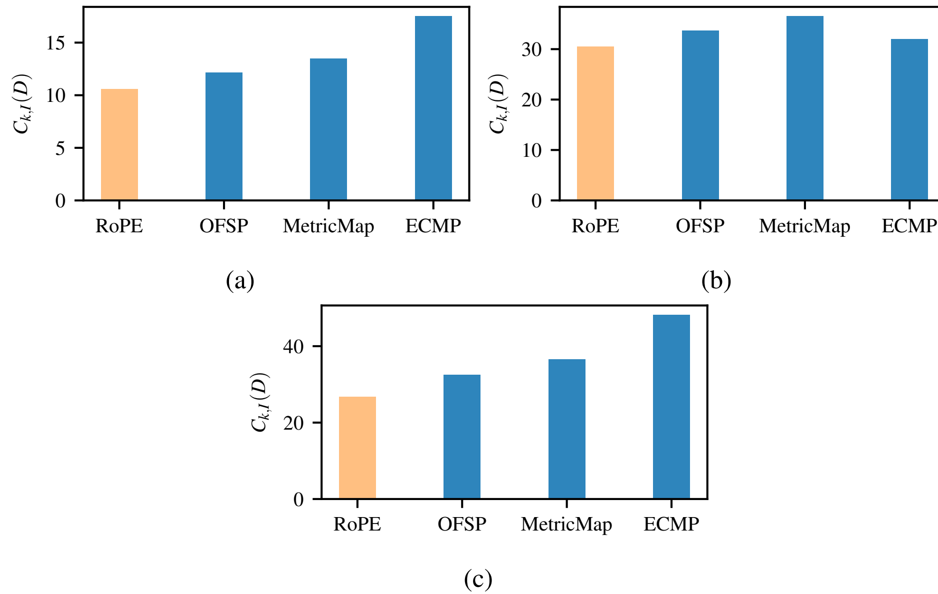


Fig. 3.6 Comparison for different routing strategies over the GENI testbed. Our solution outperforms the related work, as confirmed by (a)-(b)-(c), where the cost function with the proper coefficients is computed for the three use cases, Disaster Scenario, Telepathology and Tactile Internet, respectively.

approach ECMP, highlighting the benefits brought by an adaptive routing combined with SDN.

Although throughput and latency can be considered as the most major metrics to evaluate, we rely on the cost function (Eq. 3.1) for a more general evaluation. Figures 3.5b-c-d depict the function value for the three exposed use cases. As can be seen, RoPE significantly outperforms all the other methods. The resulting routing policy reduced the latency while keeping a stable jitter and high throughput among the three use cases. We can state that our approach yields the best results for the considered applications. We may observe how, while OFSP optimizes the routing for elephant flow that is not long in time, our approach can modify the path even in a second phase, useful for long transmission. Similarly for MetricMap, where online training does not lead to a significantly improved quality.

3.6.3 Real-case Environment on GENI

To establish the practicality of our approach, we test its scalability over the GENI testbed [86], which provides physical machines and physical links for testing pur-

Table 3.4 Performance evaluation in the context of the Disaster Response application running on the GENI virtual network testbed. Even in these real settings RoPE outperforms related solutions.

Solutions	Thr. [kbps]	Lat. [μs]	Jit. [ms]	Δ Jit. [ns]	$C_{K,I}(D)$
RoPE	3929.7	10.48	0.83	9.53	**10.58**
OSFP	4107.1	12.07	1.00	7.22	12.16
MetricMap	4077.1	13.39	1.02	9.03	13.48
ECMP	3702.2	17.43	0.96	4.68	17.52

poses. In particular, we deployed the three applications and the models are re-trained over real-world data following the same procedure exposed in Section 3.5, but the emulated network of Mininet is replaced with physical and virtual links. Based on the previous findings, we select the optimal predictors for each use case and the results are compared against the above state-of-the-art algorithms, as detailed in Figure 3.6. A comparison between Fig. 3.5 and 3.6 shows that conclusions about RoPE in Mininet hold in GENI as well, even though a higher latency and throughput is obtained in real networks. The RoPE cost function is adequately smaller than the state-of-the-art. Moreover, Table 3.4 provides details on each component of the cost function for a Disaster Response scenario. As can be seen, no algorithm outperforms the others in all the adopted metrics, but RoPE achieves excellent results in both the latency and the jitter, which leads to an overall better outcome.

In addition to the evaluation by means of Eq. 3.1, we also consider the requirements of the applications and we check whether or not these are satisfied by RoPE. In Table 3.5 we compare the specific requirements against results achieved by using RoPE on the GENI testbed. We can notice how RoPE brings benefit even from a user perspective, fulfilling the demands of the applications and enabling the deployment of such services.

Tactile applications entail at least 1-ms latency to work appropriately, hence we select an algorithm that best suits such circumstances, with the help of the cost function. In particular, by using SVR as a predictive algorithm, we can satisfy the requirement and guarantee an adequate service.

Similarly, we select Holt-Winters for the Telepathology use case, where we focus on the achieved bitrate and latency. We desire the latency to be lower than 100 ms to

Table 3.5 Application requirements and the satisfiability achieved by using RoPE.

Use Case	Measurement	Required Value	Obtained Value
Tactile	Latency	1 ms	532.274 μs
Telepathology	Video Bitrate	900 kbps	902.45 kbps
	Latency	100 ms	22.76 ms
Disaster	Jitter	1 ms	0.832 ms
	App. Throughput	3 Mbps	3.929 Mbps

assure the real-time control of the microscope. Holt-Winters was chosen based on previous experiments, and it provides excellent results, as proved in Table 3.5.

Finally, we select ARIMA for the Disaster Response use case. The requirements are selected so that Google libraries used to process audios work best and to enable a fast response. The results (Table 3.5) reveal that the use of RoPE ensures the application to function properly.

3.7 Conclusion

This chapter presents RoPE, a new solution to speed up the transfer of critical data. Its main novelty resides in its traffic engineering logic: it predicts the future load on links of a path and then chooses the best one according to data computed. This algorithm allows avoiding congested paths and reduces delay in the transmission, providing a more effective way of routing critical information with respect to other algorithms existing in the literature. The results confirm the impossibility of one prediction algorithm to fit all the use cases. Apparently, Machine Learning provides excellent results, which reduces the latency in critical communications. However, Time Series (TSs) can be used for their fast training phase and the straightforward model. In fact, the results suggest that for Disaster response applications TSs are more appropriate.

RoPE leverages SDN features, e.g., centralized controller and the context-based control path, to collect information about the traffic load on the links and takes a new road in case of predicted congestion. Leveraging SDN switches programmability, the framework can quickly react to excessive predicted load on links and adapts the routing to address the congestion. This framework is intended to overcome well-

known problems related to edge-based applications, such as latency and throughput requirements. Due to the diversity of applications and data generated, RoPE addresses the user needs by autonomously detecting the data properties, selecting the proper model and applying prediction values to the routing.

Moreover, the chapter presents a comprehensive analysis of regression algorithms to evaluate the advantages and disadvantages of the class of methods and depicts the logic behind the presented framework. Possible future work might focus on investigating whether new models can be used in addition to the ones implemented.

Chapter 4

Automatic Network Planning Decisions

4.1 Introduction

As users and traffic demands grow, the need to optimize our communication networks magnifies, denoting the evidence that networks dictate our technological world. Recent advantages in artificial intelligence (AI) and machine learning (ML) are paving the path to autonomous networks: networks that measure, analyze and control themselves autonomously [12]. Network automation has been desired in the last years, since it is almost impossible for human operators to render real-time network management [87, 35].

Our focus in this system is on network reliability and network elasticity, *i.e.*, the subproblem of autonomous networks that deals with the ability to auto-scale resources up and down, in harmony with changes in the environment, *e.g.*, traffic demand. The advantages brought by the auto-scaling techniques are multiple. They reduce the cost of resource management, by deactivating resources that may increase unnecessary (energy) costs. At the same time, the network can provide redundant facilities to reroute traffic when workload peaks to unexpected levels.

The work presented in this chapter has been partially published in [21].

As networks are becoming more programmable and virtualized, their complexity also increases, with the consequence that exploiting the offered programmability to guarantee high availability is a non-trivial task.

As networks are becoming more programmable and virtualized, their complexity also increases, with the consequence that exploiting the offered programmability to guarantee high availability is a non-trivial task. Traditional threshold-based and recent ML-based auto-scaling policies are often unable to address the high complexity of networks and consequently to satisfy carrier-grade requirements such as reliability and stability. Furthermore, state-of-the-art solutions hardly combine these features altogether, such as [88] whose primary goal is the energy efficiency, or [89], which automatically scales Virtual Network Function instances via an ML classifier. Although reinforcement learning is emerging as a valuable technique to solve many networking problems, as in [90, 91], there is no solution incorporating network information to automatically and efficiently orchestrate network resources in a decentralized manner.

To this end, we propose Mystique, a network management schema that, using Multi-Agent Reinforcement Learning (MARL), auto-scales to accommodate the traffic demand and reacts to possible failures. On the one hand, Mystique unburdens network nodes that are over-congested with traffic, to preserve the high bandwidth and high availability of the applications. On the other hand, it leverages healing strategies [92] to repair failing nodes and links.

Each MARL agent, a process running within a network controller, can learn an auto-scaling policy from experience, without any *a priori* knowledge or human intervention. By continuously monitoring the state of the network, the agent can make sharp decisions on how to optimize network performance and users' experience, exploiting SDN to promptly change the configuration. Moreover, the distributed nature of MARL makes it possible to exploit a (possibly) large number of SDN switches spread across the topology as probes. The system automatically re-balances both existing and new flows across nodes, while the agents communicate among them to obtain information about the other sub-network.

At the same time, it is well-known that from the operator's point of view, Quality of Experience (QoE) is an important aspect in keeping customers satisfied, and thus decreasing churn [93]. To this end, the decisions taken by Mystique aim to maximize

overall QoE across multiple users and achieve a desired level of QoE fairness, while reducing the energy costs for active links and nodes.

Results validate our decentralized control plane, showing how Mystique can promptly adapt and modify its behavior to handle variations in workloads. Compared to other benchmark solutions, our algorithm can jointly improve the user satisfaction and more wisely utilize the network resources.

4.2 Related Work

Managing network or application resources elasticity implies a first mapping performance requirements to the underlying available resources. Such a process of adapting resources to the on-demand requirements of an application, called scaling, can be very challenging. Resource under-provisioning will inevitably hurt performance and generates QoS violations, while resource over-provisioning can result in idle instances, thereby incurring unnecessary costs. Auto-scaling techniques, *i.e.*, resource allocation strategies that automatically scale resources according to demand, are more than a need and can be differentiated into two classes: *reactive* and *proactive*. While the former class refers to algorithms reacting to system changes, but not anticipating them, the latter stands for strategies that predict and anticipate the future needs and consequently acquire or release resources in advance, to have them ready when they are needed. In the literature, auto-scaling solutions have been extensively discussed from several points of view, especially for cloud computing [94–96].

Reactive and proactive techniques. Threshold-based policy is a common example belonging to the reactive category, whereas time-series analysis, reinforcement learning, queuing theory, and control theory can be examples of proactive approaches. Queuing theory, given its ability of estimating performance metrics such as the queue length or the average waiting time for requests, has been largely applied to model applications, e.g., general Internet or cloud infrastructure applications [90, 97, 98]. For example, [97] solves an optimization problem by distributing servers among different applications, while maximizing the revenue. The authors characterize the arrival process of requests to an application using a real trace of an e-commerce system, where the arrival process is adequately described by a Poisson process.

Control theory has been applied to automate the management of web server systems, data centers/server clusters, storage systems, cloud computing platforms, and other systems, showing interesting results across this variety of systems. Many papers have discussed adaptive control techniques, by adjusting the controller tuning parameters online [99–102]. For instance, [101] combines two proactive adaptive controllers for scaling down with dynamic gain parameters based on the input workload, and a reactive approach for scaling up.

Time-series are massively used in finance and economic domains to represent the change of a measurement over time. Recently, this technique has also gained attention in engineering and workload or resource usage prediction problems [20]. At the very basic, a time-series is a sequence of data points, e.g., number of requests that reaches an application, measured at successive time instants spaced at uniform time intervals, e.g., one-minute intervals. The time-series analysis is able to find repeating patterns in the input workload and to forecast future values. The auto-regression method has been largely used [103–107] and time-series forecasting can be combined with reactive techniques [108]. For example, [106] proposed a hybrid scaling technique that, based on CPU usage, utilizes reactive rules for scaling up and a regression-based approach for scaling down.

Lastly, reinforcement learning (RL) approaches for dynamic resource allocation problems were successfully applied in the literature. RL can well fit auto-scaling problems by online capturing the performance model of a target application and its policy without any *a priori* knowledge. However, these methods have mainly focused on allocating tasks, services, and Virtual Machines (VMs), especially to face the greater or smaller demand, where [109, 90, 110] are examples of a profitable usage of RL. As such, little work has been proposed to address the problem of network resources.

Dynamic Resource Creation of Network Agents. Recent studies have explored scaling softwerized or virtualized network functions in telco and cloud networks. Among them, [89] proposes a proactive ML-based approach to perform auto-scaling of VNFs in response to dynamic traffic changes. The classifier learns from historic auto-scaling decisions and measured network loads, and outputs the number of VNF instances required to serve future traffic without violating Quality of Service (QoS) requirements and deploying unnecessary VNF instances. [88] describes ElasticTree, a network-wide energy optimizer that continuously monitors data center traffic

conditions and chooses the set of network elements that must stay active to meet performance and fault tolerance goal. To decide which subset of links and switches to use, a fast heuristic is used. The primary goal of ElasticTree is the savage of energy in data centers containing thousands of nodes. Although we share the general approach with these solutions, we propose a self-learning model that embraces more the QoS aspects.

A reinforcement learning approach is described in [91], where the authors present SRSA, a resource-efficient approach to auto-scale telco-cloud. The decision of allocating or de-allocating VMs is performed to guarantee the QoS and to reduce the cloud cost. Our solution is also built upon an RL framework, but differs in the modeling aspects and enables us to scale to more complex networks by learning in a distributed fashion.

4.3 System Design

In this section, we first identify the softwarized infrastructure and the advantages of auto-scaling solutions, like Mystique, in this scenario. Then, we present the mechanisms underpinning the overall system with particular focus on the offered features. In particular, we start analyzing a single entity, and we continue highlighting the elements employed to realize the parallelization of the model.

4.3.1 Edge Network Scenario

Although our proposed model is general enough to suit several network deployments, in this manuscript we target edge networks built from software-defined networking (SDN) architectures, due to their flexibility and possible customization. While the model is general and does not require specific functionalities offered by the underlying technology, to achieve automatic responses to network traffic, softwarized or virtualized networks are needed.

By edge network, we refer to a network located on the periphery of a centralized network. It sits entirely between the services and the endpoint devices using them, as well as between all the edge servers themselves.

In this context, network operations can take advantage of data-driven, machine-learning-based models to achieve more high-level goals and a holistic view of the underlying network. Our solution constitutes an attempt towards a fully automated network, often referred to as *self-driving* network. Self-driving networks can measure, analyze, and control themselves in an automated manner, reacting to changes in the environment, e.g., demand, while adjusting and optimizing themselves as needed [12]. This idea has been around in a variety of shapes, such as self-organized networks [111], cognitive networking [112], knowledge-defined networks [113] and data-driven networking [114], and lastly, self-driving networks [115].

Fig. 4.1 represents a common scenario that needs auto-scaling components. At first, the traffic demand increases reaching a non-tolerable level of congestion, and the system decides the consequent creation of resources to satisfy the traffic demand alongside re-routing for the interested flows. In a second time, the exigency of additional resources vanishes gradually as the traffic decreases, and the system reacts by removing the unnecessary devices. This simple example illustrates the feature and benefits of auto-scaling networks.

In this regard, recent advances in machine learning (ML), e.g., deep learning, and networking, e.g., SDN, programmable data planes, and edge computing, have fostered the development of these networks. However, a desirable and still missing feature is represented by the distributed detection of congestion with no centralized congestion recognition and control. Furthermore, to improve system performance, bottlenecks need to be identified, and efforts should be invested in alleviating these bottlenecks.

4.3.2 System Components

The main functions of Mystique are to auto-scale according to the traffic demand and react to failures when they occur. We developed and implemented these features in a system depicted in Fig. 4.2. In this context, the controller monitors the state of each switch in its sub-network to detect if one of the following events occurs: the switch is overloaded (congestion), the switch in under-utilized and can be deleted (cost-saving), the switch fails and the connectivity can be no longer guaranteed (failure). However, the network implements the control plane with several distributed controllers. Each of them controls a subset of switches and communicates with the

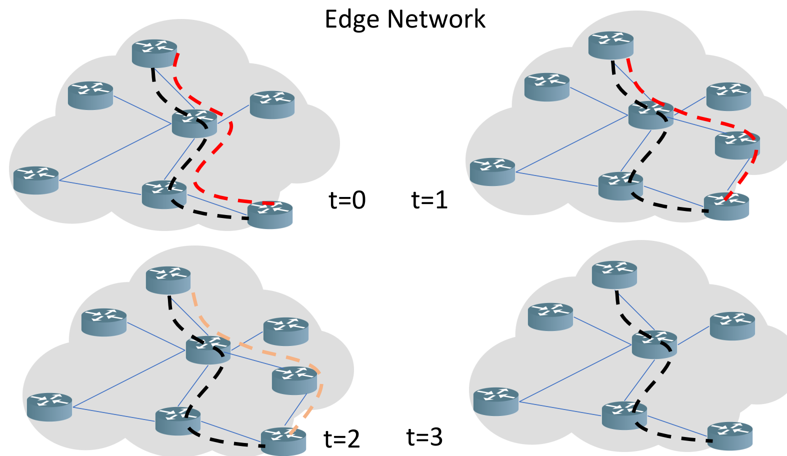


Fig. 4.1 The goal of our Mystique system is to learn via reinforcement learning how to adapt to network demand fluctuations by creating new virtual switches and split traffic (from t=0 to t=1), and to scale-down nodes and network resources when demands decrease (from t=2 to t=3).

other controllers, via the *Info exchange* process, to obtain a consistent network view. For any change in the controlled network region, e.g., new link, the controller notifies its peers. They also exchange the information required for computing the QoE for connected users.

The reinforcement learning (RL) module selects the best action, *i.e.*, active network resources, but interacts with other processes to collect the information required for the decision and to notify about the outcome. In fact, we avail multiple processes to better separate concerns, but they cooperate to achieve our stated goal. The main functionalities are summarized thereafter.

Routing. Each agent dynamically creates and destroys virtual switches and virtual links in response to network fault or substantial network traffic changes. This means that, in these events, the agent is also responsible for re-steering the traffic and deciding what flows to move in response to these actions.

At the beginning, the route for each flow is selected by the controller based on the shortest path algorithm. In the case of multiple available paths between source and destination, a load balancing strategy is applied, *i.e.*, flows are equally distributed among the multiple paths. In the following, we separate the events to face during the execution with the aim of a more clear presentation.

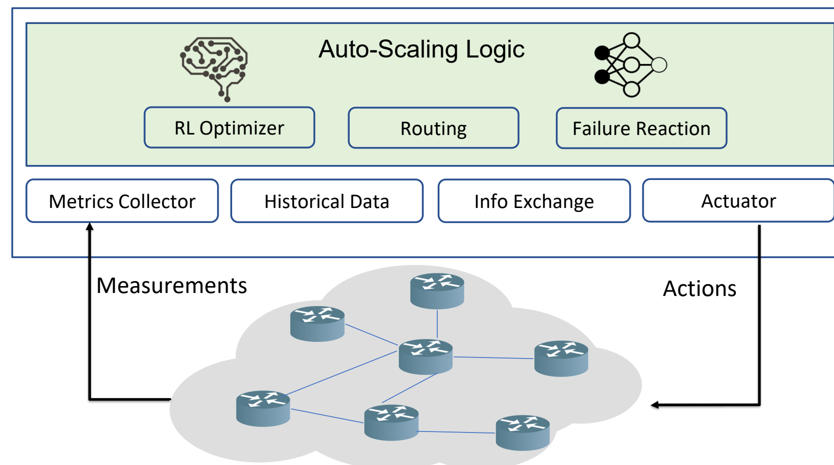


Fig. 4.2 System overview. The Software-Defined Network controller receives as input traffic statistics and outputs new flow routes and power on/off commands.

In the case of a *link or node failure*, the same resource is re-created. For a link failure, a new edge is created connecting the same source node and destination node. The neighbor of the switch modifies the forwarding rules reflecting the new port ids. For a switch failure, a new one is generated with the same links that the faulty switch had. This implies that all the flows previously installed on the old switch are moved to the new one, and the neighbor nodes that were connected need to be re-instructed with the new ports.

When a *new link or switch is created*, the topology is analyzed, and the flows that can take advantage of the new path(s) are identified. Among them, a subset of flows, *i.e.*, in order to select half of the identified traffic, is transferred to the new alternative path. However, we remark that the number of flows to move is a consequence of a load balancing strategy that attempts to equally redistribute flows.

Finally, when *a link or switch is deactivated* due to energy saving considerations, all the flows traversing the deactivated item are considered, and a new path for each of them is set via the shortest path strategy.

Failure Reaction. We desire to react accordingly to the degree of the issue and take a proportional action. For this reason, the utilization of the switch (and connected links) is handled by the RL model, while a separate module manages the failure detection and reaction. Inspired by previous work [116, 117], we consider 5 possible faults that can take place in our scenario: (1) communication with the controller ended, (2) timeout of the response expired, (3) port fault, (4) flows of a particular

host have blocked unexpectedly by the switch, (5) unexpected behavior of the switch. As the controller continuously monitors the state of the switches, it can replace the switch in case of (1)-(2) (three consecutive timeout expirations)-(5). Instead, in case of (3)-(4), the link originating from the fault port is re-created.

The mechanism of fault reaction is in addition to the fast failover provided by new versions of OpenFlow [118]. In this procedure, it is possible to install rules whose forwarding behavior depends on the local state of the switch. Hence, it allows fast failure recovery, as long as the SDN controller is able to anticipate every possible failure and precompute appropriate backup paths. However, OpenFlow fast failover is just to react to link failures, and no other events are taken into account, for example, switch failure. Even though this is equivalent to a failure of all the adjacent links, we argue that the controller can benefit from our model and adapt the routing and the application logic dynamically, as the network evolves. The fast failover is orthogonal to a reactive solution, as our model is. For this reason, both strategies are utilized for an improved fault reaction.

RL Optimizer. The optimizer's role is to find the network subset that satisfies current traffic conditions while avoiding the waste of resources. As input, it receives the topology, the power model of switches, and the current traffic conditions. These measurements are collected by the switches and reported periodically to the network controller, where resides the *metric collector* component. The collected data then feed the model on the agent, that outputs the best decision for the network itself, exploiting *historical data* to learn the goodness of a particular action upon the occurrence of similar conditions. When the decision is made, the *actuator* receives the output consisting of the set of active components and performs the appropriate commands. Moreover, the actuator also pushes the new routes into the network.

4.4 Auto-Scaling System Model

The ultimate goal of Mystique is to deploy the network resources in order to balance the management costs and goodness in application performance. In this section, we first describe the variables used in the following to specify the model; then, we formalize our problem.

4.4.1 Network Model and Preliminary Definitions

We define our model in two phases: the overseeing phase, the healing phase. In the overseeing phase, a network controller monitors the system to check for failures or congestion conditions. Specifically, it communicates with the nodes of the network to obtain information about the load on links attached. Given the context of an SDN topology, a network node commonly denotes a switch. However, our model is independent of the layer the device operates and can work when the nodes act as either routers or switches.

When a link (and implicitly its connecting nodes) becomes or is going to be too congested, the controller intervenes to mitigate the congestion. In the healing phase, we determine the quantity of new resources to create, given the intolerable network congestion. Hence, enough nodes and links are created, and traffic is redirected to them to ensure that the fairness index is close to 1 in the network system.

Aside from the congestion detection, another goal of our system is to react to the failure of a network resource (node or link). In such a case, the response is similar, with the creation of one or more replacements. Moreover, a re-routing process occurs to notify the nodes of the existence of the new resources. After the actuation, a failure and congestion avoidance phase starts again, and the controller will continue to monitor the system to guarantee that traffic is returned to normal operation.

We formalize such phases using the following notation. Let the network be modeled by means of a graph $G = (V,E)$, where V is the set of vertices (nodes of the network), and E is the set of edges, standing for the links. Similarly to the multi-commodity flow problem, we assume that in the system there are k commodities $K_1, K_2, \ldots, K_k$, defined by $K_i = (s_i, d_i)$, where s_i and d_i are the commodity i source and destination. The flow of commodity i has an end-to-end throughput defined as f_i. Moreover, the throughput of flow i along the link (u,v) is $f_i(u,v)$ where each link of the network has a fixed capacity $c(u,v)$.

Let $L_{u,v}$ be the load on the link (u,v), computed as follows:

$$L_{u,v} = \frac{\sum_i^k f_i(u,v)}{c(u,v)}, \tag{4.1}$$

considering that $f_i(u,v) = 0$ when flow i does not traverse the link (u,v).

QoE. As other user-centric services and application management [119], we focus on the quality of experience (QoE) perceived by the end-user. It is generally accepted that the degree or annoyance of a user of a networked service depends, in a non-trivial and often non-linear way, on the network's QoS [93]. Furthermore, the QoE is service-specific and is often different given the same network conditions, *i.e.*, the way in which QoS can be mapped to QoE depends on the service offered.

For this reason, we define a general mapping function B to compute the QoE value y_i for the user i, given the QoS parameters in the set x_i:

$$B : x_i \mapsto y_i = B(x_i) \in [L;H], \tag{4.2}$$

where L is the lower bound and H the upper bound of the QoE value. The function B is required to map QoS parameters into the QoE domain $[L;H]$ and map QoS to QoE uniquely. Thus, it does not need to be monotonic. We leave the choice of the mapping function out of the scope of this chapter, since the QoE models are often derived by subjective user studies. In our formalization we only require the value $y_i = B(x_i)$ to be normalized in $[0;1]$. If B does not naturally return values in the range $[0;1]$, this can be achieved normalizing the QoE values $y_i^* = \frac{y_i - L}{H - L}$.

QoE Fairness. Recent results [120] have argued for a more informative notion of fairness, not limited to flows as classically studied in TCP, but more holistically over a set of metrics defining the QoE. We adopt the same definition of QoE fairness index F as follows:

$$F = 1 - \frac{\sigma}{\sigma_{max}} = 1 - \frac{2\sigma}{H - L}, \tag{4.3}$$

where σ is the standard deviation of the QoE values and quantifies the dispersion of the users' QoE in a system. The fairness index F is a linear transformation of the standard deviation σ of y_i to $[0;1]$, where $F = 1$ indicates perfect fairness and hold for minimum standard deviation ($\sigma = 0$). Conversely, $F = 0$ denotes the minimum fairness and is found when the standard deviation is at its maximum. The observed σ is normalized with the maximal standard deviation σ_{max} and specifies the degree of unfairness. Further, the maximum standard deviation is $\sigma_{max} = \frac{1}{2}(H - L)$, and we can observe that in our case where $y_i \in [0;1]$, $\sigma_{max} = 0.5$. In conclusion, a system is absolutely QoE fair when all users receive the same QoE value.

This definition of fairness appears to be intuitive, *i.e.*, high value if fair and low value if unfair, and compared to the most frequently used metric of Jain's fairness

index [121] provides some additional properties. First, the index F is independent of QoE level (QoE level independence), and second, only depends on the absolute value of the deviation from the mean value, not whether it is positive or negative (deviation symmetric). The Jain's fairness index, instead, is sensitive to the used rating scale. These features ensure our model to be general enough and valid for multi-applications. We can also note that, as the F index is defined, QoE fairness says nothing about how good the system is and thus needs to be considered together with the achieved QoE in system design.

4.4.2 Optimizing Quality of Experience, Costs, and Fairness

Based on the aforementioned network model, we now describe the problem as an optimization problem aiming to simultaneously reduce management costs, alleviate congestion effects providing an adequate service, and ensure fairness among users.

First, we formalize the power consumption of network topology as:

$$C = \sum_{(u,v)\in E} cl(u,v)\phi_{u,v} + \sum_{u\in V} cs(u)\tau_u, \tag{4.4}$$

where $\phi_{u,v}$ is a binary decision indicating the power status of link (u,v), *i.e.*, $\phi_{u,v} = 0$ refers to power off and $\phi_{u,v} = 1$ to power on. The same for the power status of a switch, where τ_u is a binary variable indicating if switch u is powered on. Besides, $cl(u,v)$ and $cs(u)$ are the power cost for link (u,v) and switch u respectively.

To optimize the power consumption, we act on these binary variables for every link and switch, and we constrain traffic to only active links and switches. We can now present the overall problem as follows:

$$
\begin{aligned}
\underset{X,Y}{\text{maximize}} \quad & \mu \frac{1}{k} \sum_{i=1}^{k} B(x_i) + \lambda F - \omega C_{norm} && (4.5) \\
\text{s.t.} \quad & \sum_{i=1}^{k} f_i(u,v) \le c(u,v)\phi_{u,v}; \forall (u,v) \in E && (4.6) \\
& \tau_u \le \sum_{r \in R_u} \phi_{u,r}; \forall u \in V && (4.7) \\
& \phi_{u,r} \le \tau_u; \forall u \in V, \forall r \in R_u && (4.8) \\
& \sum_{r \in R_u} Z_i(u,r) \le 2; \forall u \in V, \forall r \in R_u && (4.9) \\
& \phi_{u,v} = \phi_{v,u}; \forall (u,v) \in E, && (4.10)
\end{aligned}
$$

where C_{norm} is the normalized cost, *i.e.*, $0 \le C_{norm} \le 1$, to make it comparable to the other variables. Line (4.5) specifies the objective function, which attempts to maximize the average QoE and the fairness index F perceived by the end-user while reducing the total network power consumption. μ, λ, and ω are three parameters that balance the importance of power consumption with respect to the QoE and fairness index. By tuning these coefficients, the model can be tailored to specific requirements.

The constraints from (4.6) to (4.10) include some requirements to satisfy. In particular, (4.6) ensures that deactivated links have no traffic. Each flow is indeed restricted to the links powered on, *i.e.*, for which $\phi_{u,v} = 1$. Therefore, for all links (u,v) used by commodity i, $f_i(u,v) = 0$ when $\phi_{u,v} = 0$. The objective of cost minimization enforces also the opposite: links with no traffic can be turned off. This line also imposes capacity constraints, as the total flow along each link must not exceed the link capacity. Further, (4.7) and (4.8) set a correlation between the link and the switch decision variable. Specifically, (4.7) imposes that when all links connected to a switch are off, the switch can be powered off. Similarly, when a switch is powered off, all the links connected to such a switch are also powered off, as stated by (4.8). Although splitting the flow across multiple links in the topology might reduce power by improving link utilization overall, it is known that this may cause undesirable packet reordering effects that negatively impact TCP performance [122]. For this reason, we prevent flow from getting split in the above problem by enforcing (4.9). This constraint ensures that the switch receives flow i from the incoming link

and forward it to only one outgoing link, and the flow uses a total of at most two links attached to the switch u. Finally, in (4.10) we define that the link power status is bidirectional, and there is no concept of half-on Ethernet link. Thus, the full power cost for an Ethernet link is incurred when traffic flows in either direction.

Our optimal solution must select which resources to turn on and off, while satisfying the traffic constraints. The presence of binary variables makes the stated problem a mixed-integer linear program, which is NP-complete. Given the computation complexity, which can hardly scale to networks with a large number of nodes, in the following we attempt to solve the problem via a reinforcement learning approach.

4.5 The Mystique Solution

The learning process at the basis of the system is built upon the reinforcement learning concepts for a continuous acquisition of knowledge of the network. In the following, we define the main elements which characterize our reinforcement learning problem. Since our MARL model can be viewed as an extension of centralized model, in the following, we describe the procedure as in a single agent variant in order to clearly describe the learning process.

Reinforcement learning is a well-known on-line technique that approximates the conventional optimal control technique known as dynamic programming [123]. The external world is commonly modeled as a discrete-time, finite-state, Markov Decision Process (MDP). The agent interacts with the external world and performs actions, where each action is associated with a reward. The objective of reinforcement learning is to maximize the long-term discounted reward per action. In our solution, each reinforcement learning agent uses the one-step *Q-learning* algorithm [124]. In this context, the learned decision policy depends on the state-action value function Q, which estimates long-term discounted rewards for each state-action pair. Given a current state x and the possible available actions a_i, the agent selects each action with a *softmax* policy, which consists of a softmax function [125] that converts output to a distribution of probabilities.

Hence, the RL agent receives the inputs and selects the best action, then updates the state of the table and proceed with this process continuously.

4.5.1 Reinforcement Learning States

Decisions taken by the agent should consider the network status and react properly to events that occur. Hence, the state of our RL model is composed of the load on links, $L_{u,v}$, for each link in the network. This metric is employed in the learning process to choose the best action and evaluate the performed action.

4.5.2 Actions

The RL algorithm defines the mapping between the inputs of the reaction logic, *i.e.*, the network state, and the actions to be performed to address the issues coming from the ongoing traffic. The action selection is the key to the proposed algorithm to be effective. As in the other RL approaches, the control decisions are learned from experience, eliminating the burden of a more rigorous model. In line with the optimization problem defined, the RL agent computes a scaling action $a_t^{u,v} \in \mathscr{A} = \{1,0\}$, for the link $(u,v) \in E$ at time step t. As mentioned above, $a_t^{u,v} = 1$ represents the link in the state on, and $a_t^{u,v} = 0$ represents the link in the state off. When the decision for the link differs from the current link state, the controller sends specific commands to change the state and actuate the output of the RL process.

In conformity with the cost optimization and the considerations previously explained (Section 4.4.2), when all the links of a switch are down, the switch can be turned down as well. For this reason, in spite of the fact that the actions only refer to links, they also impact the switch state.

4.5.3 Reward

In accordance with the reinforcement learning approach, the agent finds the best resource allocation that maximizes the network-aware reward. In fact, the *reward* of the RL formulation specifies the appropriateness of the action taken in a particular state. The utility function instead specifies the objective of our algorithm by looking at the environment response. The aim is to find the decision policy of resource allocation with the maximum utility function for the network agent. Since the utility function is the real objective that we try to optimize, we directly use it as a *reward* for

the learning process. Thus, we define a reward function, r, similar to the optimization function of the aforementioned formal problem:

$$r = \mu \frac{1}{k} \sum_{i=1}^{k} B(x_i) + \lambda F - \omega C_{norm}, \tag{4.11}$$

where μ, λ and ω are coefficients that control the importance of the average $B(x)$, which defines the QoE and the customer satisfaction, F, for the fairness of the QoE, and C_{norm}, that assess the network cost management. The function aims to minimize the over-provisioned resources while also minimizing the switches load in order to improve QoS parameters.

4.5.4 Multi-Agent Reinforcement Learning Framework

To scale-up and distribute the burden among separate agents, we leverage multiple controllers, where each of them is responsible for a sub-network. Along with improvements in terms of scalability, distributing the burdens among more agents brings resiliency to the event of failure of one controller. When one agent fails, the switches under its control are temporarily migrated under the supervision of a near controller. As such, our solutions can handle both failures at the data-plane (switches) and at the control-plane (controllers).

More formally, in a multi-agent setting, each agent maintains an individual policy π_i for the specific state space $\mathscr{S}_i$ and action space $\mathscr{A}_i$. The state space is, thus, limited to the sub-network managed, and includes a diverse set of switches and links. In the same way, the action set reflects the diverse set and is restricted to only the managed links.

Despite the difference in the RL model, the agents interact among them to obtain information about the global topology, as the routing decisions should consider a global view. A model is trained for each node, but they communicates possible metrics needed to compute the user QoE.

4.5.5 Mystique Overall Algorithm

In the light of all these elements, we propose a decentralized RL-based procedure to address the congestion problem in the network.

Algorithm 2 Resource optimization using RL

1: Let P be metrics collection interval, and I the learning step
2: Let Q be the network approximating the Q-values
3: Let TN be the target network
4: Initialize $Q(s,a) = 0, T_1 = 0, T_2 = 0$
5: **for** each episode **do**
6:     **if** $t - T_1 > P$ **then**
7:         Collect the state s at time t
8:         Verify the presence of failures, and in case react
9:         Choose a using policy derived from Q
10:         Take action a by activating/deactivating selected resources
11:         Adjust routing accordingly
12:         Observe r, s' and update the Q network
13:         **if** $t - T_2 > I$ **then**
14:             $TN \leftarrow Q$
15:             $T_2 \leftarrow t$
16:         $T_1 \leftarrow t$
17:         Notify updates to other controllers

In Algorithm 2 we summarize the main steps underpinning our self-learning process aboard of each controller. For one thing, we initialize the Q-values table, $Q(s,a)$ and other hyper-parameters. Next the continuous learning procedure starts (line 5). With a period P, the controller gets information about the congestion of the links, represented by the value $L_{u,v}$. Hence, the agent observes the current state, s, and verifies the absence of failures. In case of link or node failure the proper reaction is enacted. Either way, an action a is chosen for that state based on one of the action selection policies explained previously. Taking the action may involve creation or removal of links or nodes. In these circumstances, re-routing must take place, and the controller instructs the switches with the new flow rules to engineer the traffic. Once the adjustment is completed, the agent observes the reward, r, as well as the new state, s', and updates Q-value for the state s using r and the maximum reward possible for s'. The updating is done according to the formula and parameters described of Q-learning. Afterwards, it is checked if I seconds are elapsed since last

update, if so the Q network is copied to the target network TN. Finally, the agent set the state to the new state, and repeats the process until a terminal state is reached.

4.6 Experimental Settings

In this section, we first describe the testbed configuration and the settings common throughout our experimental campaign. Further, we identify a representative network scenario for the evaluation and the algorithms to compare Mystique against.

4.6.1 Implementation

To demonstrate the practicality of our approach, we have implemented the proposed scheme in an emulated scenario. Mystique needs to receive the current utilization and to operate on the flow paths decisions. These network capabilities can be achieved via NetFlow [41] and SNMP for the traffic data, while source routing and policy-based routing allow the path control. The network switches can be implemented via different technologies, such as Quagga, FRRouting, OVS, Bind, P4. However, due to the ease of use of SDN in prototyping, we use OVS switches [42] featuring OpenFlow, combined with an SDN-controller to obtain the above requisites. OpenFlow, providing a flow table abstraction, is used to push the computed set of application-level routes to each switch. Besides the flow installation component, it provides the flow-specific counters that can be accessed by external entities via open API, and enables the port power control.

Every switch communicates with an SDN controller, which is implemented with Ryu [126]. Ryu is a component-based software-defined networking framework that provides network visibility and control atop a network of OpenFlow switches. In particular, the Ryu controller communicates with the switches to collect the metrics, and pushes the adjusted flow routes. The controller behavior can be customized via the REST APIs that the controller is equipped with. Finally, in the centralized version, the number of Ryu controllers is limited to one, whereas in the distributed version, the number can increase to guarantee fault tolerance of the controller agent.

4.6.2 Experimental Setup

We deployed several environments to assess the performance of Mystique. However, herein we summarize some common settings and remarks valid throughout the performed experiments. For the sake of simplicity, we limited the QoS parameters x_i of user i to the end-to-end throughput, and the mapping function B is an identity function. By doing so, we limit our focus to metrics that can be collected smoothly, and we leave the study of a system based on more indicators (to be summarized for example using a neural network) as future work.

Evaluation settings. The topology and the switches are emulated over Mininet [127]. Mininet is a network emulator that creates a realistic virtual network, running real kernel, switch and application code, on a single machine. This networking tool makes use of namespaces, a feature of the Linux kernel that partitions kernel resources. In current setup, switches and ports are not powered off, but are only deactivated, since the switches are virtual switches. Nonetheless, in a real deployment, it is possible to exploit existing mechanisms to control the power, such as SNMP, command line interface, or recent mechanisms of power control over OpenFlow.

Traffic workload. The energy, performance, and robustness of the system heavily depend on the traffic pattern. In the following, we explore how a variety of communication patterns affect system behavior. Specifically, we evaluate a *uniform demand*, where every host sends one flow to another host of the network. We use two more types of traffic patterns to evaluate performance, *moderate increase* and *sharp increase*. During the former, the hosts linearly increase the traffic sent to double it within 20 seconds. In the latter, instead, traffic is doubled in 5 seconds. Finally, due to the lack of public traces specific for this problem, we generate traffic *synthetically*, where each sender-receiver pair runs *TCP iperf3* for 100 seconds, alternating between different rates.

Hyper-parameter settings. We developed the agent's neural network with Keras [128], a high-level neural networks API written in Python. The neural network is composed of the input layer with three hidden layers with respectively 256, 128 and 64 neurons. The input layer number of neurons corresponds to the number of links in the managed network, whereas the number of output layer neurons is the amount of links of action set, E^a. Such a neural network runs over Tensorflow [129], an end-to-end open-source platform for machine learning. For hyper-parameter setting,

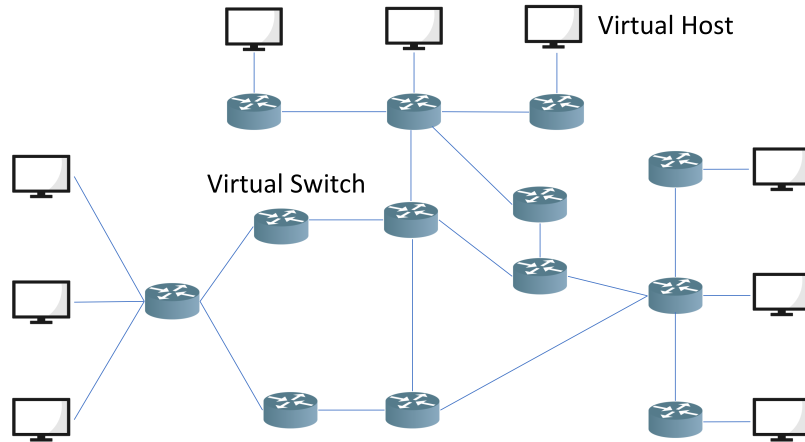


Fig. 4.3 A sample of emulated network topology used during the experiment campaign. The design reflects the desire of allowing a variety of test conditions and a multitude of available paths.

we set the discount factor $\lambda = 0.99$ and the learning rate $\alpha = 0.9$. The batch size is 256 to enable a high learning rate. If not otherwise specified, cost saving, application performance, and fairness are equally important, *i.e.*, $\mu = \lambda = \omega = 0.3$. We set the default interval for collecting statistics $P = 0.5$ s, and the action decision interval to $I = 1$ s. Nonetheless, in what follows, we also evaluate the consequence of alternative intervals over the system performance.

Network use case. The network topology considered throughout experiments should offer sufficient path redundancy, enough hosts to deploy the desired traffic patterns, and be reasonably suitable for an edge network. Taking all these considerations in mind, we converged on a network use case shown in Fig. 4.3. This topology is the default environment utilized, but also other settings with different link densities are considered. OVS switches are utilized for the measurement of the throughput per each ongoing flow, in turn, mapped to a single QoE value, following considerations of Section 4.6.2.

Benchmark algorithms. To validate our solution we compare against three state-of-the-art solutions adequately adapted to our use case: an ML classifier-based method to perform auto-scaling, [89], SRSA [91], and ElasticTree [88]. In [89], an ML-based method converts the auto-scaling decision to an ML classification problem, so that it can learn from the insights and temporal patterns hidden inside measured data from the network. As shown in their study, Random Forest (RF) is the algorithm

performing better; thus, we apply this method, and, for simplicity, we refer to this solution as Auto-RF. Conversely, SRSA is a reinforcement learning approach to auto-scaling VMs in a telco cloud platform [91]. Even though our solution shares the RL formalization, substantial differences involve the model, the objective (and reward), the use case, actions associated, and a single agent vs. multi-agent version. Finally, ElasticTree attempts to solve a power optimization problem and compares multiple solutions strategies. A greedy bin-packing heuristic has been advocated as an adequate solution, and for this reason, we employ the version of ElasticTree using such a heuristic. However, it is relevant to note that our difference with the state-of-the art not only resides in a multi-agents configuration, but also in an optimized algorithm which can handle more complexity for more conscious decisions, as seen in Section 4.7.

4.7 Performance Evaluation

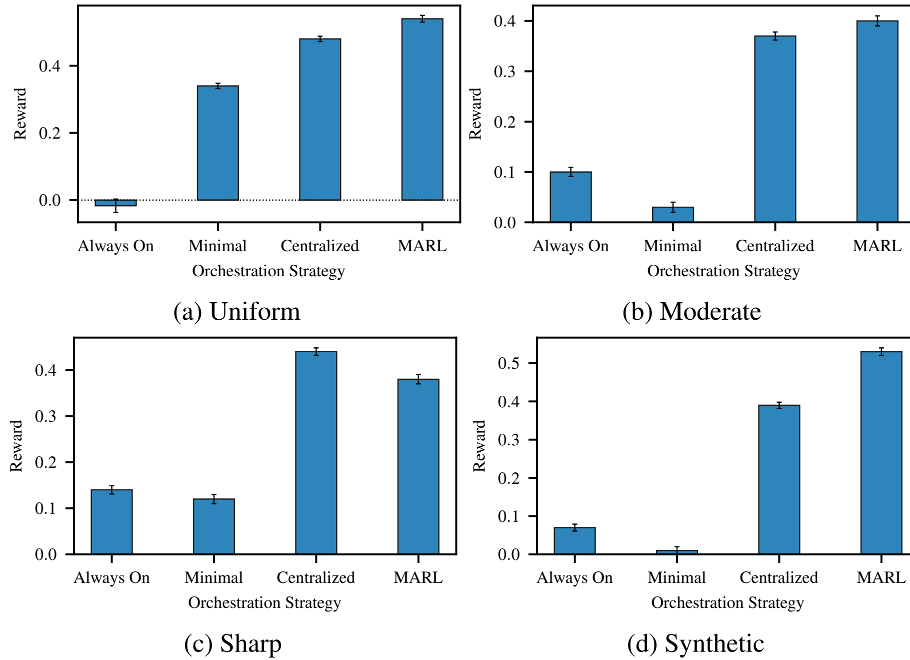
In the following, we compare the goodness of our data-based approach with respect to model-based solutions, usually solved via heuristics, and other data-based techniques. After a brief explanation of the considered metrics, we measure the impact of a MARL approach over network performance. Then, we extensively compare Mystique versus related solutions in several network conditions. Lastly, we also run sensitivity experiments by varying some algorithm parameters.

4.7.1 Evaluation Metrics

In this section, we make use of metrics and quantity defined in Section 4.4 and 4.5, such as the QoE fairness and reward function.

Besides them, one of the primary metrics we inspect is the percentage of power savings, computed as:

$$\begin{aligned}\%\text{ power savings} &= 100 - \%\text{ original power} = \\ &100 - \frac{\text{power consumed by solution} \times 100}{\text{power consumed in original network}}, \end{aligned} \tag{4.12}$$



(a) Uniform (b) Moderate

(c) Sharp (d) Synthetic

Fig. 4.4 Reward function for different traffic patterns: (a) uniform, (b) moderate increase, (c) sharp increase, (d) synthetic generation. Four available strategies are compared, highlighting differences between centralized versus decentralized (MARL) model.

which gives an accurate idea of the overall power saved by turning off switches and links. The original power, indeed, represents the consumption for the *always-on* baseline.

Clearly, the savings depend heavily on the network utilization, u, defined as the average of the load on the link, $L_{u,v}$, over all links weighted by their capacity. In practice, this is the sum of the link flows over the entire network divided by the sum of link capacities, formally:

$$u = \frac{\sum_{(u,v)\in E} \sum_i^k f_i(u,v)}{\sum_{(u,v)\in E} c(u,v)}. \tag{4.13}$$

4.7.2 Centralized versus Multi-Agent Reinforcement Learning

Firstly, we compare the performance of our distributed solution based on multiple agents with a centralized version build upon the same model (Section 4.4). For the sake of completeness, to understand the advantages of self-learning capabilities, we

also compare them against *always-on* and *minimal orchestration* baseline algorithms. In the former setting, all the switches and links are maintained during the experiment. This policy ensures the best applicative performance but high energy consumption. On the other hand, in a minimal orchestration, no redundancy is exploited, and only the minimal subset to let the network works is kept. This configuration leads to a minimum in management cost, but a degradation in the performance.

Figure 4.4 compares the reward function defined in (4.11), for the four algorithms: always-on and minimal orchestration as baseline algorithms, and the centralized and decentralized version of the RL process, i.e., MARL. In the centralized, a single controller handles the switches of the network, while for the decentralized setting, three controllers are in charge of managing the network and instruct the switches. Evaluating their strengths and weakness implies appraising the behavior for different traffic demands. For this reason, we analyze the reward function across four traffic patterns: uniform, moderate increase, a sharp increase, and synthetic generation for a more realistic use case. We can observe how the distribution of the concerns across multiple agents, as in the decentralized version, produces higher rewards in almost every context. The only exception resides in the sharp increase of the traffic demand, given the limited knowledge of the network status.

4.7.3 State-of-the-art Comparison

After a first assessment of Mystique performance, we evaluate our solution against the benchmark algorithms stated above. In a similar way to the previous evaluation, we report in Fig. 4.5 the (a) energy efficiency, (b) application throughput, (c) QoE fairness, and (d) reward function, for the considered methods. By considering the plots, we can notice how our system outperforms the related algorithms in all the examined metrics. In particular, fairness is distinctly one of the most improved quantities in Mystique, as one of the desiderata. Furthermore, none of the other algorithms can efficiently optimize more metrics simultaneously, but they can successfully improve only some of them. Conversely, Mystique stably outperforms other solutions, demonstrating its ability to optimize management costs and QoS parameters altogether in multiple network scenarios.

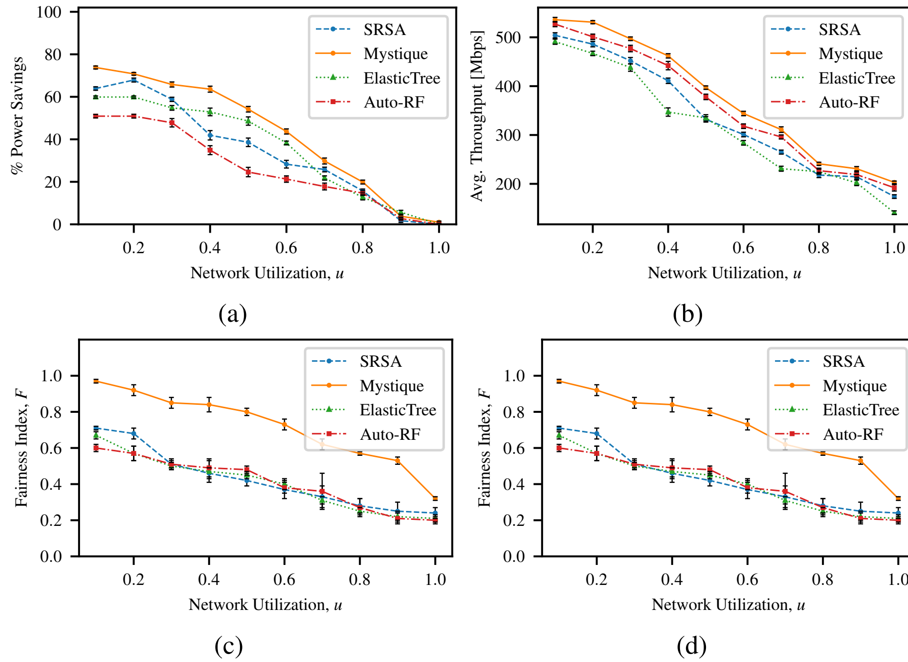


Fig. 4.5 Performance comparison with other benchmark algorithms in terms of (a) energy saved with respect to original setting, (b) mean application throughput achieved, (c) fairness index for the flows, (d) system reward.

4.8 Conclusion

This chapter presented Mystique, a system that allows scaling network resources up and down to track network utilization. The network controller can dynamically activate or deactivate links and nodes in an "as needed" fashion with the aim of minimizing the energy consumption and improving QoE and fairness among users. Furthermore, the system can promptly react to network failures as these happen. In this context, the chapter highlights the benefits of splitting the decision logic across multiple controllers, for a distributed and fault tolerant architecture. We show how this approach can improve the management when the quantity of information needed for the model becomes large and can lead to more accurate actions.

Chapter 5

High-Performance Transport Protocol

A performing congestion control protocol is fundamental for proper network operation as it ensures telecommunication stability, fairness in computer network resource utilization, high throughput, and a low switch queuing delay. Although many solutions have been proposed in the last decade, Transport Control Protocol (TCP) still constitutes the overwhelming majority of current Internet and Long Term Evolution (LTE) communications, and the vast majority of congestion control mechanisms are implemented on TCP [130].

Despite the wide deployment of TCP, various studies have shown how it performs poorly in scenarios that require adaptability or that departs from the original network conditions on which it was designed in the '70s [131–135]. In particular, problems may occur in cellular and wireless networks, where TCP misinterprets the stochastic packet losses as congestion, hence leading to performance degradation [135]. This issue has motivated many authors to propose innovative congestion control approaches that follow a domain-specific design philosophy, in which the design is limited to a specific network scenario and it leverages its specific characteristics to boost the performance. Examples are in data centers [136, 137] and edge networks [134, 135].

The challenge of adequately updating the *congestion window (cwnd)* in resource-constrained networks, such as wireless networks and IoT, is exacerbated by inherent problems arising from their limited bandwidth, processing, and battery power, as well as from their dynamic conditions [138, 20]. The deterministic nature of TCP is indeed more prone to cause *cwnd* synchronization problems and higher contention losses,

The work presented in this chapter has been partially published in [22].

due to node mobility that continuously modifies wireless multi-hop paths [139, 140]. Several TCP variations (e.g., PCC [141] and Copa [142], to mention a few) have been recently proposed to overcome these shortcomings. Nevertheless, the fixed rule strategies used by these solutions are often inadequate to adapt to the rapidly changing environment.

To solve the problem of an adequate congestion window update strategy, we have developed *Owl*, a novel transport protocol based on *reinforcement learning (RL)*. Although many transport protocols have been designed, with reinforcement learning [143, 144] or without for a network-aware solution [145, 136, 146], the most recent solutions using RL do not exploit network intelligence fully.

Our transport protocol *Owl* is able to increase the throughput and fairness while reducing the number of packets lost and delay by learning from several end-to-end and in-network metrics. In particular, our contributions are summarized as follows. We designed and implemented as a kernel module *Owl*, a new congestion control protocol that leverages partial network knowledge to train a reinforcement learning model based on Deep Q-Learning [147], improving the network performance with respect to recent work [148]. The outcome of Owl model is the next congestion window value, a crucial and volatile parameter for any reliable telecommunication. We then evaluate our solution extensively: first, we compare Owl with other sixteen transport implementations. Some of these solutions were designed for wireless networks, such as Sprout [134] or the more recent ABC [145], while others [149–152] were chosen since they are widely deployed in several Linux distributions.

Our performance results (obtained using emulations with real available traces from Verizon and T-Mobile and a deployment over the GENI testbed [86]) show that Owl has consistently bandwidth and delay improvements across several scenarios. We also evaluate the parameters of our deep neural network used in our reinforcement learning and tested Owl's fairness performance, finding that our transport protocol behaves less aggressively than others.

As owls that (are wise and) can see with poor light conditions, our protocol operates with partially visible networks.

5.1 Related Work

Congestion control and avoidance problems have been widely discussed in the literature due to the great importance in reliable data transmissions. To solve the optimal congestion window inference problem, recent machine learning-based algorithms have been proposed with promising results in different network scenarios. In this section, we focus on highlighting how these solutions differ from our protocol.

Congestion Control is a fundamental service offered by TCP, so much so that significant improvements and variations have been proposed over the years. A few examples are TCP Vegas [151], Compound [153], Fast [154], BBR [152], and Data Center TCP (DCTCP) [136]. Rather than relying on indications of lost packets to adjust the *cwnd* as traditionally happens, BBR considers RTT and average delivery rate measurements to decide how fast to send data over the network. This enables BBR to be resilient to the bufferbloat problem, but it frequently exceeds the link capacity, causing excessive queuing delays [145]. Other protocols, e.g., Compound [153] and Fast [154], instead attempt to optimize losses, but they rely on some predefined functions or rules to handle network conditions.

In summary, all these solutions share the limitation of fixed-rule strategies, that is, their performance is challenged in networks that require rapid adaptations. Our solution, instead, uses a (reinforcement) learning approach to overcome this limitation and predicts the best *cwnd* update at each transmission event.

Recent end-to-end congestion control solutions, such as Remy [133], PCC [141], PCC-Vivace [155], define an objective function to optimize the process of online actions definition, e.g., on every ACK or periodically. Remy [133], for example, offline trains every possible network condition to find the optimal mapping with the sender's behavior. These mappings are stored a-priori in a lookup table, and rely on what has been seen and hence can accommodate new network conditions only by recomputing the lookup table. On the other hand, PCC [141] and PCC-Vivace [155] perform online optimizations. For instance, PCC can rapidly adapt to the varying conditions in the network by aggressively searching for more accurate actions to change the sending rate. However, these online rules are often complex and require considerable lags in estimating all the parameters to be accurate.

Our protocol also uses a utility-bases approach, but exploiting a deep neural network to better adapt to a specific network, leaving the utility customization as a policy that can be tailored to more specific requirements.

Learning for Congestion Control. As a recent trend, Machine Learning (ML) has been widely applied to various problems arising in network operation and management [14]. We use ML to adapt the *cwnd* estimation. The majority of alternative approaches are specifically designed to cope with a resource-constrained network, including IoT [138] and WANETs [140, 151, 139]; others instead address a wider range of network architectures [156, 141, 133]. Recently, RL has permeated many congestion control mechanisms, such as Orca [157] and Aurora [148], where in Aurora, the previous Performance-oriented Congestion Control (PCC) protocol was extended with a Deep-RL approach. Our RL approach differs from others for the agent state: we effectively combine features from both the transport and the network layers, without significant burdens to the Linux kernel module.

In-Network versus End-to-End Congestion Control. Several protocols leverage the Explicit Congestion Notification (ECN) to provide network-level feedback to end hosts. For example, DCTCP [136] modifies the Red Early Drop thresholds of ECN to achieve high throughput, high burst tolerance, while keeping queues empty hence experiencing low latency. ABC [145] instead improves on ECN by sending accelerate and brake signals instead of merely random early drop signals, and hence more accurately adjusts the source sending rate. As ABC, Owl also uses network-level information as well (when available), however, our feedback comes from a network controller, e.g., a measurement agent or an SDN controller, that computes statistics about device utilization. Also, Owl does not need any modifications to packets headers or custom routing devices logic, which leads to challenging deployments. In fact, Owl only relies upon client-side changes and a network statistics collector, a standard operation across multiple network scenarios. On the one hand, our network-level feedback carries more information than a simple bit in the TCP header. On the other hand, Owl functions properly also without network knowledge, while ABC and other ECN-based approaches require network knowledge to work.

5.2 Congestion Control via Reinforcement Learning

We now overview our primary components in the RL method, starting with our considered state set, then with the set of actions on the congestion windows, and finally, with the utility that drives the choice of the next protocol action.

Table 5.1 The network statistics gathered for estimating the upcoming performance.

	Features of the Owl congestion window predictor
1	Time-stamp [jiffies]
2	Congestion Window Size (*cwnd*) [packets]
3	Round Trip Time (RTT) [ms]
4	RTT variation between two consecutive samples [ms]
5	Maximum Segment Size (MSS) [bytes]
6	Number of delivered packets
7	Packets lost during a transport session
8	Current packets in-flight
9	Number of retransmissions [packets]
10	Partial Network Knowledge (*PNK*) [packets]
11	Percentage of known network [%]

State Space. Table 5.1 summarizes the features that we selected to build our model state space. We consider both end-to-end statistics (features 1 to 8) and network-level statistics (features 10 and 11). Thus, the former set of features is collected at the sender side at each time interval, any *jiffy*, where jiffy is the finest time granularity on Linux systems. Instead, the last two features represent the partial information coming from the network (feature 10), and a parameter stating the quantity of knowledge, as a percentage of the whole network (feature 11), respectively. For each switch under control, let P_{in} be the total number of packets received in a given time interval (one second in our implementation), and P_{out} the total number of outgoing packets. We then define `diff` as $|P_{in} - P_{out}|$. Our *Partial Network Knowledge* (*PNK*) represents an indicator of the known level of congestion within the network. In particular, given a source receiving statistics or updates from z switches on the path between a source and a destination, *PNK* is computed using the following equation:

$$PNK = \max(diff_1, diff_2, ..., diff_z). \tag{5.1}$$

PNK informs about the current congestion level, and consequently, the loss rate occurring in the network. We choose *PNK* as it is easy to compute and accessible by a vast number of protocols and network measurement applications, such as OpenFlow or NetFlow. Nonetheless, we remark that Owl can also be configured as an end-to-end protocol, in case the network knowledge is hidden or impractical to obtain.

In defining our states, we also consider a history window of k values for each chosen feature as our state. This approach helps our algorithm to predict the network conditions adequately and to adjust the congestion window accordingly. The neural network of our deep reinforcement learning algorithm receives a matrix N by k, where k are the historical values for each of the N features. In our experiments, k has been set to 5. We augment our state space with a history of generic length k to help the agent's learning. However, we do not set this hyperparameter to a large value since that prevents the state from growing unreasonably, and because forgetting history faster is beneficial.

Actions. The congestion window (*cwnd*) is one of the per-connection state variables that is used by TCP to limit the amount of data a sender can transmit before receiving an ACK. Since TCP was designed based on specific network conditions and handles all packet losses as network congestion, in wireless lossy links it unnecessarily lowers its rate by reducing the *cwnd* at each packet loss, negatively affecting the end-to-end performance. Hence, we exploit an online training algorithm based on RL to update the *cwnd* properly.

The selection of actions is the key to the proposed algorithm's effectiveness. The list of actions specifies how Owl should change the *cwnd* in response to every packet acknowledge. The set of acceptable congestion window values is large and tied to the reward of the RL system. Hence, there is no unique solution across every network condition. After an empirical evaluation, we converged on the set that has given us the highest utility, that is:

$$A = \{-10, -3, -1, +0, +1, +3, +10\}. \tag{5.2}$$

We allow the agent to change the *cwnd* in any direction with different intensities. The first three options reduce the size of the congestion window with a distinct extent, whereas the last three increase it by three different values. The last action does nothing to the size of the *cwnd*, letting it remains the same as before. We want to

encourage the agent to explore diverse ways to influence the connection by assigning different magnitudes to the performed change. Indeed, not only the learning agent should predict when increasing or decreasing the *cwnd*, but also to what extent. For example, our algorithm must learn when the network state suggests that a large part of the bandwidth is unused to aggressively increment the window size, while it must only slightly increase it when the network approaches any congestion. Our network module starts with an initial *cwnd* of 10.

Due to the opted approach, the protocol learns how to make control decisions from experience and, thus, eliminates the need for necessary pre-coded rules to adapt to the variety of network environments. Finally, we converged to the action set in Eq. 5.2 after having performed a substantial number of empirical trials. Nevertheless, the action set A is a policy that can be tailored to specific use cases, by either modifying values of the congestion window size (as we did) or acting upon other TCP parameters, e.g., timeout estimation or slow-start threshold.

Utility function (RL reward). The selection of the congestion control schema relies on a utility function that models the application-level goal of "high throughput and few losses". In particular, the utility U of sender i is a function of throughput λ and packet loss rate p, as follows:

$$U_i = \lambda_i - \delta_i \lambda_i \left(\frac{1}{1 - p_i} \right), \tag{5.3}$$

where $p \in [0, 1)$ and δ is an adjustable coefficient determining the importance of the components. For example, a larger δ implies that lower packet delays are preferable. The goal of each sender i is to maximize its utility function U_i.

In the following we focus on the utility's motivation. In particular, we show that processes running our protocol converge to a stable rate assignment.

5.2.1 Stability Analysis

We will demonstrate how no sender has the incentive to deviate its sending rate from the strategy defined by our Owl protocol objective function, hence reaching a Nash equilibrium. At the equilibrium condition, we have the n-tuple of sending rates

defined as $(\lambda_1, ..., \lambda_n)$. Formally we have that:

$$U_i(\lambda_1, ..., \lambda_i, ..., \lambda_n) > U_i(\lambda_1, ..., x, ..., \lambda_n), \quad (5.4)$$

for any sender i and any non-negative sending rate x, and so the following theorem holds.

Theorem 5.2.1. *(Stability). Consider n senders sharing a bottleneck link, and λ_i to be the rate of sender i; if for every sender i the objective function is defined by Equation 5.3, the sending rates converge to a stable equilibrium. Moreover for every sender i, we have:*

$$\lambda_i = \frac{C\left(\frac{n}{\delta_i} - \hat{z}\right)}{n+1}, \quad (5.5)$$

where $\hat{z} = \sum_{j \neq i} \frac{1}{\delta_j}$.

Proof. We need to show the existence of a Nash equilibrium, *i.e.*, no sender can increase its objective function value by unilaterally changing its rate. We consider a network model with n competing senders sharing a bottleneck link of capacity C and a FIFO-queue. Assuming a tail drop queue eviction policy, the loss rate function can be described as:

$$p_i = \begin{cases} 1 - \frac{C}{\sum_i \lambda_i} & \text{if } \sum_i \lambda_i > C \\ 0 & \text{otherwise} \end{cases} \quad (5.6)$$

Let us denote the arrival rate in the queue by $S = \sum_i \lambda_i$. Since the term $1 - \frac{C}{S} = \frac{S-C}{S}$ is independent of i and it is equal for all senders, all senders should experience the same loss rate, we denote p_i simply by p. By substituting these new terms into Equation 5.3, we obtain:

$$U_i = \lambda_i - \delta_i \lambda_i \frac{S}{C}.$$

First we compute the partial derivative, $\frac{\partial U_i}{\partial \lambda_i}$, and we split S into the two addends $S = \lambda_i + \sum_{j \neq i} \lambda_j$. Thus, for each i yields:

$$\frac{\partial U_i}{\partial \lambda_i} = 1 - 2\frac{\delta_i}{C}\lambda_i - \frac{\delta_i}{C}\sum_{j \neq i} \lambda_j.$$

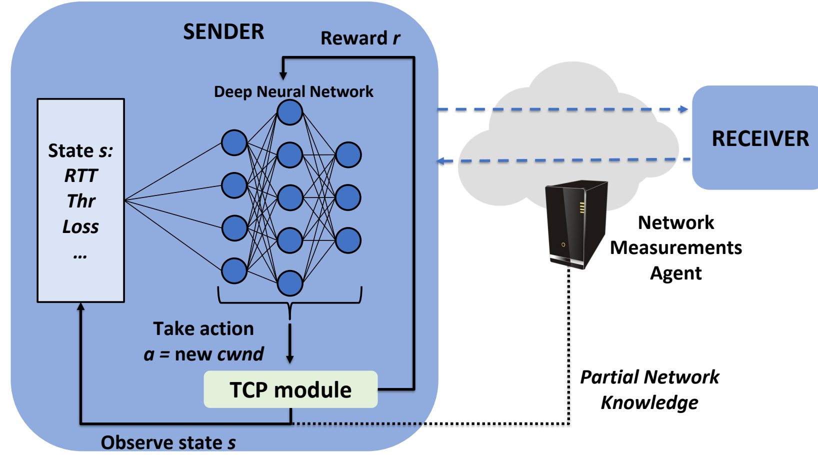


Fig. 5.1 Owl Overview: reinforcement learning sender's agent interaction with the network.

We then compute the second derivative of U_i, with respect to the rate, and we obtain the negative quantity $-\frac{2\delta_i}{C}$. Hence, the utility is concave and the Nash equilibrium is achieved if, and only if, $\frac{\partial U_i}{\partial \lambda_i} = 0$. Next, to find the rate at which the equilibrium condition is achieved, we introduce $\hat{z}$ defined as $\hat{z} = \sum_{j \neq i} \frac{1}{\delta_j}$. Hence we have:

$$1 - 2\frac{\delta_i}{C}\lambda_i - \frac{\delta_i}{C}\sum_{j \neq i}\lambda_j = 0$$

$$2\lambda_i + \sum_{j \neq i}\lambda_j = \frac{C}{\delta_i}$$

The solution to the stated system of linear equations is:

$$\lambda_i = \frac{C\left(\frac{n}{\delta_i} - \hat{z}\right)}{n+1},$$

which is the desired sending rate of sender i.

□

5.2.2 Owl Protocol Design

In Figure 5.1, we detail the main actions performed by the sender. The collected metrics are fed to the Neural Network, and the protocol starts (Algorithm 3).

Algorithm 3 `Owl cwnd update`

1: Let S and D be the target source and destination
2: $F \leftarrow$ flow connecting S and D
3: Collect state vector `s` at time `t` for flow F
4: `cwnd`*`(t)` $\leftarrow$ `cwnd_Prediction(s, t)`
5: Set `cwnd` to `cwnd`*`(t)`

Specifically, we collect the state of the end-to-end communication, e.g., RTT and throughput, exploiting the TCP Linux API. Concerning the network feedback, the network measurement agent computes *PNK* by controlling the underneath topology, and notifies it to the sender. We argue it is not always possible to obtain the entire path between the source and the destination. However, even when the network feedback is incomplete or unavailable (the neural network does not use the in-network features), our protocol still provides valuable results.

Once Owl has collected such values, it selects the next *cwnd* by choosing the "action" according to the Q-table. The algorithm to predict the next *cwnd* value is detailed in Algorithm 4. In particular, the algorithm avails the states, actions, and

Algorithm 4 `cwnd_Prediction(s: state, t: time)`

1: At time $t = 0$ initialize $Q(s,a) = 0$ and set reward r as in Eq. (5.3)
2: At time t :
3: Observe r as a consequence of the last action
4: Update $Q(s,a)$ function according to $Q(s,a) = Q(s,a) + \alpha\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
5: `cwnd`*`(t)` $\leftarrow$ `softmax(a,s,t)`
6: **return** `cwnd`*`(t)`

reward to select the best value and update the Q-table. It selects the next *cwnd* using the softmax criteria, which allows to rank and weight the actions according to the estimated utility. Such a decision occurs every time a packet is acknowledged to guarantee an adequate refresh of the *cwnd* used in the congestion avoidance phase. The state set is then updated to assure k historical values for each metric at any interval.

5.3 Owl Prototype Implementation

Network Scenario. In designing our protocol, we considered practical scenarios in which networks are *partially unknown*. Wide-area networks may require (undesirable) cooperation and coordination of multiple (federated) gateways, and unstable network conditions may hide information. Part of our evaluation in Section 5.4 focuses on the performance analysis of our protocol with such partial network knowledge, showing that the in-network information may add value if available, but it is not required as in other in-network congestion control mechanisms.

To analyze and respect this partial unavailability constraint, we designed and implemented a system in which a software-defined network (SDN) controller acts as a measurement collector and manages only some of the deployed (virtual) switches. While we use an SDN controller in our implementation, our approach is not limited to this specific technology. The controller interacts periodically with the switches to collect statistics about the number of packets transmitted and received. Such statistics are then used by our implementation to learn and predict the end-to-end action to take given the level of congestion. In our implementation, the controller receives packets' statistics from all switches with a (re-configurable) sampling rate of one-second, a good trade-off between overload and freshness of information. The controller also runs a simplistic web server and exposes REST API to obtain these values, which are part of the input of our RL algorithm.

Kernel Module. The Owl module is responsible for setting the optimal congestion window. To operate, it obtains network states by communicating with a measurement agent, for example, an SDN controller. Our prototype is composed of two main processes: one running in the kernel and one in user-space. The kernel module exploits functions included in the classical *tcp_cong.c* to have access to the underlying congestion control functionalities of TCP. Like any other module, our kernel implementation can be mounted as a pluggable congestion control algorithm. It can set and get end-to-end transport states such as Sequence Number, ACKed Packets, RTT, and efficiently compute the throughput.

The application process running in user-space collects information about the current TCP socket and uses them to build the input matrix of a Deep Neural Network running the reinforcement learning algorithm. The module takes actions in line with the RL feedback and modifies the *cwnd* as a reaction to events (Section 5.2).

Storing the required states to run a reinforcement learning algorithm and to keep communications with the network controller can be costly at the kernel level. On the other hand, a user-space application can leverage a more extensive set of libraries to fit the learning algorithm's needs. Besides, the transmission of packets to/from the network controller could arise issues and requires proper management without having to switch from user-space to kernel-space. For this reason, we implemented the network management components of our congestion control algorithm at the user-space and marshall current TCP socket states between user-space and kernel via the *Netlink* service, commonly used for this purpose.

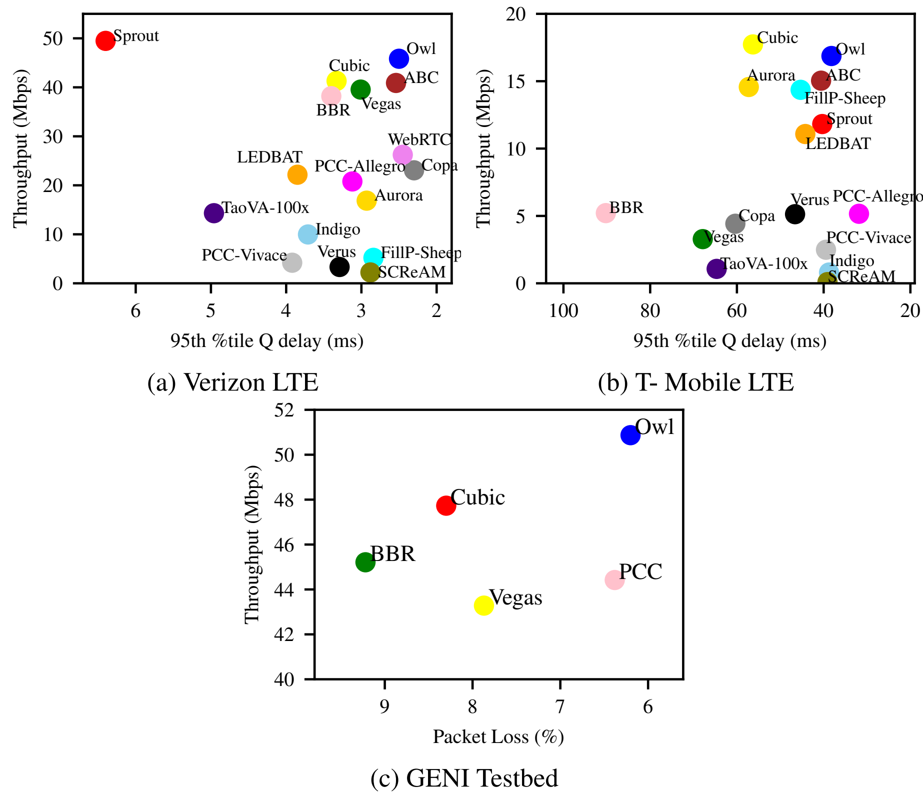
The reinforcement learning-based congestion controller accumulates network statistics from ACKs over a fixed period and sends the action asynchronously in a separate thread. The speed in retrieving data from the kernel is indeed higher than the rapidity of the reinforcement learning processing.

5.4 Protocol Evaluation

To evaluate our proposal, we tested Owl against sixteen other transport protocols. We built the services using a virtual network testbed and the Mahimahi emulator [158], a recent cellular link emulator that allows testing with real cellular traces from two of the largest US telecommunication providers, Verizon and T-Mobile. The network is emulated through namespaces, via Mininet [127]. The transmission goes through a Software-Defined Network (SDN), where switches interact with a centralized controller (in our implementation, we used Ryu). We also evaluate the performance over real hosts, and we deployed Owl over the GENI testbed [86]. Throughout our experimental campaign, we use the utility function described in Eq. 5.3, where δ has a value of 0.7. If not otherwise specified, we set a default percentage of known paths to be 80%. To evaluate each protocol, we average 35 experiments in which each sender-receiver pair runs *TCP iperf3* for 100 seconds.

5.4.1 Trace-Driven Emulation Results

To understand how Owl performs compared to other solutions, we deployed our protocol over an emulated network created with Pantheon [159], a well-known fairly recent testbed developed to evaluate congestion control schemes. In particular, we



(a) Verizon LTE (b) T- Mobile LTE

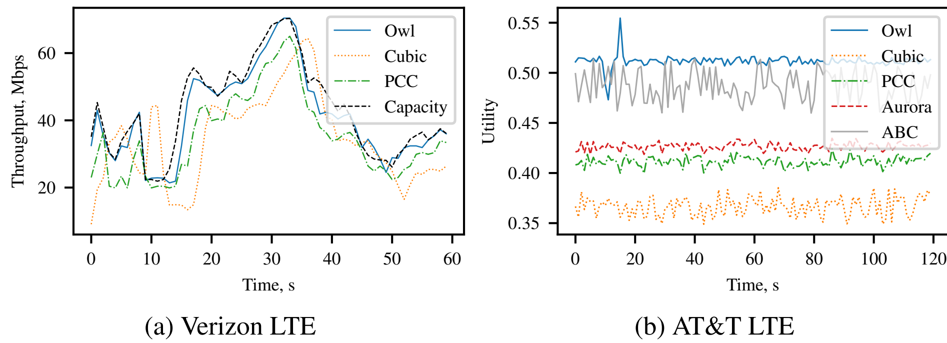
(c) GENI Testbed

Fig. 5.2 **(a)-(b) LTE Trace-driven emulation.** Owl vs. previous schemes (using RL or not) tested over two cellular network traces (top-right are better). In both cases, Owl outperforms our benchmark, and has the highest fairness, on average, in both our tested use cases (Section 5.4.1). **(c) GENI testbed evaluation.** Throughput-loss rate trade-off for kernel-level solutions over real networks. Owl optimizes the two quantities simultaneously (Section 5.4.2).

compared Owl against sixteen other protocols, divided into five categories: (*i*) end-to-end TCP designs: Cubic [149], Vegas [151], BBR [152], Copa [142], PCC [141] and its variants; (*ii*) end-to-end cellular, *i.e.*, LTE protocols: Verus [135], Sprout [134]; (*iii*) Machine Learning-based transport protocols: Indigo [159] and Aurora [148]; (*iv*) explicit congestion control: ABC [145] and (*v*) mixed schemes: LEDBAT [160], SCReAM [161], WebRTC [162], Tao-VA [163]. For our LTE evaluation settings, we use the publicly available [158] Verizon and T-Mobile traces, with separate packet delivery for uplink and downlink. The traces were captured directly on those cellular networks and reports the available bandwidth over time. These traces are also loaded on our local SDN-based virtual network testbed. Our OpenFlow controller is only aware of the virtual switches (instances of Open Virtual Switch (OVS) [42]) that are connected to the SDN controller. For in-network algorithms, such as ABC, we

emulate compliant routers as Mininet hosts that marks the packets according to the algorithm's logic.

Figures 5.2a-b shows that Owl performs efficiently in all tested scenarios. In the case of Verizon LTE traces (Figure 5.2a) and a single sender, Owl achieves both good throughput and 95th percentile per-packet-delay, and no other solution has shown a better combined throughput-delay performance. Even though the RL reward was designed to achieve high throughput and low loss rate, we can observe that our mechanism can simultaneously obtain a low RTT, as a consequence of the imposed utility. Similar conclusions hold even for T-Mobile traffic (Figure 5.2b), where Owl provides a desirable trade-off between throughput and delay. It is worth noticing that none of the other algorithms outperform Owl in both tested environments: our solution appears to be more stable across traces.



(a) Verizon LTE (b) AT&T LTE

Fig. 5.3 **Our protocol best follows the available bandwidth.** (a) A 60-seconds throughput's evolution compared to the actual link capacity. Owl fits best the Verizon LTE trace; while, especially for Cubic, overshoots in throughput lead to large standing queues. The curves shown have been selected for visual clarity. (b) A 120-seconds utility's evolution. Owl guarantees an adaptive response to the network dynamic changes.

Figure 5.3 shows the shortcomings of transport protocols in use and the lack of adaptation required for a good transport protocol. The Figure 5.3a represents a sample of the throughput evolution over the Verizon LTE downlink traces for 60 seconds. For the sake of clarity, we report only our comparison to Cubic, as it is the default in many Linux implementations, and PCC, as it one of the best performing within utility-based approaches. Owl adapts its sending rate so as to closely match the bottleneck link's available bandwidth (dashed black line in the figure). In contrast, Cubic slowly reacts to changes in the network, and PCC partially approximates the link capacity. *Our protocol can cope with rate variations in a reactive manner and closely approximates the desired behavior by learning the optimal action.*

This result is also confirmed in Figure 5.3b where we plot the utility (Eq. 5.3) obtained with different algorithms over AT&T LTE downlink. This time, we compare against ABC [145] as it is the most representative of explicit congestion control and Aurora as a novel RL-based congestion control algorithm. Likewise, we can observe how Owl regularly provides a higher utility than the benchmarks over time. This is due to the ability of the framework to learn the optimal behavior during training and then react efficiently during network dynamics. We can also observe how Aurora and Cubic fail to promptly react to the events.

5.4.2 Evaluation over the GENI Testbed

To establish the practicality of our approach and understand how Owl performs over wide-area Internet paths with real cross-traffic and real packet schedulers, we deploy our solution on the GENI testbed. In these experiments, we evaluate how congestion control schemes behave across two federated GENI aggregates and three senders transmit at the same time. We measure the performance of each schema when competing with another flow to accentuate the possible congestion occurrences. To evaluate our protocol in these realistic settings, we average the throughput and delay over 60-second flows, while the senders share a bottleneck link with 3ms RTT and a bandwidth of 100 Mbps.

We summarize in Figure 5.2c the performance of our protocol when compared to other protocols available on Linux. Our prototype evaluation deployed in real settings match our emulation results: our implementation can jointly achieve high throughput and a low loss rate when compared to other solutions, balancing the two components effectively.

5.4.3 Partial Network Knowledge Impact

Next, we discuss our experiments regarding the impact of the required network state knowledge that Owl needs to train the RL system effectively. Figures 5.4 display the (a) throughput and the (b) RTT, when different transport protocols run over a network composed of 20 nodes emulated on our local Mininet virtual network testbed. Specifically, we compare against Cubic as a reference end-to-end congestion control, Aurora, as a reference RL-based congestion control, and ABC, as a reference

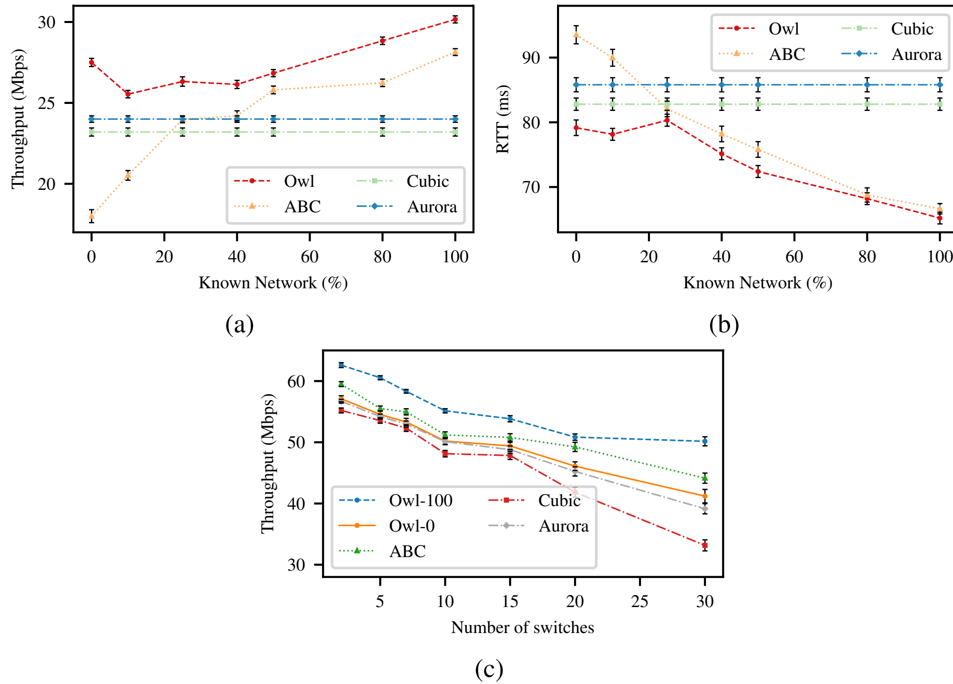


Fig. 5.4 **Network Knowledge Impact on Performance.** (a) Throughput and (b) RTT of Owl protocol for increasing percentage of known network. Somehow surprisingly, the highest performance gaps with respect to other algorithms are obtained when the percentage of network knowledge is either low or very high. (c) Throughput performance with or without network knowledge averaged over different network topologies and increasing number of informing switches.

in-network control. The performance of Cubic and Aurora are not affected by the lack of in-network knowledge since they are both end-to-end congestion control algorithms. On the other hand, ABC performs worse than Owl when the number of ABC-compliant routers is relatively low. Our results validate that the value of *PNK* is beneficial to the algorithm, but our protocol works even as a pure end-to-end strategy. Our measurements reveal that even when less than 50% of the switches are utilized to collect statistics, our solution outperforms both end-to-end approaches (like Cubic) and novel in-network protocols (like ABC). On the other hand, if a partial network knowledge (more than 50%) is available, Owl drastically speeds up the transmission in terms of throughput and reduces latency. The worst result occurs approximately when half of the devices are controlled, as the agent cannot assign the proper importance to the coming information, resulting in occasionally misleading values. Nonetheless, even though in this scenario the information does

not help improve the overall performance, Owl has results that are comparable to other protocols.

Lastly, we compare Owl against a few representative protocols as we increase the number of informing switches over randomly generated topologies, *i.e.*, links are randomly generated while we fix the network size. The link capacity is also uniformly distributed at random between 50 and 100 Mbps. We are interested in assessing the impact of the network size on our congestion control algorithm. To this aim, we compare the perceived throughput when our solution has no in-network congestion feedback, and when the network is as informative as it can be, *i.e.*, the in-network feedback arrives from 100% of the switches. In Figure 5.4c, these two Owl policies are denoted with Owl-0, namely, zero-percent of total switches are communicating with the source, and Owl-100, respectively. It is notable how a full network awareness is beneficial and allows a less prominent (and inevitable) performance degradation when an increasing number of switches compose an end-to-end path. However, we note how even Owl-0 provides better results than recent end-to-end congestion control solutions based on RL [148].

5.5 Conclusion

In this chapter, we presented Owl, a reinforced learning-based transport protocol designed to learn from end-to-end and in-network signals. Our evaluation, with a kernel implementation and real traces, confirms that Owl is effective under various network conditions, and it can speed up transmissions and reduce delays and loss rate better than most existing protocols in the vast majority of the tested scenarios.

We also analyzed the stability condition of Owl and evaluated its fairness demonstrating that it is less aggressive than other performant solutions when it competes with other protocols and when it competes with itself across other sources. Finally, we showed how taking into account information involving the network layer leads to increasingly better results, especially when at least 50% of the network congestion state is available at the source.

Chapter 6

Task Offloading in UAV Networks via Multi-Agent Reinforcement Learning

The past decade has witnessed an explosive growth in mobile internet applications consuming a significant amount of computational resources, e.g., face recognition, virtual/augmented reality, realtime media streaming, mainly favored by the development of the Internet of Things (IoT). A specific area of interest entails vehicles and, in particular, Unmanned Aerial Vehicle (UAV) systems, that have experienced a constantly increasing popularity in the last years, mainly thanks to their maneuverability, flexibility, and limited deployment costs. UAVs have been primarily used for military applications, but they are now expanding into business, science, agriculture, and civilian fields, where successful examples include supports of first responders, surveillance, aerial photography to cite a few [164] (see Section 2.2). Their constrained resources, however, open the problem of offloading part of their tasks to the close multi-access edge computing (MEC) in order to speed up the computation. In such a scenario, the IoT device can offload computationally intensive tasks to nearby edge cloud to reduce computation latency and energy consumption [165–168].

The problem of task offloading has been extensively studied in the literature [169, 35, 165, 170, 171], where recent solutions attempt to significantly reduce the processing time of mobile vehicle applications while greatly reducing data processing delays and energy consumption. With the advent of machine learning (ML) and, specifically, reinforcement learning (RL), this learning approach became dominant in solving

The work presented in this chapter has been partially published in [23].

the offloading decisions in vehicular scenarios. Compared to traditional approaches based on heuristics, these solutions have shown the ability to learn the best strategies adapting to the challenging and highly varying environments [172–174, 23].

Inspired by these positive results, we addressed the task offloading problem from two different perspectives. In this chapter we present our RL-based offloading strategy, while we leave the discussion of a more lightweight ML-based strategy or the next chapter (Chapter 7).

6.1 Collaborative RL for task offloading

To enable learning in an unknown environment, reinforcement learning (RL) has been shown as a promising solution, which can help overcome the prohibitive computational requirements. Recent RL-based online offloading decisions solutions have demonstrated improvements compared to traditional approaches, e.g., [172–174]. However, none of them take full advantage of a possible collaborative framework and decisions are taken independently by each agent of the system.

To this end, we propose the use of multi-agent reinforcement learning (MARL) to jointly improve the energy efficiency (EE) and task completion time of edge computing enabled UAVs swarms, while considering distributed offloading decision strategies. The proposed MARL algorithm can solve the computation offloading optimization problem in real-time by combining information coming from other devices, i.e., in a collaborative way, in order to decide if computing a task locally or offloading it to the closest edge cloud. In the case of offloading, the second decision entails the radio access technology (RAT) to consume, i.e., Wi-Fi or cellular, to transmit the task from the device to the edge cloud.

The presented decentralized algorithm leverages the actor-critic framework and is applicable to large-scale problems where both the number of states and the number of agents are massively large. Specifically, the actor step is performed individually by each agent with no need to communicate and infer the policies of other agents. On the other hand, for the critic step, each agent shares its estimate of the value function with its neighbors in order to achieve a consensual estimate, further used in the subsequent actor step. In this regard, the local information at each agent is able to diffuse across the network, making the network-wide maximum reward achievable.

As in standard distributed algorithms over networked systems, our algorithm provides the advantages of scalability to a large number of agents, robustness against malicious attacks, and communication efficiency.

6.2 Related Work

The problem of shortening task completion time by exploiting the close edge cloud is crucial for any type of IoT network in general, and robotic or drone networks in particular; so it is not surprising that there are several proposed solutions to tackle this problem. In the following, we first analyze the class for the RL model exploited by our solution and the differences between our implementation with previous approaches. Secondly, we cite a few representative (centralized and distributed) solutions to clarify our contributions to the decision task offloading problem.

In the last years, edge computing has been proved to be an effective method in supporting some latency-critical tasks [175, 176]. This paradigm can be particularly beneficial for UAV swarms, or in general unmanned aerial systems (UAS), e.g., self-driving vehicles, to conduct a computation offloading scheme with edge computing. Edge computing-based UAV swarms [177], are able to improve the latency and energy-efficiency issues caused by cloud computing [173]. In general, using ML/AI to optimize offloading process in vehicular environments has gained the attention in recent studies [178–180].

The minimization of transmission energy for single-user MEC systems, for instance, has been addressed under specific latency constraints in [181, 182]. Furthermore, in [166] the authors presented a game-theoretic approach to distributed offload computation among mobile device users, modeling the problem as a multi-user offloading game. You et al. [167] conceived a solution that determines the offloading data volume, the offloading duration, and the transmission resources of each user in an energy-efficient manner. Kalatzis et al. [183] decreased energy consumption in UAV based forest fire detection applications by adopting the edge and fog computing principles. However, such approaches fail in addressing the dynamicity of the environment, which is one of the main features of disaster scenarios, and hinders from high long-term performance.

Some researches studied the online computation offloading problem when edge computing resources are available. For instance, a task offloading solution built upon rent/buy problem aiming to minimize the task completion time in mobile clouds has been presented in [184]. At the same time, another recent trend is the utilization of RL in these circumstances, given its ability to adapt to highly dynamic environments [185, 186]. Huang et al. [172] proposed a deep reinforcement learning-based online offloading framework (DROO) to decide whether to offload tasks to the edge cloud and proportionally allocate wireless resources. Despite the similarity in the RL framework, our work differs from this class of solutions for the distributed nature that leads to multiple heterogeneous agents with potentially distinct policies and rewards, and the further improvements on protocol decisions. Besides, although distributed approaches in task offloading decisions leveraging deep RL exist, e.g., DDLO [187] and a hotbooting Q-learning based schema [188], these solutions use multiple parallel deep neural networks, rather than collaboratively take offloading decisions.

6.3 Model and Problem Definition

In this section, we first present some preliminary notions on actor-critic and multi-agent reinforcement learning (Section 6.3.1), used in our UAV task offloading model (Section 6.3.2 and Section 6.3.3) and problem definition (Section 6.3.4).

6.3.1 Background on Actor-Critic and Multi-agent Reinforcement Learning

Before describing the details and the notation of our model, we first describe the actor-critic framework and the MARL concepts, where our system is built upon.

Actor-Critic Algorithm. The Actor-Critic belongs to the class of model-free, online, on-policy reinforcement learning methods. The goal of an agent is to optimize the policy (actor) directly and train a critic to estimate the return or future rewards. Hence, at the very basis, a Markov decision process exists and is characterized by a quadruple $\mathscr{C} = \langle S, A, P, R \rangle$, where S denotes the finite state space, A is the finite action space, $P(s'|s,a) : S \times A \times S \rightarrow [0,1]$ refers to the state transition probability

from state s to state s' determined by action a, and $R(s,a) : S \times A \to \mathbb{R}$ is the reward function defined by $R(s,a) = \mathbb{E}[r_{t+1}|s_t = s, a_t = a]$, where r_{t+1} is the instantaneous reward at time t. The probability of choosing action a at state s is the policy of the agent, defined as the mapping $\pi : S \times A \to [0,1]$. The agent has the objective of finding the optimal policy that maximizes the expected time-average reward, i.e., the long-term return, which is given by $J(\pi)$:

$$J(\pi) = \lim_{T} \frac{1}{T} \sum_{t=0}^{T-1} \mathbb{E}[r_{t+1}] = \sum_{s \in S} d_\pi(s) \sum_{a \in A} \pi(s,a) R(s,a), \tag{6.1}$$

where $d_\pi(s) = \lim_{t\to\infty} \mathbb{P}(s_t = s|\pi)$ represents the stationary distribution of the Markov chain under policy π. Such a distribution $d_\pi(s)$ and the limit in (6.1) are well defined when the Markov chain resulting from the Markov Decision Process (MDP) is irreducible and aperiodic with any policy π.

Given any policy π, the action-value associated with the state s and action a, $Q_\pi(s,a)$, is thus defined, according to [189], as:

$$Q_\pi(s,a) = \sum_t \mathbb{E}[r_{t+1} - J(\pi)|s_0 = s, a_0 = a, \pi]. \tag{6.2}$$

Furthermore, the state-value associated with state s under policy π can be defined as $V_\pi(s) = \sum_{a\in A} \pi(s,a) Q_\pi(s,a)$. In the following, we simply refer to $Q_\pi(s,a)$ and $V_\pi(s)$ as *action-value* and *state-value* functions respectively. When the action or state spaces are massively large, these two functions are usually approximated by some parameterized functions $Q(\cdot,\cdot;\omega)$ and $V(\cdot;\nu)$, depending on the parameters ω and ν. Also the policy π can be parameterized by parameter θ in π_θ. For the sake of simplicity, hereafter we replace the subscript π_θ with just θ, e.g., V_{π_θ} to V_θ.

Actor-critic (AC) algorithms have been advocated to solve, with this parameterization, the optimal policy π_θ. Built on the well-known policy gradient theorem [190], AC algorithms are characterized by the gradient of the return $J(\theta)$ written as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d_\theta, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot (Q_\theta(s,a) - b(s))], \tag{6.3}$$

where the term $b(s)$ is commonly named baseline, and $\nabla_\theta \log \pi_\theta(s,a)$ is referred as the score function for policy π_θ. Also, let the advantage function be:

$$A_\theta(s,a) = Q_\theta(s,a) - V_\theta, \tag{6.4}$$

which specifies how much better it is to take a specific action compared to the average, general action at the given state. Indeed, it has been recognized, e.g., in [191], that the minimum variance baseline in the action-value function estimator is the state-value function $V_\theta(s)$. Defining $Q_t(\omega) = Q(s_t, a_t; \omega)$ at time t, and let A_t the sample at time t of the advantage function, we get:

$$A_t = Q(s_t, a_t; \omega_t) - \sum_{a \in A} \pi_{\theta_t}(s_t, a) Q(s_t, a; \omega_t), \tag{6.5}$$

Let then $\psi_t = \nabla_\theta \log \pi_{\theta_t}(s_t, a_t)$ be the sample of the score function. The AC algorithm based on the action-value function approximation is based on the following updates:

$$\begin{aligned} \mu_{t+1} &= (1 - \xi_{\omega,t}) \cdot \mu_t + \xi_{\omega,t} \cdot r_{t+1}, \\ \omega_{t+1} &= \omega_t + \xi_{\omega,t} \cdot \delta_t \cdot \nabla_\omega Q_t(\omega_t), \\ \theta_{t+1} &= \theta_t + \xi_{\theta,t} \cdot A_t \cdot \psi_t, \end{aligned} \tag{6.6}$$

where $\xi_{\omega,t}, \xi_{\theta,t} > 0$ are the stepsizes, μ_t tracks the unbiased estimate of the average return, and δ_t refers to the action-value temporal difference (TD) error and is defined as:

$$\delta_t = r_{t+1} - \mu_t + Q(s_{t+1}, a_{t+1}; \omega_t), \tag{6.7}$$

where action a_{t+1} is retrieved from the policy $\pi_{\theta_t}(s_{t+1}, \cdot)$. This TD error is used to evaluate the action just selected, i.e., the action a_t taken in state s_t. A positive TD error suggests that the tendency to select this action should be strengthened for the future, whereas a negative TD error suggests the tendency should be weakened.

The standard AC algorithm is defined as a two-time-scale algorithm, where the two stepsizes are set such that $\lim_{t \to \infty} \xi_{\omega,t} \cdot \xi_{\theta,t}^{-1} > 0$. The first two updates in (6.6) belongs to the *critic step*, which operates at a faster time scale; while the last update in (6.6) corresponds to the *actor step* that occurs at a slower time scale. The actor controls how our agent behaves by improving the policy along the gradient ascent direction; on the other hand, the critic measures how good is the action taken, by estimating the action-value function under policy π_{θ_t}.

Finally, actor-critic algorithms are able to achieve state-of-the-art performance in many complicated application domains, as shown in [192–194]. Inspired by these achievements, we further define a MARL algorithm based on the AC approach.

Multi-Agent Reinforcement Learning. We consider now a system of N agents operating in a common environment with no central controller that either collects rewards or makes the decisions for the agents. In this context, the set of agents is denoted by $\mathcal{N}_t$, whose cardinality is N, and each agent can communicate with each other. In general, the set of agents is a time-varying set, defined as $\mathcal{N}_t$ at time $t \in \mathbb{N}$.

A time-varying multi-agent MDP is defined as a tuple $(S, \{A^i\}_{i\in\mathcal{N}}, P, \{R^i\}_{i\in\mathcal{N}}, \{\mathcal{N}_t\}_{t\geq 0})$, where S denotes the global state space shared by all the agents in $\mathcal{N}_t$, and A^i is the action set that agent i can execute. Besides, let $A = \prod_{i=1}^{N} A^i$ be the joint action space of all agents, also referred to as global action profile. We then define $R^i : S \times A \to \mathbb{R}$ the local reward function of agent i, while $P : S \times A \times S \to [0,1]$ is the state transition probability. In this system, we assume that the states and the joint actions are globally observable, while the rewards are observed only locally.

At time step t, assuming the global state space is $s_t \in S$ and the joint actions of agents are $a_t = (a_t^1, \dots, a_t^N) \in A$, each agent will receive a reward r_{t+1}^i, which is a random value with $R^i_{(s_t,a_t)}$ as expected value. Also, the model shift to the new state $s_{t+1} \in S$ with probability $P(s_{t+1}|s_t, a_t)$. Our model is considered as fully decentralized since the reward is locally received and the action is performed locally by each agent.

As the state space S may be large, it is convenient to consider policies that are in a parametric function class, similar to the single AC. For agent i the local policy is then given by $\pi^i_{\theta^i}$, where $\theta^i \in \Theta^i$ is the parameter, and $\Theta^i \subseteq \mathbb{R}^{R_i}$ is a compact set. We then pack these parameters altogether in $\theta = [(\theta^1)^T, \dots, (\theta^N)^T] \in \Theta$, where $\Theta = \prod_{i=1}^{N} \Theta^i$. Therefore, the joint policy is given by $\pi_\theta(s,a) = \prod_{i=1}^{N} \pi^i_{\theta^i}(s, a_i)$, and is often shortened as π_θ.

Joint objective of the agents is to collaboratively find the joint policy π_θ that maximizes the globally averaged long-term return based solely on local information. The optimization problem to solve is:

$$\begin{aligned} \max_\theta J(\theta) = \lim_T \frac{1}{T}\mathbb{E}\left[\sum_{t=0}^{T-1} \frac{1}{N} \sum_{i\in\mathcal{N}} r^i_{t+1}\right] = \\ \sum_{s\in S} d_\theta(s) \sum_{a\in A} \pi_\theta(s,a) \cdot \overline{R}(s,a), \end{aligned} \tag{6.8}$$

where $\overline{R}(s,a) = N^{-1} \cdot \sum R(s,a)$ is the globally averaged reward function. Further, given $\overline{r}_t = N^{-1} \cdot \sum_{i\in\mathcal{N}} r^i_t$, it yields $\overline{R}(s,a) = \mathbb{E}[\overline{r}_{t+1} | s_t = s, a_t = a]$. Hence, the global

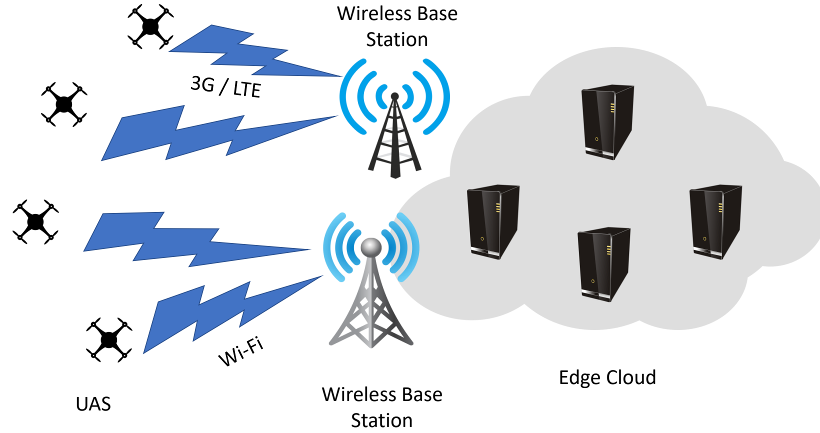


Fig. 6.1 System Overview: mobile devices, e.g., UAVs, interaction with the edge cloud via cellular and Wi-Fi network.

expected action value function for a state-action pair (s,a) under policy π_θ is:

$$Q_\theta(s,a) = \sum_t \mathbb{E}\left[\bar{r}_{t+1} - J(\theta) | s_0 = s, a_0 = a, \pi_\theta\right], \tag{6.9}$$

Finally, the global state-value function $V_\theta(s)$ is given by $V_\theta(s) = \sum_{a \in A} \pi_\theta(s,a) Q_\theta(s,a)$.

6.3.2 System Model

As shown in Fig. 6.1, we consider a UAV swarm consisting of a set of agents $\mathcal{N}_t = \{A_1, \ldots, A_N\}$, each of which has a task to be completed. We consider that the set $\mathcal{N}_t$ can change over time since the agents may suffer failures or running out of power. However, for simplicity, we often refer to this set as $\mathcal{N}$ in the following, without any ambiguity.

The overall system is compound of M tasks, denoted by a set of tasks $\mathcal{M} = \{T_1, \ldots, T_M\}$. The mobile node can either compute the task locally or offload the computation to the edge cloud in two ways, i.e., through a mobile network (LTE) or through Wi-Fi access points. In this case, we consider an application where tasks are independent.

Communication Model

As mentioned earlier, the access point for wireless communication can be either a Wi-Fi access point, or a base-station in cellular networks. The channel from mobile node i to access point s follows quasi-static block fading.

Let $o^1_{i,m}$ denote the computation offloading decision of task m of mobile device n. Specifically, $o^1_{i,m} = 1$ means that the node offloads the task via the wireless channel, while $o^1_{i,m} = 0$ means that the node performs the task locally on its own device. When task is set to be performed at the edge cloud, the communication can occur over cellular (e.g., LTE) network if $o^2_{i,m} = 1$ or Wi-Fi network for $o^2_{i,m} = 0$. Given the global action profile A for any node i and task m, we can compute the uplink data rate for computation offloading over cellular technology of task m of mobile device i as:

$$Rt^c_{i,m}(A) = W^c \cdot \log_2\left(1 + \frac{P^c_{i,m}H^c_{i,m}}{(\sigma^c_{i,m})^2 + \sum\limits_{j\neq i,k\neq m,o^1_{j,k}=1,o^2_{j,k}=1} P^c_{j,k}H^c_{j,k}}\right), \quad (6.10)$$

where $P^c_{i,m}$ is the transmission power of node i offloading task m to the edge cloud via cellular connectivity; $H^c_{i,m}$ denotes the channel gain from node i to access point s when transmitting task m due to the path loss and shadowing attenuation; $(\sigma^c_{i,m})^2$ indicates the thermal noise power associated with the link between the node i and the access point s, and W^c is cellular channel bandwidth. From (6.10) we can observe that when many mobile devices offload their tasks via cellular access simultaneously, they may lead to severe interference and low data rates.

Likewise, we define the uplink rate of Wi-Fi network similar to the cellular transmission as follows:

$$Rt^w_{i,m}(A) = W^w \cdot \log_2\left(1 + \frac{P^w_{i,m}H^w_{i,m}}{(\sigma^w_{i,m})^2 + \sum\limits_{j\neq i,k\neq m,o^1_{j,k}=1,o^2_{j,k}=0} P^w_{j,k}H^w_{j,k}}\right), \quad (6.11)$$

where the involved variables have the same meaning of those in (6.10).

Computation Model

Let $D_{i,m}$ denote the size of computation data (e.g., the recorded audio in UAVs swarm) related to computation task m of node i. $L_{i,m}$ denotes the computing workload, i.e., the total number of CPU cycles needed to accomplish task m of node i. In the following, we consider the computation overhead in terms of energy consumption and application completion time for local and edge cloud computing. Further, we differentiate the edge offloading into two cases, that represent the two possible communication options: cellular and Wi-Fi networks.

Local Computing Mode. We denote the computation capability, i.e., the clock frequency of the CPU chip, of node i, on task m, as $f_{i,m}$. Our model allow different mobile devices to have different computation capability with different clock frequency per task. The local execution time of task m on node i is hence given by:

$$T^{l,exec}_{i,m} = \frac{L_{i,m}}{f_{i,m}}, \tag{6.12}$$

while the energy consumption of the device is given by:

$$E^{l}_{i,m} = kL_{i,m}f^{2}_{i,m}, \tag{6.13}$$

where k denotes the effective switched capacitance for the specific chip architecture. In line with previous studies, e.g., [195, 196], we set $k = 10^{-11}$. Clearly, the clock frequency of the CPU chip can be adjusted by using the DVFS technique to achieve the optimum computation time and energy consumption on a device.

Aside the execution time, the time to complete task m is also affected by the *waiting time* $T^{wt}_{i,m}$. The waiting time of a task is defined as the time that task m spends on board of i before its execution.

Consequently, the completion time for a local execution of task m on node i is the sum of the local computation execution time and the waiting time in local computing,

$$T^{l}_{i,m} = T^{l,exec}_{i,m} + T^{wt}_{i,m}. \tag{6.14}$$

We are now ready to introduce the computational cost of a task, which dictates our energy-efficient strategy.

Definition 6.3.1. *Computational Cost. The computational cost is defined as the weighted sum of energy consumption and completion time related to the execution of a task m belonging to node i.*

In the case of local execution, it is given by:

$$Z^l_{i,m} = \alpha^l_{i,m} T^l_{i,m} + \beta^l_{i,m} E^l_{i,m}, \tag{6.15}$$

where $\alpha^l_{i,m}$ and $\beta^l_{i,m}$ are the weights for the energy consumption and the computation completion time respectively.

This form of computational cost enables to meet different user demands by adjusting the weights, and, for example, save more energy rather than shortening the delay. For delay-sensitive applications, such as rapid disaster response set-up, a larger $\beta^l_{i,m}$ is recommended to meet the strict user requirements. In this regard, the weights control the importance of the perceived latency and energy consumption respectively.

Edge Computing Mode. In case the mobile node i offloads the computation task m to the edge cloud, the latter executes the computation task and returns the results to the device. When the task is offloaded to the edge cloud, the execution entrails three phases: *(i)* the transmission phase, *(ii)* the edge computation phase, *(iii)* the outcome receiving phase.

Starting with the first phase, we consider the time and energy consumed during transmission. In line with the computation and communication model, we can define the transmission time and energy consumption for task offloading over cellular network as:

$$T^{c,tra}_{i,m}(A) = \frac{D_{i,m}}{Rt^c_{i,m}(A)}, \tag{6.16}$$

$$E^{c,tra}_{i,m}(A) = P^c_{i,m} T^{c,tra}_{i,m}(A), \tag{6.17}$$

respectively. The transmission over Wi-Fi technology entails different transmission time and energy consumption, as follows:

$$T^{w,tra}_{i,m}(A) = \frac{D_{i,m}}{Rt^w_{i,m}(A)}, \tag{6.18}$$

$$E^{w,tra}_{i,m}(A) = P^w_{i,m} T^{w,tra}_{i,m}(A). \tag{6.19}$$

Besides, for edge cloud execution we can derive the computation execution time for task m of node i as:

$$T_{i,m}^{e,exec} = \frac{L_{i,m}}{f_e}, \tag{6.20}$$

where f_e refers to the clock frequency of the edge cloud. In this case, the assumption is that the frequency does not change during the computation and is constant over time. Moreover, we assume the energy consumption in the edge cloud is negligible since the cloud is in general powered by alternating current and has enough energy to execute the offloaded tasks. Although the offloaded task needs to wait before it is assigned to the proper resource in the cloud for the execution, we omit this waiting time for simplicity, as it is negligible with respect to the other quantities involved. Finally, as it is done in several other studies, e.g., [197, 198], we ignore the time for receiving the outcome of task m, since the received data is typically small. As such, the completion time for the edge offloading is the sum of the execution time and the transmission time over the wireless channel. For the cellular case we have:

$$T_{i,m}^{c} = T_{i,m}^{c,tra}(A) + T_{i,m}^{e,exec}. \tag{6.21}$$

On the other hand, if the offloading is performed over the Wi-Fi network, the completion time is computed as:

$$T_{i,m}^{w} = T_{i,m}^{w,tra}(A) + T_{i,m}^{e,exec}. \tag{6.22}$$

Consequently, the *computational cost* of task m of node i on the edge cloud through the cellular network is:

$$Z_{i,m}^{c} = \alpha_{i,m}^{c} T_{i,m}^{c} + \beta_{i,m}^{c} E_{i,m}^{c,tra}(A), \tag{6.23}$$

where a small data transmission rate $Rt_{i,m}(A)^c$ of the device i would result in high energy consumption in the wireless communication and long transmission time for offloading data to the closest edge cloud.

Similarly, we define the *computational cost* for the Wi-Fi offloading as:

$$Z_{i,m}^{w} = \alpha_{i,m}^{w} T_{i,m}^{w} + \beta_{i,m}^{w} E_{i,m}^{w,tra}(A), \tag{6.24}$$

where the weights may differ from the ones utilized in (6.23).

Notably, to enable diversity among the three cases in the importance of latency with respect to the energy, the computational costs have different weights, depending on where the task is performed. That is, $\alpha_{i,m}^{w}$ is not necessarily equal to $\alpha_{i,m}^{c}$ and $\alpha_{i,m}^{l}$. Likewise for $\beta_{i,m}^{w}$, $\beta_{i,m}^{c}$ and $\beta_{i,m}^{l}$.

6.3.3 MARL Framework Formulation

Following the standard notation for reinforcement learning algorithms, we define the *state space* as the set of metrics used to select the best action among all the actions defined in the *action space*. The action selection occurs with the aim of maximizing a *reward function*, which represents the objective (utility) to optimize.

State Space. We report in Table 6.1 the features adopted to build our model state space. For each agent i in the network, we save the shown metrics for cellular and Wi-Fi communications. The first information esteems the distance between the agent and the base station, and is the same for both Wi-Fi and cellular transmissions. The subsequent features consider the quality of the signal, the throughput, the round-trip-time (RTT), and the loss rate, for the cellular and Wi-Fi channels separately. These quantities change over time as effect of the single and combined actions of the

Table 6.1 The contextual metrics gathered for building the state space.

	Features	Description
1	d_i	Distance between agent i and base station [m]
2	q_i^c	Cellular Reference Signal Received Quality (RSRQ) [dB]
3	i_i^w	Wi-Fi Received Signal Strength Indicator (RSSI) [dB]
4	t_i^c	Cellular throughput [kbps]
5	t_i^w	Wi-Fi throughput [kbps]
6	r_i^c	Cellular RTT [ms]
7	$r_i^w i$	Wi-Fi RTT [ms]
8	l_i^c	Cellular lossrate [%]
9	$l_i^w i$	Wi-Fi lossrate [%]

system, so we define the state space at time t as s_t.

The choice of such features is dictated by a design goal of balancing the overhead introduced by the metrics collection and the precision in grasping the system conditions. Empirically, we found that this state set produces the optimal trade-off,

as also outlined by the goodness of our results (Section 6.5). It can be noted, indeed, as part of these quantities are already captured by the TCP protocol and form its state. Thus, our solution can easily leverage these quantities, reducing the overhead.

Action Space. The main decision that the agent is supposed to take, is whether or not to offload the task to the edge cloud. Formally, the first decision for each agent i is the binary offloading decision $o^1_{i,m}$:

$$o^1_{i,m} = \begin{cases} 1, & \text{if tasks are to be offloaded} \\ 0, & \text{otherwise.} \end{cases}$$

If $o^1_{i,m} = 0$, task is computed locally, whereas for $o^1_{i,m} = 1$ the incoming task is offloaded to the closest station. In the latter case, the subsequent decision regards the technology on which the transmission occurs. In fact, as the offloading occurs, the protocol and technology for transmitting bytes are extremely relevant for shortening the latency. With this respect, we define a second binary decision $o^2_{i,m}$:

$$o^2_{i,m} = \begin{cases} 1, & \text{if cellular technology is preferred} \\ 0, & \text{if Wi-Fi technology is preferred.} \end{cases}$$

Such a decision takes place only for an $o^1_{i,m} = 1$, and we can observe how the total number of actions for each agent i is three, for an action set as follows: $A_i = [a^1_i, a^2_i, a^3_i]$, where a^1_i denotes $o^1_{i,m} = 0$, a^2_i is $o^1_{i,m} = 1, o^2_{i,m} = 0$, and a^3_i is $o^1_{i,m} = 1, o^2_{i,m} = 1$.

Utility function (RL reward). Based on reinforcement learning, the agent selects the action with the highest global reward. This choice relies upon the utility function, that specifies the objective of our algorithm. While RL can take a variety of different objectives, we define a function as follows to minimize the total latency and the usage of resources:

$$U_{i,m} = -o^1_{i,m} o^2_{i,m} Z^c_{i,m} - o^1_{i,m}(1 - o^2_{i,m}) Z^w_{i,m} - (1 - o^1_{i,m}) Z^l_{i,m}, \tag{6.25}$$

where a high cost in terms of computational time and energy consumption leads to small utility value. Further, we can easily define the utility per agent for all tasks as

follows

$$U_i = \sum_{m=1}^{M} U_{i,m}. \tag{6.26}$$

The utility function is the real objective that each agent attempts to optimize; still, its value cannot be used to specify the desirability of the action taken in a particular state and hence cannot be directly used as a *reward* for the learning process [144]. The ambiguity in action evaluation comes from the unique dynamic network environment the learning agent is interacting with, and it means we cannot merely take the utility value to define the reward. For this reason, we consider the difference between consecutive utility values as the reward. This is because an increase in the utility value denotes an improvement and hence, the corresponding action should be encouraged, regardless of the original value of the utility. Consequently, we define the reward value as follows:

$$r_t^i = \begin{cases} a & \text{if } U_t^i - U_{t-1}^i > \varepsilon \\ b & \text{if } U_t^i - U_{t-1}^i < -\varepsilon \\ 0 & \text{otherwise,} \end{cases} \tag{6.27}$$

where U_t^i refers to the cumulative utility at time t, a is a positive value, and b is a negative value. Both indicate the reward (a reinforcement signal) given the direction of changes between two newly observed consecutive utility values, while ε is a tunable parameter that sets the sensitivity of the learning agent to changes in the utility values (i.e., it sets a tolerance in the value change).

It is worth noticing that each agent can potentially utilize a different reward, and the system can be easily extended towards this scenario. However, for the sake of simplicity, in the following we assume that all agents share the same utility.

6.3.4 Problem Formulation

Given the system model, we can formulate the optimization problem that our MARL algorithm aims to solve. First, let the computational cost of a sequence of tasks $\mathscr{M}$

for the mobile node i be:

$$Z_i = \sum_{m=1}^{M} Z_{i,m} = \sum_{m=1}^{M} \left(o^1_{i,m} o^2_{i,m} Z^c_{i,m} + o^1_{i,m}(1 - o^2_{i,m}) Z^w_{i,m} + (1 - o^1_{i,m}) Z^l_{i,m} \right), \quad (6.28)$$

where M is the size of the set $\mathscr{M}$.

Formally, we have the following optimization problem:

$$\min_{A} \sum_{i} Z_i \quad (6.29)$$

$$\text{s.t.} \quad o^1_{i,m} o^2_{i,m} T^c_{i,m} + o^1_{i,m}(1 - o^2_{i,m}) T^w_{i,m} + o^1_{i,m}(1 - o^2_{i,m}) T^w_{i,m} \leq T^{max}_m \quad \forall m = 1, \dots, M \quad (6.30)$$

where $A = \{o^1_{i,m}, o^2_{i,m} | i \in \mathscr{N}, m \in \mathscr{M}\}$. The constraint stated by (6.30) imposes that the total completion time of all the tasks is bounded by the required maximum completion time, T^{max}_m. This time deadline is application-specific, and can vary based on user needs.

The key challenge in solving the optimization problem is that the integer constraint of the device actions, i.e., $o^1_{i,m}, o^2_{i,m}$, makes the problem a mixed integer programming problem, which is generally non-convex and NP-hard. Thus, solving the problem by using a multi-agent reinforcement learning approach reduces complexity and allows reaching a feasible solution in polynomial time.

6.4 Our Algorithm

Based on the previous formulations, we design an algorithm to establish the offloading decision. An Actor-Critic (AC) algorithm comes with multiple flavours, e.g., Q Actor-Critic, Advantage Actor-Critic, TD-error Actor-Critic. Among them, we follow the TD-error variant for the computation of the Critic.

In the following we first show the formulation of the policy gradient in a multi-agent setting. Then, we present the proposed MARL algorithm for our decentralized multi-agent system.

6.4.1 MARL system optimization

We recall that $\pi_\theta : S \times A \to [0,1]$ is the derived joint policy for the packed weights of the neural networks $\theta \in \Theta$, the globally long-term averaged return is $J(\theta)$, and Q_θ and A_θ are the action-value function and advantage function, respectively. Then, for any $i \in \mathscr{N}$, we define the local advantage function $A^i_\theta : S \times A \to \mathbb{R}$ as:

$$A^i_\theta(s,a) = Q_\theta(s,a) - \widetilde{V^i_\theta}(s,a^{-i}), \tag{6.31}$$

where a^{-i} denotes actions of all agents except for agent i, and $\widetilde{V^i_\theta}(s,a^{-i}) = \sum_{a^i \in A^i} \pi^i_{\theta^i}(s,a^i) \cdot Q_\theta(s,a^i,a^-i)$. Given the outcome of the Policy Gradient Theorem for MARL systems [199], we can compute the gradient of $J(\theta)$ as follows:

$$\nabla_{\theta^i} J(\theta) = \mathbb{E}_{s\sim d_\theta, a\sim\pi_\theta}[\nabla_{\theta^i} \log \pi^i_{\theta^i}(s,a^i) \cdot A^i_\theta(s,a)] \tag{6.32}$$

This gradient is applied to $J(\theta)$, previously defined in (6.8).

This result is precious as it shows that the policy gradient with respect to each θ^i can also be computed locally using the corresponding score function $\nabla_{\theta^i} \log \pi^i_{\theta^i}(s,a^i)$. However, local information is insufficient to estimate the global action-value and the advantage functions. These functions are necessary to compute the gradient and they require the reward values $\{r^i_t\}_{i\in\mathscr{N}}$ of all agents. For this reason, our proposed algorithm fosters collaboration among the agents and includes a consensus-based phase to diffuse the local information among them.

6.4.2 Local Updates and Consensus-based phase

The AC algorithm consists of two steps that occur at different time scales. In the critic step, the update is similar to the action-value TD-learning in (6.6), followed by a linear combination of its neighbor's parameter estimates. This parameter sharing step is also known as the consensus update, and involves a weight matrix $C_t = [c_t(i,j)]_{N\times N}$, where $c_t(i,j)$ denotes the weight on the message transmitted from i to j at time t. In the following process, each agent only uses the transition at time t, i.e., sample (s_t, a_t, s_{t+1}) for updating the parameters. First, we estimate $J(\theta)$ and V_θ with, respectively, a scalar μ and a parameterized function $V(\cdot,v) : S \to \mathbb{R}$, where parameter $v \in \mathbb{R}^L$ with $L \ll |S|$. Each agent i shares local parameters μ^i and v^i, and

updates its information as follows:

$$\begin{aligned}
\widetilde{\mu}_t^i &= (1-\xi_{v,t})\cdot\mu_t^i + \xi_{v,t}\cdot r_{t+1}^i,\\
\mu_{t+1}^i &= \sum_{j\in\mathbb{N}} c_t(i,j)\cdot\widetilde{\mu}_t^j,\\
\delta_t^i &= r_{t+1}^i - \mu_t^i + V_{t+1}(v_t^i) - V_t(v_t^i),\\
\widetilde{v}_t^i &= v_t^i + \xi_{v,t}\cdot d_t^i\cdot\nabla_v V_t(v_t^i),\\
v_{t+1}^i &= \sum_{j\in\mathbb{N}} c_t(i,j)\cdot\widetilde{v}_t^j,
\end{aligned} \tag{6.33}$$

where, for the sake of simplicity, $V_t(v) = V(s_t;v)\forall v\in\mathbb{R}^L$, and $\xi_{v,t}>0$ is the stepsize. In this context, differently from the single AC case, δ_t^i denotes the state-value TD-error of agent i.

Given the globally averaged reward $\overline{R}(s,a) = N^{-1}\cdot\sum R(s,a)$, the agent estimates the value $\overline{R}(s,a)$ in the critic step. Formally, let $\overline{R}(\cdot,\cdot;\lambda): S\times A\to\mathbb{R}$ be the class of parameterized functions and $\lambda\in\mathbb{R}^M$ be the parameter with $M\ll|S|\cdot|A|$. Motivated by the distributed optimization literature [200, 201], in order to obtain the estimate of $\overline{R}(\cdot,\cdot;\lambda)$, we minimize the following weighted mean-square error at the faster time scale:

$$\min_\lambda \sum_{i\in\mathcal{N}}\sum_{s\in S,a\in A}\delta_\theta(s)\cdot\pi_\theta(s,a)\cdot\left[\overline{R}(s,a;\lambda)-R^i(s,a)\right], \tag{6.34}$$

where δ_θ refers to the stationary distribution of the Markov chain $\{s_t\}_{t\geq 0}$ under policy π_θ. To solve this minimization problem, the updates to λ_t^i are as follows:

$$\begin{aligned}
\widetilde{\lambda}_t^i &= \lambda_t^i + \xi_{v,t}\cdot\left[r_{t+1}^i - \overline{R}_t(\lambda_t^i)\cdot\nabla_\lambda\overline{R}_t(\lambda_t^i)\right],\\
\lambda_{t+1}^i &= \sum_{j\in\mathbb{N}} c_t(i,j)\cdot\widetilde{\lambda}_t^j,
\end{aligned} \tag{6.35}$$

where $\overline{R}_t(\lambda)$ is a compact notation for $\overline{R}_t(s,a;\lambda)$. It is worth noticing that this procedure preserves the privacy of agents on their rewards and policies, since the rewards of other agents are not transmitted and the estimate $\overline{R}(\cdot,\cdot;\lambda)$ cannot be used to reconstruct original reward of other agents.

The updates in (6.35), (6.33) forms the critic step. On the other hand, the actor step uses the estimate $\overline{R}_t(\lambda^i)$ to evaluate the globally averaged TD-error $\widetilde{\delta}_t^i$ and

performs the updates:

$$\begin{aligned}\widetilde{\delta}_t^i &= \overline{R}_t(\lambda_t^i) - \mu_t^i + V_{t+1}(v_t^i) - V_t(v_t^i),\\ \theta_{t+1}^i &= \theta_t^i + \xi_{\theta,t} \cdot \widetilde{\delta}_t^i \cdot \psi_t^i,\end{aligned} \tag{6.36}$$

where ψ_t^i is defined as $\psi_t^i = \nabla_{\theta^i} \log \pi_{\theta_t^i}^i(s_t, a_t^i)$ and $\xi_{\theta,t} > 0$ is the stepsize.

We summarize the steps of the presented algorithm in Algorithm 5. After a first initialization phase, the agents start the individual actor and critic steps. These steps occur with a period of Δt in order to not overload the agent itself, where the optimal Δt is selected via a sensitivity analysis (Section 6.5). The elaborated values are then sent to the neighbors, and upon receiving such values, each agent updates its parameters to embrace a global view of the action performed. The actor is a neural network working as a function approximator and its task is to produce the best action for a given state. The network shape is optimized empirically and motivated in the evaluation (Section 6.5). The critic is another function approximator, i.e., a neural network, which, receiving as input the environment and the action by the actor, outputs the action value (Q-value) for the given pair.

Given the values to be stored for critic and actor steps, online implementing this algorithm requires a memory complexity of $\mathscr{O}(N+L+M+R_i)$ for each agent i. This complexity results in a great benefit compared to the regular reinforcement learning algorithm, where a huge Q-table need to be stored in each agent for a large N.

6.5 Evaluation Results

6.5.1 Experimental Setup

To evaluate the proposed solution, we run extensive experiments on an emulated cloud edge system scenario where several agents (the UAVs) can offload tasks to the edge by means of either cellular or Wi-Fi communications. Each agent is represented by a process running in the system, while the edge cloud is replicated by means of a further process emulating the execution of offloaded tasks. Channel parameters regarding the cellular and Wi-Fi connections are obtained from a real dataset publicly available [202]. The LTE technology is considered as a reference for the cellular case.

Algorithm 5 MARL actor-critic

1: Initialize $\mu_0^i, \widetilde{\mu_0^i}, v_0^i, \widetilde{v_0^i}, \lambda_0^i, \widetilde{\lambda_0^i}, \theta_0^i, \forall i \in \mathcal{N}$
2: Initialize $s_0, \{\xi_{v,t}\}_{t\geq 0}, \{\xi_{\theta,t}\}_{t\geq 0}$
3: Each agent i implements $a_0^i \sim \pi_{\theta_0^i}(s_0; \cdot)$
4: Step counter $t \leftarrow 0$
5: **for** all $i \in \mathcal{N}$ **do**
6:     **if** queued tasks **then**
7:         **for all** tasks **do**
8:             Take action $a_t^i \sim \pi_{\theta_0^i}(s; \cdot)$
9:             **if** $a_t^i = a_i^1$ or $a_t^i = a_i^2$ **then**
10:                 Offload task m to the edge
11:             **else**
12:                 Compute task m locally
13:     **for** every interval Δt **do**
14:         Collect metrics that form state s_t
15:         Update $\widetilde{\mu_t^i}, \delta_t^i$ according to (6.33)
16:         Update $\widetilde{\lambda_t^i}$ according to (6.35)
17:         **Critic Step:** $\widetilde{v}_t^i \leftarrow v_t^i + \xi_{v,t} \cdot d_t^i \cdot \nabla_v V_t(v_t^i)$
18:         Update $\widetilde{\delta_t^i}$ according to (6.36)
19:         Update $\psi_t^i \leftarrow \nabla_{\theta^i} \log \pi_{\theta_t^i}^i(s_t, a_t^i)$
20:         **Actor Step:** $\theta_{t+1}^i \leftarrow \theta_t^i + \xi_{\theta,t} \cdot \widetilde{\delta_t^i} \cdot \psi_t^i$
21:         Send $\widetilde{\mu_t^i}, \widetilde{\lambda_t^i}, \widetilde{v}_t^i$ to the neighbors
22:         **Consensus Step:**
23:             $\mu_{t+1}^i \leftarrow \sum_{j\in\mathbb{N}} c_t(i,j) \cdot \widetilde{\mu}_t^j$
24:             $v_{t+1}^i \leftarrow \sum_{j\in\mathbb{N}} c_t(i,j) \cdot \widetilde{v}_t^j$
25:             $\lambda_{t+1}^i \leftarrow \sum_{j\in\mathbb{N}} c_t(i,j) \cdot \widetilde{\lambda}_t^j$
26:         $t \leftarrow t+1$
27: **close**;

To represent these channel conditions, we use the Mahimahi emulator [158], a recent network emulator that allows testing with real traces. First, we adapt the information of the dataset to the format accepted in Mahimahi, and then, we create two interfaces for each agent, one with LTE traces and one with Wi-Fi traces. The task arrival rate at the agent follows a uniform distribution, and in the case of edge offloading, each task transmission is performed running *TCP iperf3* over the emulated link for the size of transmitted data, $D_{i,m}$, of 7 MB.

The channel bandwidth is set to the default value available in Mahimahi (i.e., $W^w = 5$ MHz for the Wi-Fi access, and $W^c = 4$ MHz for LTE). The thermal noise power is set equal for the two technologies, as $(\sigma^c_{i,m})^2 = (\sigma^w_{i,m})^2 = 50$ dBm. For the channel gain we have $H_{i,m} = d^{\nu}_{i,s}$, where $d_{i,s}$ denotes the distance between mobile node i and access point s, and $\nu = 4$ is the path loss factor. We then simply set the default values of the weights defined in (6.15), (6.23), and (6.24), so that energy consumption and task completion time have an equivalent importance in the computational cost evaluation, i.e., $\alpha^l_{i,m} = \beta^l_{i,m} = \alpha^c_{i,m} = \beta^c_{i,m} = \alpha^w_{i,m} = \beta^w_{i,m} = 0.5$. For the sake of simplicity we also set $f_{i,m} = 2.3$ GHz for all nodes, $f_e = 3.4$ GHz, and if not otherwise specified, $L_{i,m} = 25 \times 10^9$. The other metrics change over time and are collected when needed. In the following evaluation, the average values are computed after 35 experiments.

Each agent maintains two neural networks for actor and critic, respectively, and both of them have one hidden layer, containing 64 neural units (this number is motivated in the following), and use ReLU as the activation function. While the output layer for the actor network is softmax, that for the critic network is linear. Considering the graph G_t of the N agents, in which, at first, all agents can communicate with the others, we create the consensus weight matrix C_t by normalizing the absolute Laplacian matrix of G_t to be doubly stochastic. The stepsizes for the actor and critic step are set as constants, respectively $\xi_{\theta,t} = 0.001$ and $\xi_{v,t} = 0.01$.

6.5.2 Trace-Driven Emulation Results

In the following experiments we compare our solution against other currently deployed algorithms. Among the related studies described in Section 6.2, we select as benchmarks the most similar algorithms using some variants of machine learning-

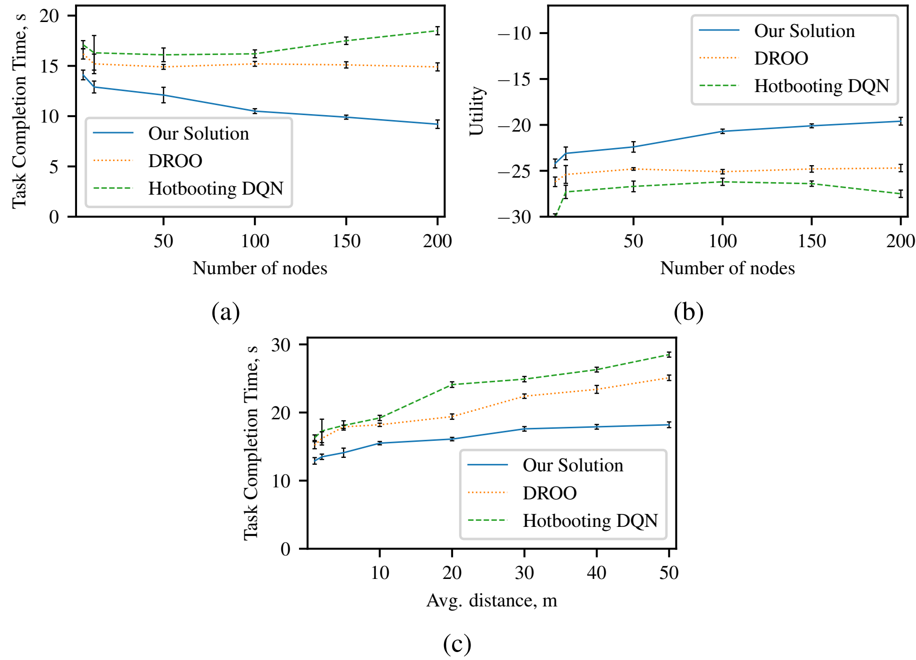


Fig. 6.2 System performance in terms of (a) task completion time and (b) utility for a varying number of agents and (c) node-antenna distance. The results are compared with similar solutions whose aim is analogous to ours.

based methods for the offloading process. Specifically, we compare our approach against the DROO framework [172], which implements a deep neural network that learns the binary offloading decisions, and a hotbooting Q-learning based computation offloading scheme [188], that for simplicity we refer to as hotbooting DQN, as it uses a fast deep Q-network (DQN) model to further improve the offloading performance.

Fig. 6.2a and Fig. 6.2b show the impact of the UAV swarm size (i.e., the number of agents) on the task completion time and on the utility function defined in (6.26), which also contemplates the power consumption. Decisions of each agent about whether to offload the task or not, as well as which technology to use for the offloading, are based on the information received from other nodes, according to the cooperative algorithm at the basis of our solution. We can notice how this approach can take full advantage of a rising number of computing nodes, shortening the task completion time and increasing the overall utility. Conversely, for hotbooting DQN occurs the opposite: if a large number of agents are present in the system, the task completion time increases. Besides, with an increasing number of computing nodes, power consumption increases as well. In the DROO case, the two quantities remain

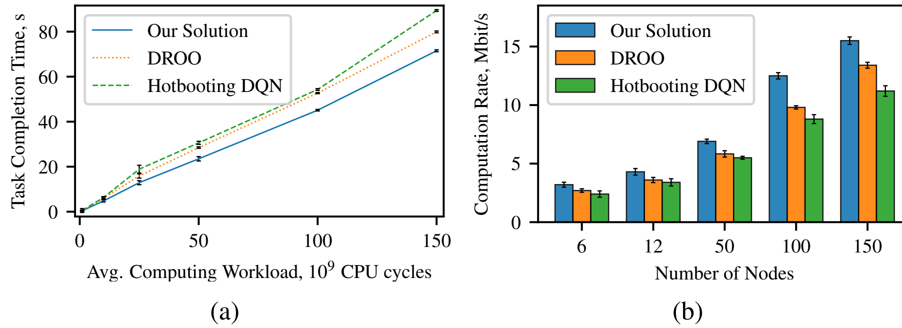


Fig. 6.3 System performance evaluation. (a) Application completion time for increasing average computing workload. (b) Comparison in terms of computation rate for various offloading solutions.

almost constant when the number of nodes increases, in any case leading this solution to perform worse than ours. These results show how a proper algorithm for task offloading decisions plays a crucial part in the system performance, and a multi-agent approach to optimize actions more efficiently is a valuable solution.

Besides the dependence on the number of agents, we further examine how the distance between nodes and the antenna affects the performance in Fig. 6.2c. We perform experiments for a fleet of 50 nodes, and we can observe how, clearly, the distance degrades the performance of the system because of the higher delays in the communication with the edge cloud. However, in the case of our solution, the curve is flattened, thus further proving its effectiveness in taking the offloading decision. In fact, our state space also includes the distance to the antenna, which is then considered in the decision process.

Moreover, we compare the performance of the three solutions with respect to the average computing workload, i.e., the average amount of CPU cycles required to complete the tasks submitted to the system. Fig. 6.3a depicts the energy consumption and the task completion time, respectively, for the three considered solutions. We can observe how the task completion time increases with the average computing workload, for all the considered solutions. However, for DROO and hotbooting DQN, the increment in the time is notably larger. This is because they do not have the adaptive and control mechanism of energy consumption we have in our model, which adaptively takes the offloading decision in a distributed manner.

In light of the previous findings, we can conclude that *the knowledge not only of the states, but also of some model parameters of the other agents (see Section 6.4),*

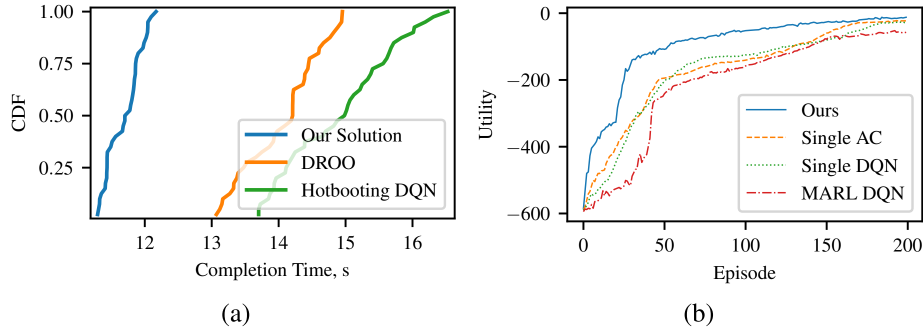


Fig. 6.4 (a) CDF of the task completion time for the compared solutions. (b) Utility evolution for different RL-based algorithms. Our model can shorten convergence time compared to the alternatives.

improves the decisions of the single agent. In fact, in this way the action can also consider the likely actions of other agents, thus possibly anticipating their future behavior.

Then, we evaluate the computation rate of all the agents, i.e., the number of processed bits within a unit time from the system. In Fig. 6.3b we report the computation rate for different algorithms at varying sizes of agents fleet. It is straightforward to observe how our algorithm outperforms the analogous approaches, and the more agents, the larger the rate improvements compared to the other methods. Although this metric is only implicitly covered by the utility function, our solution offers a high computation rate due to the optimized resource management and distributed approach. In fact, minimizing the computational time for tasks results in better computation rate performance too.

To analyze the variability of performance among nodes, we also evaluate the cumulative distribution function (CDF) of the task completion time for the three considered solutions. Results are reported in Fig. 6.4a and refer to a case when the number of nodes is 15. Not only does our approach provide a lower completion time on average, but most of the nodes complete the task at a time close to the average. This small variance is extremely important in UAV systems, especially for real-time applications requiring low and constant task completion times.

For the sake of completeness, we finally compare the convergence performance of our MARL-based method against other possible RL-based algorithms when applied in our solution. Specifically, we consider the following three alternative possibilities. Firstly, *Single AC*, an approach still based on the Actor-Critic (AC) framework, but

where each agent takes independent decisions. Secondly, *Single DQN*, a similar approach where the RL algorithm belongs to the class of Value-based methods that exploit Q-values to determine the probabilities of actions and any other parameter of the algorithm. In this class of algorithms, deep Q-network (DQN) is one of the most common methods that integrate deep neural networks into RL, originating the deep reinforcement learning. It has been shown how deep neural networks can empower RL to directly deal with high dimensional states thanks to techniques used in DQN [203]. Finally, *MARL DQN*, which implements the DQN algorithm in a multi-agent context, where the Q-values are transmitted among the agents for a collaborative approach. Fig. 6.4b shows the result of this comparison. It is possible to observe how the utility function increases as the number of episodes increases, until it attains a relatively stable value, in all the methods. However, we can notice that our approach provides a higher value for the utility function and that the convergence is faster. *MARL DQN*, for example, despite the cooperation among agents, is unable to properly handle the information of other nodes, whose learning process hardly fits this context. On the other hand, both *Single AC* and *Single DQN* have comparable yet better results with respect to *MARL DQN* due to the simplicity of their approach, which is able to achieve quite fast convergence. However, with local reward and action, classical reinforcement learning algorithms, i.e., *Single AC* and *Single DQN*, fail to maximize the system-wide average reward, whose value is determined by the joint actions of all agents. In conclusion, our algorithm can distribute the information in an efficient way, thus resulting in an appropriate solution for our context.

6.6 Conclusion

This chapter presents a distributed algorithm for the offloading task decision whose aim is to speed up the task completion time and, at the same time, limit the overall energy consumption. To this end, we propose a multi-agent reinforcement learning algorithm to decide whether or not to offload a task to the edge cloud. The overall state of the system is appropriately shared between the nodes and used when each agent has to decide where to perform an assigned task: locally or in the edge cloud by means of an offloading procedure. Each node, in case of task offloading, can further decide the transmission technology to use, Wi-Fi or LTE, according to the current utilization.

Results validate our algorithm, demonstrating the good performance of our system. Our evaluation also shows how the developed algorithm can manage the large quantity of information coming from the environment in an efficient way, thus making our distributed solution a truly viable approach for task offloading decision problems.

Chapter 7

A Self-Learning Strategy for Task Offloading in UAV Networks

Despite the good results of newly computation offloading techniques, however, RL-based methods have a severe impact on the memory and processing usage of the mobile nodes. Moreover, it still remains challenging to develop a reliable system that can anticipate future demands and take advisable computation offloading decisions.

In the following, we present a self-learning strategy that supports the UAV during the decision of offloading incoming tasks. This decision is taken on the basis of the predicted behavior of the agent, suggesting whether edge cloud is beneficial or not to the incoming tasks. Two alternative methods are designed to perform a prediction about future device load: a model belonging to time-series class, i.e., Vector Autoregressive Moving-Average (VARMA), and a model belonging to the class of ML regressors, i.e., Random Forest Regression (RFR). In such a way, not only the agent learns how to forecast future values, but it can also learn online what type of model is more accurate, following a paradigm known as Follow the Perturbed Leader (FPL). Having chosen two different ways in treating the input metrics, this approach also provides flexibility and adaptability, resulting in a learning agent that can select which predictor best fits a particular environment.

While other RL-based models can be computationally expensive to run on board of constrained resources devices, our formulation simplifies the decision process. The results illustrate clear advantages in the implementation of our approach, which

The work presented in this chapter has been partially published in [24].

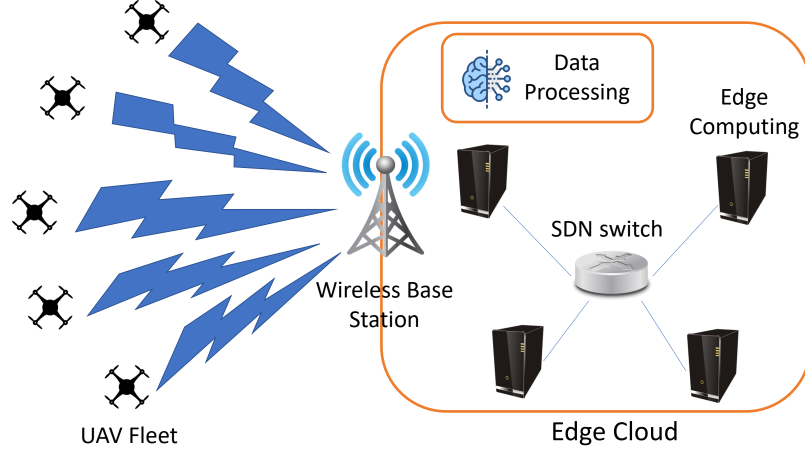


Fig. 7.1 System Overview. The mobile devices, e.g., UAVs, interacts with the edge cloud asking help for the processing of the collected data.

can shorten the time required to accomplish a task. Besides, our solution can reduce the energy consumed and the resource usage, i.e., memory and CPU, compared to other benchmark algorithms. These benefits are originated by our predictor, which outperforms alternatives, leading to a small error and a very accurate decision.

7.1 System Model

The considered system consists of a UAV swarm including a set of agents $\mathcal{N}_t = \{A_1,\ldots,A_N\}$, each of which has a task to be completed. We consider that the set $\mathcal{N}_t$ can change over time since the agents may suffer failures or running out of power. However, for simplicity, we often refer to this set as $\mathcal{N}$ in the following. The overall system is then composed of M tasks, denoted by a set of tasks $\mathcal{M} = \{T_1,\ldots,T_M\}$. We consider that tasks are independent among them. Each task is assigned to a node, which can decide either to compute the task locally or to offload the computation to the edge cloud.

To capture real-world scenarios, we consider a limited capacity of nodes. We model this constraint in resources as a finite queue where to store waiting tasks. Thus, we denote the amount of tasks of the i-th node as s_i within $[0,s_i^{max}]$, $s_i^{max} \in \mathbb{R}^+$. We also assume that the agent can execute only one task at a time, and, to avoid burdening the notation, task deadlines have not been considered.

For each new task arriving to a node, it has to decide where to perform the computation of such a task. We denote the computation offloading decision of task m of mobile device i by $o_{i,m}$. Specifically, $o_{i,m} = 1$ means that the node offloads the task to the close cloud, while $o_{i,m} = 0$ means that the node executes the task locally.

Table 7.1 Symbols and notations.

Symbol	Description
$\mathscr{N}_t$	Set of agents at time t
$\mathscr{M}$	Set of tasks
$T^{l}_{i,m}$	Task completion time for local computation
$T^{e}_{i,m}$	Task completion time for offloading computation
s^{max}_{i}	Maximum amount of possible enqueued tasks in node i
s_i	Number of enqueued tasks in node i
x_t	Observation at timestamp t
y_t	Prediction for timestamp t
$o_{i,m}$	Offloading decision of node i
$T^{l,exec}_{i,m}$	Local execution time of task m on node i
$C_{i,m}$	Computing workload of task m on node i
$T^{wt}_{i,m}$	Local waiting time of task m on node i
$T^{e,tra}_{i,m}(A)$	Transmission time of task m
$T^{e,exec}_{m}$	Execution time of task m in the edge cloud
$T^{e,rec}_{i,m}(A)$	Reception time of task m
W^{up}	Wireless uplink channel bandwidth
W^{down}	Wireless downlink channel bandwidth
$r^{e,up}_{i,m}(A)$	Wireless uplink data rate for offloading of task m
$r^{e,down}_{i,m}(A)$	Wireless downlink data rate for offloading of task m
σ^{up}	Background noise power in the uplink channel
σ^{down}	Background noise power in the downlink channel
$G^{up}_{i,m}$	Channel gain from node i to the access point
$G^{down}_{i,m}$	Channel gain from the access point to the node i
$p^{up}_{i,m}$	Transmission power of node i to the edge cloud
$p^{down}_{i,m}$	Transmission power of the edge cloud to the node i
$d_{i,s}$	Distance between node i and access point s

We summarize the main components of the system in Fig. 7.1. The UAV fleet relies on the close edge cloud for shortening the task completion time via task offloading. In such a case, the task is sent to the edge, where the appropriate network

and computational infrastructure resides. The sent data is thus used for extrapolating helpful information by means of AI/ML algorithms. For clarity, we report in Table 7.1 all the symbols used throughout the chapter.

7.1.1 Local Execution

When the task is locally executed, the completion time for a local execution of task m on node i is the sum of the local computation execution time and the waiting time aboard the agent,

$$T^{l}_{i,m} = T^{l,exec}_{i,m} + T^{wt}_{i,m}, \tag{7.1}$$

where $T^{l,exec}_{i,m}$ and $T^{wt}_{i,m}$ are the execution time and the waiting time, respectively. Formally, the waiting time of a task is defined as the time that task m spends on board of i before its execution, and mainly depends on the enqueued tasks.

On the other hand, given $C_{i,m}$ the computing workload, i.e., the total number of CPU cycles needed to accomplish task m of node i, the local execution time of task m on node i is hence given by:

$$T^{l,exec}_{i,m} = \frac{C_{i,m}}{f_{i,m}}, \tag{7.2}$$

where $f_{i,m}$ is the computation capability, i.e., the clock frequency of the CPU chip, of node i, on task m. Our model allows different mobile devices to have different computational capacities with different clock frequencies per task.

7.1.2 Edge Cloud Offloading

In case the mobile node offloads the task to the edge cloud, the latter executes the computation task and returns the results to the device. This process entails three phases: *(i)* the transmission phase, *(ii)* the edge computation phase, *(iii)* the outcome receiving phase. Before defining the resulting completion time, it must be noted that this time is affected by the joint action space of all agents, A, also referred to as global action profile. Therefore, given the global action profile A, the completion time for the edge offloading is the sum of these three phases, as such:

$$T^{e}_{i,m} = T^{e,tra}_{i,m}(A) + T^{e,exec}_{m} + T^{e,rec}_{i,m}(A), \tag{7.3}$$

where $T_{i,m}^{e,tra}(A)$ refers to the transmission of task m to the edge e; $T_m^{e,exec}$ is the execution time in the edge, and $T_{i,m}^{e,rec}(A)$ is the reception time.

Analyzing these parts in order, we start defining the transmission time for task offloading as:

$$T_{i,m}^{e,tra}(A) = \frac{D_{i,m}^{in}}{r_{i,m}^{e,up}(A)}, \tag{7.4}$$

where $D_{i,m}^{in}$ denotes the size of computation data sent over the channel (e.g., the recorded audio in UAVs swarm) related to computation task m of node i, and $r_{i,m}^{e,up}(A)$ is the uplink data rate.

Then, we consider the data rate affected by both the background noise power and the channel gain, as in other studies [168, 23]. Thus, given the global action profile A for any node i and task m, we can obtain the wireless uplink data rate for computation offloading of task m of mobile device i as:

$$r_{i,m}^{e,up}(A) = W^{up} \cdot \log_2\left(1 + \frac{p_{i,m}^{up} G_{i,m}^{up}}{\sigma^{up} + \sum\limits_{j \neq i, k \neq m, o_{j,k}=1} p_{j,k}^{c} G_{j,k}^{up}}\right), \tag{7.5}$$

where $p_{i,m}^{up}$ is the transmission power of node i offloading task m to the edge cloud; $G_{i,m}^{up}$ denotes the channel gain from node i to the access point when transmitting task m, mainly affected by the path loss and shadowing attenuation; σ^{up} indicates the background noise power, and W^{up} is the wireless uplink channel bandwidth. Clearly, we can observe from the formula that when many mobile devices offload their tasks to the edge simultaneously, the nodes can experience severe interference and low data rates.

Subsequently, the task arrives to the edge that proceeds with the execution. Although the offloaded task needs likely to wait before it is assigned to the proper resource in the cloud for the execution, in the following we omit this waiting time for simplicity, as it is negligible with respect to the other quantities involved. Thus, we can derive the computation execution time for task m in the edge cloud as:

$$T_m^{e,exec} = \frac{C_{i,m}}{f_e}, \tag{7.6}$$

where f_e denotes the clock frequency of the edge cloud, assuming that the frequency does not change during the computation and is constant over time.

Finally, the results of the computation is sent back to the mobile device, incurring in a reception time defined as:

$$T_{i,m}^{e,rec}(A) = \frac{D_{i,m}^{out}}{r_{i,m}^{e,down}(A)}, \tag{7.7}$$

where $D_{i,m}^{out}$ denotes the size of obtained output data sent over the channel and $r_{i,m}^{e,down}(A)$ is the downlink data rate. Such a wireless downlink data rate is given by:

$$r_{i,m}^{e,down}(A) \quad = \quad W^{down} \quad \cdot \quad \log_2\left(1 \quad + \quad \frac{p_{i,m}^{down} G_{i,m}^{down}}{\sigma^{down} + \sum\limits_{j \neq i, k \neq m, o_{j,k}=1} p_{j,k}^{c} G_{j,k}^{down}}\right), \tag{7.8}$$

where $p_{i,m}^{down}$ is the transmission power of the edge cloud communicating the results of offloaded task m to the node i; $G_{i,m}^{down}$ refers to the channel gain from the access point to the node i when transmitting data of task m; σ^{down} denotes the background noise power, and W^{down} indicates the wireless downlink channel bandwidth.

7.1.3 Problem Formulation

We formulate the optimization problem that aims to minimize the total delay in finishing all devices' tasks, by optimizing each node offloading decisions $o_{i,m}$:

$$\min_{o_{i,m}} \sum_{i \in \mathcal{N}} \sum_{m \in \mathcal{M}} (1 - o_{i,m}) T_{i,m}^{l} + o_{i,m} T_{i,m}^{e} \tag{7.9}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} o_{i,m} \leq 1 \quad \forall m \in \mathcal{M}, \tag{7.10}$$

$$\sum_{m=1}^{M} o_{i,m} \leq s_i^{max} \quad \forall i \in \mathcal{N}, \tag{7.11}$$

where the constraints (7.10) and (7.11) force the solution to *(i)* mutually choose if offloading task computation or executing the task locally, and *(ii)* not to exceed the resources of the mobile device, respectively.

The given optimization problem (7.9) - (7.11) can be solved to find results of offloading decision variables $o_{i,m}$. However, since the decision variables are binary, the formulated problem is not convex. Moreover, we would like to consider realistic scenarios where the interaction between devices, the communication channel conditions, and the nodes computation abilities are all dynamically changing. Given these considerations, in the following we propose a online learning method to solve this problem.

7.2 Regression Prediction Methods

With the aim of improving offloading decisions, each device predicts future conditions in order to verify is beneficial circumstances hold or not. The node can listen to the advice coming from two different class of predictors and obtain the best from both of them. In particular, we select two algorithms belonging to the class of time-series and to ML supervised regressors. In the following, we describe how these two methods behave, explaining why and where they differ.

7.2.1 Time-Series Analysis with VARMA

To model the evolution of data over time, we employ a Vector Autoregressive Moving-Average (VARMA) model. VARMA models are the multivariate generalization of univariate autoregressive-moving average (ARIMA) models. However, while ARIMA is used to represent stationary time series in almost all domains where a variable is measured at equidistant times, VARMA can contemplate multiple parallel time series, for a multivariate evolution. This class of models well fit problems in econometrics and financial markets, but boasts a wide exploration even in other fields since the 1970's [204]. Our solution, then, uses a VARMA model for "real time" model predictions (hindcasts) that are made within the independent dataset, using only data up to that date were used. The general form of VARMA(r,q) is given by

the following equation:

$$y_t = A_1 y_{t-1} + \ldots + A_r y_{t-r} + B_0 \varepsilon_t + \ldots + B_q \varepsilon_{t-q}, \quad (7.12)$$

where y_t denotes an $n \times 1$ vector of observed variables, ε_t is an $n \times 1$ vector of unobserved disturbances $\sim IID(O_{n\times 1}, I_n)$, where I_n denotes the $n \times n$ identity matrix, r and q denote any assumed nonnegative integers, such that at least one of r or q is positive.

In our solution, we predict the future values of the series by means of a forecasting method named minimization of the Mean Squared Forecast Error (MSFE), which denote the goodness of the prediction using the cumulative error encountered so far. The current information from the dataset, which constitutes the current knowledge, contains the current and past values of the series. In detail, we are focused on the one-step-ahead prediction, which just considers the prediction at the next time step, i.e., y_{t+1} given the last observation at time t.

7.2.2 ML Regression with RFR

The Random Forest Regression (RFR) is a type of additive model that predicts by combining decisions from a sequence of base models. More formally, this class of algorithms can be written as:

$$g(x) = f_0(x) + f_1(x) + f_2(x) + \ldots, \quad (7.13)$$

where the final model g is the sum of simple base models f_i. Although each base model f_i can be any ML algorithm, the most common version of RFR considers as f_i a simple decision tree. In this solution, we also consider this setting. This broad technique of using multiple models to obtain better predictive performance is also known as model ensembling. Moreover, in RFR, all the tree base models are constructed and trained independently using a different subset of data.

Predictions are, then, made by averaging the predictions of each decision tree. In other words, to extend the analogy—much like a forest is a collection of trees, the random forest model is also a collection of decision tree models. This makes random forests a strong modeling technique that is much more powerful than a single decision tree. RFR is suitable for regression problems given its features: *(i)*

it can capture non-linear or complex relationships between inputs and outputs, *(ii)* compared to a single decision tree, RFR is more robust, with a limited dependence to the noise in the training set, as it uses a set of uncorrelated decision trees, *(iii)* it is able to limit both the variance and the bias, better addressing the problem of overfitting.

7.2.3 State Variables in Our Solution

In our system, the current knowledge Y is modeled as a matrix of features, with a shape $N \times M$, where the column j represents the list of metrics gathered for task i, given i the index of the row. Such a list of features for task i are three: *(i)* the number of enqueued tasks when task i arrived to the node, *(ii)* time to complete task i, in seconds, *(iii)* a boolean stating if task i has been offloaded to the edge cloud.

Features selection is a key topic when dealing with big data, demanding for a trade-off between having a vast knowledge and time and resource constraints. This imposes to limit the complexity, with little or none effect on the performance. In fact, a smaller M yields simpler models, but it may be inadequate to represent the space of possible behaviors. On the other hand, a large M leads to a more complex model with more parameters, but may, in turn, lead to overfitting issues. While in time-series this choice for observed variables is much easier, as it usually entails the interested variable, in ML features selection is much more important. Our choice of $M = 3$ and metrics that are extremely easy to collect, moves towards this direction. Each node can train its model without the need to communicate with others, attaining null communication overhead and modest memory occupation. Results support this choice, achieving higher accuracy while reducing noise (Section 7.4).

However, given the differences in the two models, they also treat the input data differently. Regarding the VARMA model, its input $y_t, y_{t-1}, \dots, y_1$, is modeled with a vector of metrics at timestamp t. This means that the input of the VARMA model consists of all the values in the matrix. When the number of rows of Y exceeds a threshold ($Z = 1000$), the considered temporal window is limited to a sub-matrix consisting of the last Z rows. Alternatively, regarding the RFR, since it is agnostic of the time order, it only considers the last line of the matrix Y, i.e., the input of the model is composed by all the columns for the last row of the matrix.

7.3 Agent's Decision Process

In the following, we first overview the procedure as in the Follow the Perturbed Leader (FPL) method. Then, we describe how the IoT agent implements our version of FPL in our system.

7.3.1 Follow the Perturbed Leader

Learning from a constant flow of data is considered one of the central challenges of machine learning. Online learning entails sequentially decide on actions given the changes in the environments. In past years, a variety of online learning algorithms have been devised [205, 206]. Among them, in our work we investigate Follow the Perturbed Leader algorithm, whose advantage is its simplicity and computational efficiency.

Such a prediction with expert advice proceeds as follows. At each time step t the system performs sequential predictions $y_t \in \mathscr{Y}$. At times $t = 1, 2, ...$, we have access to the predictions $(y_t^i)_{1 \leq i \leq n}$ of n experts $\mathscr{E} = e_1, ..., e_n$. After having made a prediction, we receive observation $x_t \in \mathscr{X}$, and the system computes our suffered loss $l(x_t, y_t)$ and each expert's loss $l(x_t, y_t^i)$. As our observations entail continuous values, i.e., lie in a regression problem, the loss is calculated as: $l(x_t, y_t) = (y_t - x_t)^2$.

Our goal can be summarized in achieving a total loss "not much worse" than the best expert, after T time steps. More formally, we denote the cumulative loss of expert i by $L_i^T = \sum_{t=1}^T l(x_t, y_t^i)$ and the cumulative loss of our system by $L^T = \sum_{t=1}^T l(x_t, y_t)$. Thus, the goal of the system is to minimize the regret, defined as the difference between the cumulative loss of the learner and the cumulative loss of the best prediction in hindsight. The regret over T rounds is defined as:

$$R_T = \sum_{t=1}^{T} l(x_t, y_t) - \min_{i \in 1..n} \sum_{t=1}^{T} l(x_t, y_t^i) = L^T - \min L_i^T. \tag{7.14}$$

The term $\min L_i^T$, is often defined as the loss of the *best expert in hindsight (BEH)*. Moreover, when this regret is sublinear, namely $R_s \leq o(T)$, the learning algorithm is said to be *Hannan-consistent*.

One algorithm for achieving Hannan-consistency is Follow the Perturbed Leader (FPL), as demonstrated by Hannan [207] and Kalai and Vempala [208]. Let γ be some n-dimensional random variable and $\eta_i > 0$. FPL involves picking the expert *exp* that minimizes the perturbed cumulative loss:

$$exp = \arg\min_i (L_i + \eta \gamma_i) \tag{7.15}$$

Intuitively, if η is small, then we expect *exp* to be "close" to a minimizer of the (non-perturbed) cumulative loss. On the other hand, when η is large, we expect $\mathscr{E}$ to be "close" to the uniform distribution. Namely, η controls how similar the algorithm is to Follow the Leader, the version of the algorithm that always picks the expert who has minimized the cumulative loss. However, this version, and any other deterministic learning algorithm, is not Hannan-consistent [209].

We summarize our version of FPL in Algorithm 6. As demonstrated by theorems in [209, 210], for all possible sequences of losses where the loss is bounded and the noise is spread out, i.e. the noise has a sufficiently high variance, FPL achieves an expected regret that is bounded by:

$$\mathbb{E}[R_t] \in \mathscr{O}(\sqrt{T}). \tag{7.16}$$

Consequently, this result is also valid for our Algorithm 6, where we consider the random value γ sampled from a Gaussian distribution $\mathscr{N}(0, I)$.

Algorithm 6 Follow the perturbed leader

`fpl(m: task, s:state, t: time)`

1: $\eta > 0, L_i \leftarrow 0$
2: **for** every expert i **do**
3: Compute loss l of last prediction given evidence s
4: Accumulate the loss $L_i^t \leftarrow L_i^{t-1} + l$
5: Sample $\gamma_i \sim \mathscr{N}(0, I)$
6: $exp \leftarrow \arg\min_i (L_i^t + \eta \gamma_i)$
7: Predict y_t asking to the expert *exp* given state s and task m
8: **return** y_t

Furthermore, it can be noted that this problem is similar to the Multi-Armed Bandit (MAB), in the class of RL methods. FPL follows the "arm" that is assumed to have the best performance so far, adding exponential noise to it to provide exploration.

However, while the MAB algorithm offers more strict bounds to the regret, FPL can drastically simplify the entire learning process making it suitable for constrained agents as the UAVs. For example, differently from MAB that may take a long time to converge, FPL requires a shorter time, and mostly, is not eager of computation and memory resources as is the MAB.

7.3.2 Our Algorithm

We can now present our algorithm, built upon the Follow the Perturbed Leader (FPL) formulation, which dictates the offloading decision process. The overall algorithm running on each device is defined in Algorithm 7. Each learning agent is able to monitor and gather statistics required to perform the prediction, as mentioned in Section 7.2. Once a new task arrives at the node, it asks for help from the experts, as in FPL. The experts that our FPL algorithm can employ are the two regressor algorithms. This method, however, returns only one value, p, i.e., the expert's prediction for the next time step, and this predicted value is then used to determine whether offloading the incoming task as follows. If this value exceeds a determined threshold specific for the agent i, T_i, this implies a long local task execution, and the task is consequently offloaded. Otherwise, it is kept local and enqueued for future execution.

Algorithm 7 Overall algorithm

1: Initialize threshold T_i for all nodes
2: **for** all $i \in \mathcal{N}$ **do**
3:     **Wait** new task m
4:         Monitor the queue and node state
5:         $p \leftarrow$ `fpl(m, s, t)`
6:         **if** $p > T_i$ **then**
7:             Offload task m to the edge cloud
8:         **else**
9:             Enqueue task m locally
10:         Store states for the future
11:     **End Wait**

From the described algorithm, it appears as the value of the threshold T_i is a crucial parameter. Clearly, its setting depends on the environment and the nature of tasks, but, in general, it should be defined in order to balance the two actions, i.e., offloading and local execution. Offloading means diminishing the waiting time

but increasing the transmission time. Keeping the task locally implies facing the waiting time but avoiding wireless transmission. Thus, offloading should be selected only when the expected waiting time is considered “too high”. The threshold is a numerical definition of the “too high” concept.

In light of this self-learning procedure, not only can our agents progressively enhance the single predictors, but they can learn the more advisable algorithm to follow given the considered environment. Ideally, the expected behavior of this framework is that the VARMA is selected in the first place, given its ability to require a short amount of data (see Section 7.4). Then, when more metrics become available, the RFR can outperform the statistical model and, consequently, becoming the preferred choice of FPL.

It is known, indeed, that Random Forests produce better results on large datasets and are able to work with missing data by using estimations of them [20]. However, they pose a major challenge as they cannot extrapolate outside unseen data. On the other hand, VARMA has the ability to work well with unseen data, interpolating the given data to obtain the prediction. In conclusion – and as confirmed by our results – we can enumerate the differences as follows: classical models are simpler and more interpretable, while ML methods are more complex but more flexible. The choice of VARMA to represent classical models and RFR for ML is then motivated by the accuracy obtained in our experimental campaign (see Section 7.4).

7.4 Results

In this section, we report the results of experiments performed to assess the effectiveness of the proposed approach. First, we analyze the accuracy of the proposed prediction methods. Then, we consider the performance of our approach comparing it to state-of-the-art solutions.

7.4.1 Experimental Setup

To evaluate the performance of the proposed task offloading strategy, we developed a Python event-driven simulator, where a networked fleet of drones has to complete incoming tasks. The edge cloud is replicated by means of a further process emulating

the execution of offloaded tasks. To adopt realistic parameters for our experimental campaign, we base the choice of their default values on recent studies addressing the considered scenario, e.g., [174, 211]. In particular, new tasks are generated according to a Poisson process with an arrival rate of 0.2 Hz if not otherwise specified. In terms of computing resources, we assume the CPU capability of each server in the edge cloud and each UAV to be $f_e = 20$ GHz and $f_{i,m} = 1$ GHz, respectively. The computing workload is set as default to $C_{i,m} = 1 \times 10^9$. The channel bandwidth is set to be $W^{down} = W^{up} = 5$ Mbps, the transmitted data $D^{out}_{i,m} = 7$ MB, while $D^{in}_{i,m} = 1$ MB. The background noise power is set equal for the two technologies, as $\sigma^{down} = \sigma^{up} = 50$ dBm. For the channel gain we have $G^{down}_{i,m} = G^{up}_{i,m} = d^{\nu}_{i,s}$, where $d_{i,s}$ is the distance between mobile agent i and access point s, and $\nu = 4$ denotes the path loss factor. By default, the distance $d_{i,s}$ is set to 10m. Finally, we simply set the default value of the weights defined in Algorithm 6 as $\eta = 1$.

The results reported are obtained after 35 trials. The resulting graph's bars refer to a confidence interval of 90%. We summarize in Table 7.2 the configuration parameters utilized during the following evaluation, where the default values are reported in bold.

Table 7.2 Parameters setting.

Parameter	Values
Number of nodes	2, 3, **5**, 7, 10, 20, 50
Task arrival rate (Hz)	0.1, **0.2**, 0.3, 0.9
Nodes' Average Distance [m]	1, 2, 3, 5, **10**, 20, 30, 40
Computing workload, 10^9	0.5, **1**, 10, 25, 50, 100
Channel bandwidth (Mbps)	**5**
Noise power (dBm)	**50**
Number of trials	**35**
Confidence interval [%]	**90**

7.4.2 Evaluation Metrics

Throughout this section, we make use of metrics and quantities defined in Section 7.1, such as the task completion time. Besides them, in order to study the efficacy of predictors, we use the Mean Absolute Percentage Error (MAPE), which is a simple

regression error metric. For every data point, the residual is computed by taking only its absolute value so that negative and positive residuals do not cancel out. The error is then converted into a percentage, providing a clear interpretation that makes the results easily understandable. The formal equation of MAPE is given by:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n} 100 \times \left|\frac{x_t - y_t}{x_t}\right|, \tag{7.17}$$

where x_t and y_t are the real and the predicted observations, respectively. One key advantage of MAPE is its robustness to the effects of outliers thanks to the use of the absolute value. In summary, such a value describes how far the model's predictions are off from their corresponding outputs on average.

Similarly, we compute the Mean Absolute Deviation (MAD) for the predicted values as follows:

$$MAD = \frac{1}{n}\sum_{t=1}^{n} \left|y_t - \overline{X}\right|, \tag{7.18}$$

where $\overline{X}$ denotes the mean of the observed values. The MAD value, as explained in [212], is another key metric during the evaluation of regressors.

Moreover, even though it is not an explicit objective of the process, we also consider the energy consumed by agent i during the execution of task m. Since the node can either compute the task locally or offload its computation to the edge cloud, we define two types of energy consumption, $E^l_{i,m}$ and $E^e_{i,m}$, for the local execution and the edge execution, respectively. In the case of local computation, we use the widely adopted model of the energy consumption per computing cycle as $\mathscr{E} = kCf^2$ [213, 214], where k is the energy coefficient depending on the chip architecture, $f_{i,m}$ is the CPU frequency, and $C_{i,m}$ specifies the workload, i.e., the amount of computation to accomplish the task in terms of numbers of cycles. According to some realistic measurements available in [196], we set the energy coefficient k as 5×10^{-11}. Moreover, in the other event of task offloading, $E^{up} = \frac{p_{i,m}T^{tra}_{i,m}}{\xi_i}, \forall i, m$, where ξ_i is the power amplifier efficiency of node i. Without loss of generality, we assume that $\xi_i = 1 \; \forall i$. We also assume the energy consumption in the edge cloud is negligible since the cloud typically has enough energy to execute the offloaded tasks. Then, similar to the energy spent in transmission, we define the energy consumed during the reception phase, $E^{down} = \frac{p_{i,m}T^{rec}_{i,m}}{\xi_i}, \forall i, m$. Thus, the energy spent in offloading the task is the sum of these two communications, $E^e_{i,m} = E^{up} + E^{down}$.

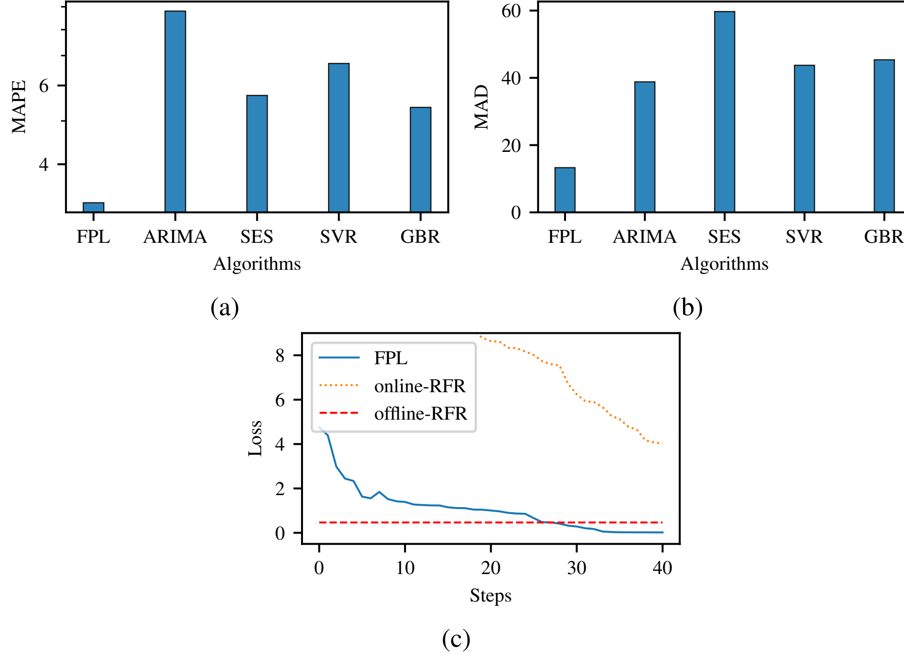


Fig. 7.2 (a) MAPE error and (b) mean absolute deviation (MAD) for different algorithms. (c) Convergence time comparison, i.e., loss evolution, for FPL and RFR methods.

In conclusion, for simplicity, we refer to the energy consumption as E, and is computed as follows:

$$\begin{aligned} E &= (1 - o_{i,m})E^{l}_{i,m} + o_{i,m}E^{e}_{i,m}, \\ &= (1 - o_{i,m})\left(k(f^{l}_{i,m})^{2}C_{i,m}\right) + o_{i,m}\left(p_{i,m}T^{tra}_{i,m} + p_{i,m}T^{rec}_{i,m}\right). \end{aligned} \tag{7.19}$$

7.4.3 Predictor Accuracy

For this first part focusing on the accuracy of the predictors, we first offline train the considered models on a relatively small dataset consisting of 5036 samples. In particular, we apply a walk-forward validation. In such a technique, the dataset is split into train and test sets by selecting a cut point, and we select a point to split the dataset as 80% training set and 20% test set. Then, even over the test set these models are fitted for every new observation, and the training phase continues online to improve the accuracy and to fit the specific circumstances on the agent.

Fig. 7.2a shows MAPE for different predictors. Specifically, we compare against two other time series forecasting methods, autoregressive-moving average (ARIMA)

and Simple Exponential Smoothing (SES), and two ML-based regressors, Support Vector Regression (SVR) and Gradient Boosting Regression (GBR). From the graph, we can observe how results validate our approach. In particular, by leveraging alternate techniques, our FPL model provides the lowest error in predicting. Notably, compared to the second-best regressor, i.e., SES algorithm, our method can halve the error.

We then compare the MAD error among the same set of predictors, and we report the results in Fig. 7.2b. We can easily conclude that not only FPL can provide a smaller error, but the variance is reduced. This result is particularly important since it assures that our approach leads to fewer outliers in the prediction task. In fact, a method with high MAD suggests that when it is wrong, the error could be too high, leading to an inappropriate conclusion. On the other hand, our FPL is always close to the real value, so even though the value is not exact, the finding is likely more accurate.

We then study the time needed by our FPL-based algorithm to converge. To this end, we compared the convergence time of FPL to RFR, as it is the most accurate at regime. We consider two different versions for the latter: an already trained version (offline-RFR) and an RFR during its learning phase (online-RFR). Fig. 7.2c displays the loss (actual value - predicted value) of these alternatives, where the offline-RFR is constant over time since the model parameters were already fixed during training. We can notice how our approach outperforms the offline-RFR after approximately 30 steps, while the online-RFR in the first 20 steps has a too excessive loss to report in the figure. Such an online-RFR method achieves a reasonable still high loss at 30 steps, the number of steps required by our FPL to stably converge. This observation validates our hypothesis of using an FPL-based algorithm to online adapt the prediction and mix online and offline parameter settings to speed up the learning.

7.4.4 Solution Performance Analysis

To study the behavior of our solution at varying environmental conditions, we consider diverse indicators for increasing fleet sizes.

In Fig. 7.3a we examine the average queue length of the agents when increasing the task arrival rate. We can observe how, when tasks are introduced in the system

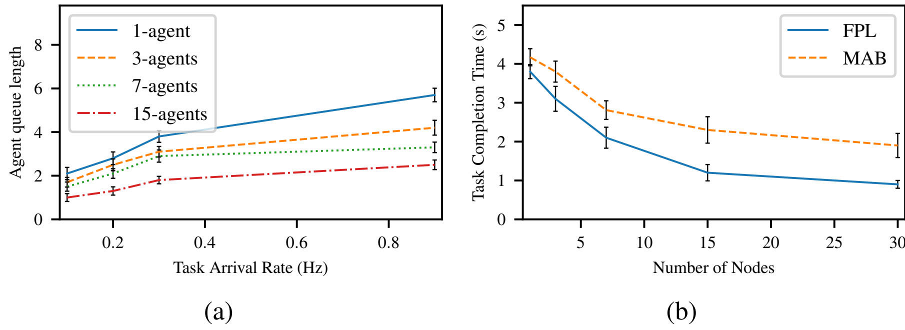


Fig. 7.3 (a) Queue agents length at varying the task arrival rate. Both experiments consider an increasing fleet size. (b) Task completion time of our FPL-based approach compared to a more complex solution as MAB. Our FPL outperforms this alternative.

at a higher rate, the growth in the queue size is logarithmic. This result suggests that our approach can efficiently handle the presence of many tasks in the system. Confirmation of this behavior is presented in Section 7.4.5, when our solution is compared against other methods.

7.4.5 Comparison with State-of-the-art

To study the effectiveness of our solution, we compare it against three similar solutions: the DROO framework [172], which implements a deep neural network that learns the binary offloading decisions; a solution based on the multi-agent reinforcement learning framework [23], that is able to select the best radio access technology for the offloading process; a hotbooting Q-learning scheme for computation offloading [188], herein referred to as hotbooting DQN, as it uses a fast deep Q-network (DQN) model to further improve the offloading performance.

We first evaluate the effects of task offloading over the queue of agents, reporting in Fig. 7.4a the CDF of queue length. The queue length is considered a key metric in this scenario, as it clearly impacts the time to complete tasks, but also the computation rate and the energy spent by the node. It can be easily observed that, with our solution, we can shorten the amount of tasks waiting in the agent's queue, while other RL-based methods are more prone to overload the agent.

Additionally, we investigate the energy consumption that the solutions lead to, reporting the results in Fig. 7.4b. While other benchmark algorithms consider the minimization of energy consumption in the problem formulation, our model is

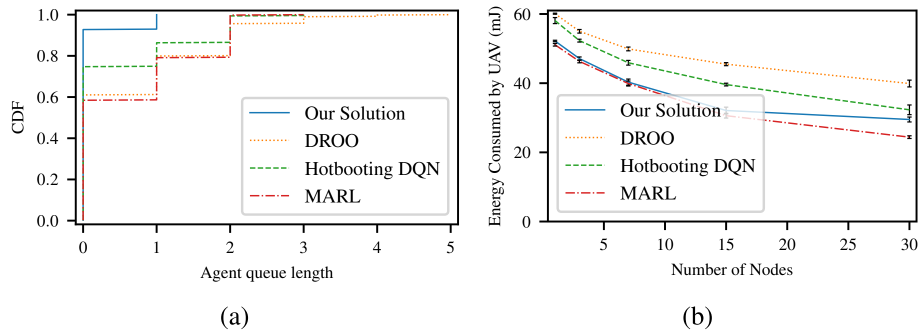


Fig. 7.4 CDF of queue agents length, and (c) energy consumption for various offloading solutions. Despite not as an objective of our algorithm, our solution can also limit the energy consumed by UAVs.

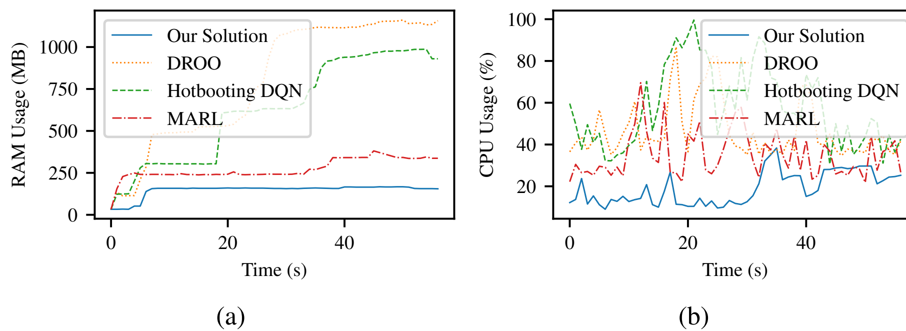


Fig. 7.5 Memory resources and (d) CPU consumed during the execution of our considered algorithms.

unaware of this aspect. Nevertheless, our solution is able to achieve comparable results with MARL that has been designed for energy efficiency purposes. Moreover, we can reduce consumption with respect to DROO and DQN. Thus we can conclude that, although our approach is blindfolded concerning power saving, it can lead to an energy-efficient method. These results confirm our hypothesis that modeling the device as a queue of tasks is a simplistic yet effective way of exploiting the edge while considering the application performance.

Finally, we consider the amount of RAM and CPU required to train and execute these algorithms. As can be seen in Fig. 7.5a, our implementation can drastically reduce the amount of memory consumed, leading to a significant improvement. While the deep learning approaches are hungry for RAM, the MARL model can optimize consumption. However, our regressors can further shorten the demand for memory. Considering then the CPU consumption in Fig. 7.5b, similar conclusions hold. Simulating the execution of the learning processes over an Intel(R) Core(TM)

i7-7500U CPU @ 2.70GHz, it is observable a reduction of CPU usage. In conclusion, we can consider our solution more lightweight than alternatives. This result is extremely important, especially in the UAV context, where a reduced memory footprint, along with less computation, is fundamental.

7.5 Conclusion

This chapter presents a learning-based solution to solve the dilemma of whether a task should be offloaded to the close edge cloud or not. Our solution lets the devices autonomously learn the offloading decisions on the basis of the current state. Such a decision exploits two classes of predictors, i.e., time series and ML regression, to predict future conditions. By doing so, the node can determine online the accuracy of these methods. Based on this value, then, the agent determines where incoming tasks should be executed. The results validate our model, evidencing how our implementation outperforms state-of-the-art solutions. In particular, despite the simplicity of our learning algorithm, its accuracy is comparable to other RL-based processes.

Chapter 8

Conclusions

Edge-based applications have evolved over the last decade because of a considerable demand from many recent applications to satisfy their strict requirements, *e.g.*, very low latency and high throughput. Among them, we can cite the Tactile Internet, which can offer haptic engagement with visual feedback, providing the illusion of remote touch, or telemedicine, where medical devices, or simply medical information, are accessed remotely during an interactive session. This posed several challenges to the network infrastructure that, to meet the stringent requirements, had to change its nature as well. To this end, there is now increasing interest in equipping networks with autonomous run-time decision-making capability via the incorporation of artificial intelligence (AI) and machine learning (ML). Given the fact that it is almost impossible for human operators to render network management in real-time, it is likely that future networks will apply AI/ML to *autonomously* identify and locate congestion or malfunctions in the network, and opportunely react. To accurately configure and manage itself, the network needs to detect the malfunction, collect and analyze measurements in a stream way. Once metrics are collected, the network reacts to address the sub-optimal behavior via network programmability.

In this dissertation, we explored novel methodologies aimed to solve some typical networking management problems with the inclusion of reactive mechanisms. In particular, we demonstrated how ML-based algorithms can be effectively used in optimizing network operations, ranging from the offloading of tasks to the close edge cloud to the management of virtualized and softwarized networks.

Our attempt to solve this issue brought us to the design and implementation of a novel load-aware routing strategy, RoPE, to enhance routing in SDN networks by means of multiple ML strategies. Our routing solution predicts the future load on links of a path and then chooses the best one according to such information. This allows to avoid congested paths and reduces delay in the transmissions. After having analyzed the regression algorithms and evaluated the advantages and disadvantages of the class of methods, we defined the RoPE logic to choose the best algorithms as the considered scenario changes.

Similarly, but to optimize (virtual) network resource allocation, we proposed Mystique, a system that auto-scales the underlying network topology to accommodate the traffic demand and reacts to possible failures via Reinforcement Learning (RL). The network controller dynamically activates or deactivates links and nodes in an "as needed" fashion with the aim of minimizing the energy consumption and improving QoE and fairness among users. At the same time, the system can promptly react to network failures as these happen.

To mitigate congestion at the end-hosts rather than on the network infrastructure, then, we proposed Owl, a new RL-based TCP congestion-control algorithm. Designed to learn from end-to-end and in-network signals, Owl is effective under various network conditions, and it can speed up transmissions and reduce delays and loss rates better than most existing protocols in the vast majority of the tested scenarios, especially in cellular networks. We also analyzed the stability condition of Owl and evaluated its fairness demonstrating that it is less aggressive than other performant solutions when it competes with other protocols and with itself across other sources.

Lastly, we specifically addressed the task offloading problem and proposed a distributed offloading decision strategy that, using Multi-Agent Reinforcement Learning (MARL), jointly improves the energy efficiency and task completion time of edge computing-enabled UAVs swarms. The overall state of the system is appropriately shared between the nodes and used when each agent has to decide where to perform an assigned task: locally or in the edge cloud by means of an offloading procedure. Each node, in case of task offloading, can further decide the transmission technology to use, Wi-Fi or LTE, according to the current utilization.

For a more lightweight solution, however, we also presented a self-learning-based methodology that reduces the overall resource consumption and is more suitable

for UAV execution. Our solution lets the devices autonomously learn the offloading decisions on the basis of the current state. Such a decision exploits two classes of predictors, i.e., time series and ML regression, to predict future conditions. Using this value, then, the agent determines whether a task should be offloaded to the close edge cloud or not.

As we deepened only a subset of the existing problems, we hope that future research will cover the remaining challenges that, if solved, could lead to the definition of a fully autonomous and functional network that can enable a better machine-human hybrid architecture. Among the future challenges, we can cite, for example, the need to cope with a few or partial data, which can hinder an accurate learning process. Taking network management decisions when the telemetry is limited may move current solutions towards production. Similarly, explaining the decisions of the ML model can open up new opportunities to better understand the logic behind the selected output, with the possibility to realize which input (and how) has most impacted the final decision. Once gained this insight, the network telemetry process can be optimized as well, improving both the process of network metrics collection and the consequent decision process. By assuring that the partial information includes the most impacting information, the ML model can be accurate enough in taking the proper network operation.

References

[1] Yong Li and Min Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[3] Data plane development kit. https://www.dpdk.org/.

[4] Alessio Sacco, Flavio Esposito, and Guido Marchetto. On control and data plane programmability for data-driven networking. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2021.

[5] Gerhard P Fettweis. The tactile internet: Applications and challenges. *IEEE Vehicular Technology Magazine*, 9(1):64–70, 2014.

[6] Alessio Sacco, Flavio Esposito, Guido Marchetto, Grant Kolar, and Kate Schwetye. On edge computing for remote pathology consultations and computations. *IEEE Journal of Biomedical and Health Informatics*, 24(9):2523–2534, 2020.

[7] Ronald S Weinstein et al. Telepathology overview: from concept to implementation. *Human pathology*, 32(12):1283–1299, 2001.

[8] Huy Trinh, Dmitrii Chemodanov, Shizeng Yao, Qing Lei, Bo Zhang, Fan Gao, Prasad Calyam, and Kannappan Palaniappan. Energy-aware mobile edge computing for low-latency visual data processing. In *5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 128–133. IEEE, 2017.

[9] Sarita Chung, C Mario Christoudias, Trevor Darrell, Sonja I Ziniel, and Leslie A Kalish. A novel image-based tool to reunite children with their families after disasters. *Academic emergency medicine*, 19(11):1227–1234, 2012.

[10] Kyle Coleman, Flavio Esposito, and Rachel Charney. Speeding up children reunification in disaster scenarios via serverless computing. In *Proceedings of the 2nd International Workshop on Serverless Computing (WoSC '17)*, pages 5–5, 2017.

[11] Nick Feamster, Arpit Gupta, Jennifer Rexford, and Walter Willinger. Nsf workshop on measurements for self-driving networks. In *Workshop on Measurements for Self-Driving Networks was held at Princeton University on April*, volume 4, page 5, 2019.

[12] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583*, 2017.

[13] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. Adaptable and data-driven softwarized networks: Review, opportunities, and challenges. *Proceedings of the IEEE*, 107(4):711–731, 2019.

[14] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):16, 2018.

[15] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications surveys & tutorials*, 21(3):2224–2287, 2019.

[16] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*, pages 44–58, 2018.

[17] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 495–511, 2020.

[18] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, pages 557–570, 2020.

[19] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, pages 107–125, 2020.

[20] Alessio Sacco, Flavio Esposito, and Guido Marchetto. Rope: An architecture for adaptive data-driven routing prediction at the edge. *IEEE Transactions on Network and Service Management*, 17(2):986–999, 2020.

[21] Alessio Sacco, Flavio Matteo, Flocco andEsposito, and Guido Marchetto. Supporting sustainable virtual network mutations with mystique. *IEEE Transactions on Network and Service Management*, 18(3):2714–2727, 2021.

[22] Alessio Sacco, Flocco Matteo, Flavio Esposito, and Guido Marchetto. Owl: Congestion control with partially invisible networks via reinforcement learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021.

[23] Alessio Sacco, Flavio Esposito, Guido Marchetto, and Paolo Montuschi. Sustainable task offloading in uav networks via multi-agent reinforcement learning. *IEEE Transactions on Vehicular Technology*, 70(5):5003–5015, 2021.

[24] Alessio Sacco, Flavio Esposito, Guido Marchetto, and Paolo Montuschi. A self-learning strategy for task offloading in uav networks. *IEEE Transactions on Vehicular Technology*, 2022.

[25] Changchun Long, Yang Cao, Tao Jiang, and Qian Zhang. Edge computing framework for cooperative video processing in multimedia iot systems. *IEEE Transactions on Multimedia*, 20(5):1126–1139, 2017.

[26] Petar Popovski. Ultra-reliable communication in 5g wireless systems. In *1st International Conference on 5G for Ubiquitous Connectivity*, pages 146–151. IEEE, 2014.

[27] Shaimaa Al-Janabi, André Huisman, and Paul J Van Diest. Digital pathology: current status and future perspectives. *Histopathology*, 61(1):1–9, 2012.

[28] Peter W Hamilton, Yinhai Wang, and Stephen J McCullough. Virtual microscopy and digital pathology in training and education. *Apmis*, 120(4):305–315, 2012.

[29] Bernard Têtu et al. Whole-slide imaging-based telepathology in geographically dispersed healthcare networks. the eastern québec telepathology project. *Diagnostic Histopathology*, 20(12):462–469, 2014.

[30] Ronald S Weinstein, Anna R Graham, et al. Overview of telepathology, virtual microscopy, and whole slide imaging: prospects for the future. *Human pathology*, 40(8):1057–1069, 2009.

[31] Alessio Sacco, Flavio Esposito, Princewill Okorie, and Guido Marchetto. Livemicro: An edge computing system for collaborative telepathology. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge '19)*. USENIX Association, 2019.

[32] Justin Franz, Tanmayi Nagasuri, Andrew Wartman, Agnese V Ventrella, and Flavio Esposito. Reunifying families after a disaster via serverless computing and raspberry pis. In *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 131–132. IEEE, 2018.

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

[34] Jianyu Wang, Jianli Pan, and Flavio Esposito. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things (SmartIoT '17)*, pages 1–6. ACM, 2017.

[35] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. An architecture for adaptive task planning in support of iot-based machine learning applications for disaster scenarios. *Computer Communications*, 160:769–778, 2020.

[36] Josiah Burchard, Dmitrii Chemodanov, John Gillis, and Prasad Calyam. Wireless mesh networking protocol for sustained throughput in edge computing. In *International Conference on Computing, Networking and Communications (ICNC)*, pages 958–962. IEEE, 2017.

[37] Charalampos Kalalas, Linus Thrybom, and Jesus Alonso-Zarate. Cellular communications for smart grid neighborhood area networks: A survey. *IEEE access*, 4:1469–1493, 2016.

[38] Meryem Simsek, Adnan Aijaz, Mischa Dohler, Joachim Sachs, and Gerhard Fettweis. 5g-enabled tactile internet. *IEEE Journal on Selected Areas in Communications*, 34(3):460–473, 2016.

[39] Abdelhamied A Ateya, Anastasia Vybornova, Ruslan Kirichek, and Andrey Koucheryavy. Multilevel cloud based tactile internet system. In *19th International Conference on Advanced Communication Technology (ICACT)*, pages 105–110. IEEE, 2017.

[40] Abdelhamied Ateya, Ammar Muthanna, Irina Gudkova, Abdelrahman Abuarqoub, Anastasia Vybornova, and Andrey Koucheryavy. Development of intelligent core network for tactile internet and future smart systems. *Journal of Sensor and Actuator Networks*, 7(1):1, 2018.

[41] Cisco ios netflow. http://www.cisco.com/web/go/netflow.

[42] Open vswitch: An open virtual switch. http://www.openvswitch.org/.

[43] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 272–285, New York, NY, USA, 2016. ACM.

[44] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 325–338, New York, NY, USA, 2015. ACM.

[45] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, Renton, WA, April 2018. USENIX Association.

[46] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 197–210, New York, NY, USA, 2017. ACM.

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

[48] Jianyu Wang, Jianli Pan, Flavio Esposito, Prasad Calyam, Zhicheng Yang, and Prasant Mohapatra. Edge cloud offloading algorithms: Issues, methods, and perspectives. *ACM Comput. Surv.*, 52(1):2:1–2:23, February 2019.

[49] Brandon Schlinker et al. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 418–431, New York, NY, USA, 2017. ACM.

[50] Kok-Kiong Yap et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 432–445, New York, NY, USA, 2017. ACM.

[51] Paulo Cortez, Miguel Rio, Miguel Rocha, and Pedro Sousa. Internet traffic forecasting using neural networks. In *The 2006 IEEE international joint conference on neural network proceedings*, pages 2635–2642. IEEE, 2006.

[52] Edmund S Yu and CY Roger Chen. Traffic prediction using neural networks. In *Proceedings of GLOBECOM'93. IEEE Global Telecommunications Conference*, pages 991–995. IEEE, 1993.

[53] Y. Li et al. Inter-data-center network traffic prediction with elephant flows. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 206–213, April 2016.

[54] Cooperative association for internet data analysis. http://www.caida.org/tools.

[55] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Transactions on Networking (TON)*, 11(4):537–549, 2003.

[56] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*, pages 39–44, 2003.

[57] Joel Sommers, Paul Barford, and Walter Willinger. A proposed framework for calibration of available bandwidth estimation tools. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 709–718. IEEE, 2006.

[58] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the seventh conference on emerging networking experiments and technologies (CoNEXT)*, pages 1–12. ACM, 2011.

[59] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–6. ACM, 2015.

[60] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *Proceedings of the 16th ACM workshop on hot topics in networks (HotNets '17)*, pages 185–191, 2017.

[61] Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE transactions on software engineering*, 21(3):181–199, 1995.

[62] Robert Beverly, Karen Sollins, and Arthur Berger. Svm learning of ip address structure for latency prediction. In *Proceedings of the 2006 SIGCOMM workshop on Mining network data (MineNet)*, pages 299–304. ACM, 2006.

[63] Paola Bermolen and Dario Rossi. Support vector regression for link load prediction. *Computer Networks*, 53(2):191–201, 2009.

[64] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the predictability of large transfer tcp throughput. *ACM SIGCOMM Computer Communication Review*, 35(4):145–156, 2005.

[65] Martin Swany and Rich Wolski. Multivariate resource performance forecasting in the network weather service. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*, pages 11–11. IEEE, 2002.

[66] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. A machine learning approach to tcp throughput prediction. *IEEE/ACM Transactions on Networking*, 18(4):1026–1039, 2010.

[67] Pascal Poupart, Zhitang Chen, Priyank Jaini, Fred Fung, Hengky Susanto, Yanhui Geng, Li Chen, Kai Chen, and Hao Jin. Online flow size prediction for improved network routing. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2016.

[68] Yong Wang, Margaret Martonosi, and Li-Shiuan Peh. Predicting link quality using supervised learning in wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 11(3):71–83, 2007.

[69] David Wolpert, Kagan Tumer, and Jeremy Frank. Using collective intelligence to route internet traffic. In *Advances in neural information processing systems (NIPS)*, volume 11, 1998.

[70] Shailesh Kumar and Risto Miikkulainen. Dual reinforcement q-routing: An on-line adaptive routing algorithm. In *Proceedings of the artificial neural networks in engineering Conference*, pages 231–238. Citeseer, 1997.

[71] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *Proceedings IEEE INFOCOM 2000. IEEE conference on computer communications*, volume 2, pages 519–528. IEEE, 2000.

[72] Reuven Cohen, Yuval Dagan, and Gabi Nakibly. Proactive rerouting in network overlays. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2018.

[73] Marco Chiesa, Gábor Rétvári, and Michael Schapira. Lying your way to better traffic engineering. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 391–398, 2016.

[74] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. *ACM SIGCOMM Computer Communication Review*, 43(4):339–350, 2013.

[75] Mohammed Alreshoodi and John Woods. Survey on QoE\QoS correlation models for multimedia services. *arXiv preprint arXiv:1306.0221*, 2013.

[76] Qi Zhang, Jianhui Liu, and Guodong Zhao. Towards 5g enabled tactile robotic telesurgery. *arXiv preprint arXiv:1803.03586*, 2018.

[77] Yeon-sup Lim, Yung-Chih Chen, Erich M Nahum, Don Towsley, and Richard J Gibbens. How green is multipath tcp for mobile devices? In *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*, pages 3–8. ACM, 2014.

[78] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[79] Paul Geladi and Bruce R Kowalski. Partial least-squares regression: a tutorial. *Analytica chimica acta*, 185:1–17, 1986.

[80] Min Xu, Pakorn Watanachaturaporn, Pramod K Varshney, and Manoj K Arora. Decision tree regression for soft classification of remote sensing data. *Remote Sensing of Environment*, 97(3):322–336, 2005.

[81] Unina Traffic and data traces. http://www.grid.unina.it/Traces/index.php.

[82] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.

[83] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[84] Sebastian Troia, Alberto Rodriguez, Rodolfo Alvizu, and Guido Maier. Senatus: an experimental sdn/nfv orchestrator. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–5. IEEE, 2018.

[85] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01)*, pages 97–106. ACM, 2001.

[86] Geni, Exploring Networks of the Future. https://www.geni.net/.

[87] Alessio Sacco, Flavio Esposito, and Guido Marchetto. A federated learning approach to routing in challenged sdn-enabled edge networks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 150–154. IEEE, 2020.

[88] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Nsdi*, volume 10, pages 249–264, 2010.

[89] Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore, and Biswanath Mukherjee. Auto-scaling vnfs using machine learning to improve qos and reduce cost. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[90] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, pages 50–56, 2016.

[91] Pengcheng Tang, Fei Li, Wei Zhou, Weihua Hu, and Li Yang. Efficient auto-scaling approach in the telco cloud using self-learning algorithm. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.

[92] Ender Ayanoglu, I Chih-Lin, Richard D Gitlin, and James E Mazo. Diversity coding for transparent self-healing and fault-tolerant communication networks. *IEEE Transactions on communications*, 41(11):1677–1686, 1993.

[93] Patrick Le Callet, Sebastian Möller, Andrew Perkis, et al. Qualinet white paper on definitions of quality of experience. *European network on quality of experience in multimedia systems and services (COST Action IC 1003)*, 3(2012), 2012.

[94] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2017.

[95] Adel Nadjaran Toosi, Jungmin Son, Qinghua Chi, and Rajkumar Buyya. Elasticsfc: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds. *Journal of Systems and Software*, 152:108–119, 2019.

[96] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[97] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Transactions on Internet Technology (TOIT)*, 7(1):7–es, 2007.

[98] Tuan Phung-Duc, Yi Ren, Jyh-Cheng Chen, and Zheng-Wei Yu. Design and analysis of deadline and budget constrained autoscaling (dbca) algorithm for 5g mobile networks. In *2016 IEEE international conference on cloud computing technology and science (CloudCom)*, pages 94–101. IEEE, 2016.

[99] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A multi-model framework to implement self-managing control systems for qos management. In *Proceedings of the 6th international symposium on software engineering for adaptive and self-managing systems*, pages 218–227, 2011.

[100] Peter Bodík, Rean Griffith, Charles A Sutton, Armando Fox, Michael I Jordan, and David A Patterson. Statistical machine learning makes automatic control practical for internet datacenters. *HotCloud*, 9:12–12, 2009.

[101] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212. IEEE, 2012.

[102] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26, 2009.

[103] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.

[104] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. Ieee, 2010.

[105] Alessio Sacco, Flavio Esposito, and Guido Marchetto. Resource inference for task migration in challenged edge networks with ritmo. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–7. IEEE, 2020.

[106] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011.

[107] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *International Workshop on Quality of Service*, pages 381–398. Springer, 2003.

[108] Sunirmal Khatua, Anirban Ghosh, and Nandini Mukherjee. Optimizing the utilization of virtual resources in cloud environment. In *2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pages 82–87. IEEE, 2010.

[109] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *2006 IEEE International Conference on Autonomic Computing*, pages 65–73. IEEE, 2006.

[110] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 67–74, 2011.

[111] Osianoh Glenn Aliu, Ali Imran, Muhammad Ali Imran, and Barry Evans. A survey of self organisation in future cellular networks. *IEEE Communications Surveys & Tutorials*, 15(1):336–361, 2012.

[112] Ryan W Thomas, Daniel H Friend, Luiz A Dasilva, and Allen B Mackenzie. Cognitive networks: adaptation and learning to achieve end-to-end performance objectives. *IEEE Communications magazine*, 44(12):51–57, 2006.

[113] Albert Mestres, Alberto Rodriguez-Natal, Josep Carner, Pere Barlet-Ros, Eduard Alarcón, Marc Solé, Victor Muntés-Mulero, David Meyer, Sharon Barkai, Mike J Hibbett, et al. Knowledge-defined networking. *ACM SIGCOMM Computer Communication Review*, 47(3):2–10, 2017.

[114] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. Unleashing the potential of data-driven networking. In *International Conference on Communication Systems and Networks*, pages 110–126. Springer, 2017.

[115] Patrick Kalmbach, Johannes Zerwas, Peter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Empowering self-driving networks. In *Proceedings of the afternoon workshop on self-driving networks*, pages 8–14, 2018.

[116] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM workshop on hot topics in networks (HotNets '14)*, pages 1–7, 2014.

[117] Alessio Sacco, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. Work-in-progress: A formal approach to verify fault tolerance in industrial network systems. In *16th IEEE International Conference on Factory Communication Systems (WFCS)*, pages 1–4. IEEE, 2020.

[118] Openflow switch specification 1.3.1. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf.

[119] Stefano Petrangeli, Jeroen Famaey, Maxim Claeys, Steven Latré, and Filip De Turck. Qoe-driven rate adaptation heuristic for fair adaptive video streaming. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 12(2):1–24, 2015.

[120] Tobias Hoßfeld, Lea Skorin-Kapov, Poul E Heegaard, and Martin Varela. Definition of qoe fairness in shared systems. *IEEE Communications Letters*, 21(1):184–187, 2016.

[121] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21, 1984.

[122] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, 2007.

[123] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[124] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[125] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

[126] Ryu controller. https://ryu-sdn.org/.

[127] Rogério Leão Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6. IEEE, 2014.

[128] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[129] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[130] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. Computing tcp's retransmission timer. Technical report, RFc 2988, November, 2000.

[131] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. An in-depth study of lte: effect of network protocol and application behavior on performance. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 363–374. ACM, 2013.

[132] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling bufferbloat in 3g/4g networks. In *Proceedings of the 2012 Internet Measurement Conference*, pages 329–342. ACM, 2012.

[133] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

[134] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, 2013.

[135] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 509–522. ACM, 2015.

[136] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM '10)*, pages 63–74. ACM New York, NY, USA, 2010.

[137] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[138] Wei Li, Fan Zhou, Waleed Meleis, and Kaushik Chowdhury. Learning-based and data-driven tcp design for memory-constrained iot. In *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 199–205. IEEE, 2016.

[139] B Venkata Ramana and C Siva Ram Murthy. Learning-tcp: a novel learning automata based congestion window updating mechanism for ad hoc wireless networks. In *International Conference on High-Performance Computing*, pages 454–464. Springer, 2005.

[140] Venkataramana Badarla and C Siva Ram Murthy. Learning-tcp: A stochastic approach for efficient update in tcp congestion window in ad hoc wireless networks. *Journal of Parallel and Distributed Computing*, 71(6):863–878, 2011.

[141] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, Oakland, CA, 2015.

[142] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, 2018.

[143] Yiming Kong, Hui Zang, and Xiaoli Ma. Improving tcp congestion control with machine intelligence. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 60–66, 2018.

[144] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2018.

[145] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, 2020.

[146] Sally Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[147] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[148] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning (ICML)*, pages 3050–3059, 2019.

[149] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[150] Janey C Hoe. Improving the start-up behavior of a congestion control scheme for tcp. *ACM SIGCOMM Computer Communication Review*, 26(4):270–280, 1996.

[151] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[152] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):20–53, 2016.

[153] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound tcp approach for high-speed and long distance networks. In *IEEE INFOCOM. IEEE International Conference on Computer Communications*, pages 1–12. IEEE, 2006.

[154] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM. IEEE International Conference on Computer Communications*, volume 4, pages 2490–2501. IEEE, 2004.

[155] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

[156] Venkataramana Badarla and C Siva Ram Murthy. A novel learning based solution for efficient data transport in heterogeneous wireless networks. *Wireless Networks*, 16(6):1777–1798, 2010.

[157] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, page 632–647, 2020.

[158] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[159] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[160] Sea Shalunov, Greg Hazel, Janardhan Iyengar, Mirja Kuehlewind, et al. Low extra delay background transport (ledbat). In *RFC 6817*, 2012.

[161] Ingemar Johansson. Self-clocked rate adaptation for conversational video in lte. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 51–56, 2014.

[162] Adam Bergkvist, Daniel C Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[163] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. *ACM SIGCOMM Computer Communication Review*, 44(4):479–490, 2014.

[164] Yong Zeng, Rui Zhang, and Teng Joon Lim. Wireless communications with unmanned aerial vehicles: Opportunities and challenges. *IEEE Communications Magazine*, 54(5):36–42, 2016.

[165] Agnese V Ventrella, Flavio Esposito, Alessio Sacco, Matteo Flocco, Guido Marchetto, and Srikanth Gururajan. Apron: an architecture for adaptive task planning of internet of things in challenged edge networks. In *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, pages 1–6. IEEE, 2019.

[166] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, 24(5):2795–2808, 2015.

[167] Changsheng You, Kaibin Huang, Hyukjin Chae, and Byoung-Hoon Kim. Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications*, 16(3):1397–1411, 2016.

[168] Hongzhi Guo, Jiajia Liu, Jie Zhang, Wen Sun, and Nei Kato. Mobile-edge computation offloading for ultradense iot networks. *IEEE Internet of Things Journal*, 5(6):4977–4988, 2018.

[169] Jie Xu, Lixing Chen, and Pan Zhou. Joint service caching and task offloading for mobile edge computing in dense networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 207–215. IEEE, 2018.

[170] Tiago Koketsu Rodrigues, Jiajia Liu, and Nei Kato. Offloading decision for mobile multi-access edge computing in a multi-tiered 6g network. *IEEE Transactions on Emerging Topics in Computing*, 2021.

[171] Alessio Sacco, Flavio Esposito, and Guido Marchetto. Resource inference for sustainable and responsive task offloading in challenged edge networks. *IEEE Transactions on Green Communications and Networking*, 5(3):1114–1127, 2021.

[172] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang. Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks. *IEEE Transactions on Mobile Computing*, 19(11):2581–2593, 2019.

[173] Xiaoyu Qiu, Luobin Liu, Wuhui Chen, Zicong Hong, and Zibin Zheng. Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. *IEEE Transactions on Vehicular Technology*, 68(8):8050–8062, 2019.

[174] Yi Liu, Huimin Yu, Shengli Xie, and Yan Zhang. Deep reinforcement learning for offloading and resource allocation in vehicle edge computing and networks. *IEEE Transactions on Vehicular Technology*, 68(11):11158–11168, 2019.

[175] Zhen Zhang, Zicong Hong, Wuhui Chen, Zibin Zheng, and Xu Chen. Joint computation offloading and coin loaning for blockchain-empowered mobile-edge computing. *IEEE Internet of Things Journal*, 6(6):9934–9950, 2019.

[176] Baichuan Liu, Weikun Zhang, Wuhui Chen, Huawei Huang, and Song Guo. Online computation offloading and traffic routing for uav swarms in edge-cloud computing. *IEEE Transactions on Vehicular Technology*, 2020.

[177] Mohamed-Ayoub Messous, Hichem Sedjelmaci, Noureddin Houari, and Sidi-Mohammed Senouci. Computation offloading game for an uav network in mobile edge computing. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[178] Junhui Zhao, Qiuping Li, Yi Gong, and Ke Zhang. Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks. *IEEE Transactions on Vehicular Technology*, 68(8):7944–7956, 2019.

[179] Minyeong Gong and Sanghyun Ahn. Computation offloading-based task scheduling in the vehicular communication environment for computation-intensive vehicular tasks. In *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 534–537. IEEE, 2020.

[180] Estefania Coronado, Gabriel Cebrian-Marquez, and Roberto Riggio. Enabling computation offloading for autonomous and assisted driving in 5g networks. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[181] Sergio Barbarossa, Stefania Sardellitti, and Paolo Di Lorenzo. Communicating while computing: Distributed mobile cloud computing over 5g heterogeneous networks. *IEEE Signal Processing Magazine*, 31(6):45–55, 2014.

[182] Olga Munoz, Antonio Pascual-Iserte, and Josep Vidal. Optimization of radio and computational resources for energy efficiency in latency-constrained application offloading. *IEEE Transactions on Vehicular Technology*, 64(10):4738–4755, 2014.

[183] Nikos Kalatzis, Marios Avgeris, Dimitris Dechouniotis, Konstantinos Papadakis-Vlachopapadopoulos, Ioanna Roussaki, and Symeon Papavassiliou. Edge computing in iot ecosystems for uav-enabled early fire detection. In

2018 IEEE International Conference on Smart Computing (SMARTCOMP), pages 106–114. IEEE, 2018.

[184] Bowen Zhou, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. An online algorithm for task offloading in heterogeneous mobile clouds. *ACM Transactions on Internet Technology (TOIT)*, 18(2):1–25, 2018.

[185] Hongchang Ke, Jian Wang, Lingyue Deng, Yuming Ge, and Hui Wang. Deep reinforcement learning-based adaptive computation offloading for mec in heterogeneous vehicular networks. *IEEE Transactions on Vehicular Technology*, 69(7):7916–7929, 2020.

[186] Xiaoyu Zhu, Yueyi Luo, Anfeng Liu, Md Zakirul Alam Bhuiyan, and Shaobo Zhang. Multiagent deep reinforcement learning for vehicular computation offloading in iot. *IEEE Internet of Things Journal*, 8(12):9763–9773, 2020.

[187] Liang Huang, Xu Feng, Anqi Feng, Yupin Huang, and Li Ping Qian. Distributed deep learning-based offloading for mobile edge computing networks. *Mobile Networks and Applications*, pages 1–8, 2018.

[188] Minghui Min, Liang Xiao, Ye Chen, Peng Cheng, Di Wu, and Weihua Zhuang. Learning-based computation offloading for iot devices with energy harvesting. *IEEE Transactions on Vehicular Technology*, 68(2):1930–1941, 2019.

[189] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[190] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[191] Shalabh Bhatnagar, Mohammad Ghavamzadeh, Mark Lee, and Richard S Sutton. Incremental natural actor-critic algorithms. In *Advances in neural information processing systems*, pages 105–112, 2008.

[192] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

[193] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[194] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning (ICML)*, pages 1928–1937. PMLR, 2016.

[195] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[196] Antti P Miettinen and Jukka K Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, pages 1–7. USENIX Association, 2010.

[197] Dong Huang, Ping Wang, and Dusit Niyato. A dynamic offloading algorithm for mobile computing. *IEEE Transactions on Wireless Communications*, 11(6):1991–1995, 2012.

[198] Alexey Rudenko, Peter Reiher, Gerald J Popek, and Geoffrey H Kuenning. Saving portable computer battery power through remote process execution. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2(1):19–26, 1998.

[199] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5872–5881. PMLR, 2018.

[200] John Tsitsiklis, Dimitri Bertsekas, and Michael Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control*, 31(9):803–812, 1986.

[201] Stephen Boyd, Neal Parikh, and Eric Chu. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[202] Cell vs wifi. http://web.mit.edu/cell-vs-wifi/.

[203] Yaohua Sun, Mugen Peng, and Shiwen Mao. Deep reinforcement learning-based mode selection and resource management for green fog radio access networks. *IEEE Internet of Things Journal*, 6(2):1960–1971, 2018.

[204] EJ Hannan. The identification of vector mixed autoregressive-moving average system. *Biometrika*, 56(1):223–225, 1969.

[205] Nick Littlestone and Manfred K Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

[206] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th international conference on machine learning (ICML '03)*, pages 928–936, 2003.

[207] James Hannan. Approximation to bayes risk in repeated play. *Contributions to the Theory of Games*, 3:97–139, 1957.

[208] Adam Kalai and Santosh Vempala. Efficient algorithms for online decision problems. *Journal of Computer and System Sciences*, 71(3):291–307, 2005.

[209] Nicolo Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006.

[210] Alon Cohen and Tamir Hazan. Following the perturbed leader for online structured learning. In *International Conference on Machine Learning (ICML)*, pages 1034–1042. PMLR, 2015.

[211] Tuyen X Tran and Dario Pompili. Joint task offloading and resource allocation for multi-server mobile-edge computing networks. *IEEE Transactions on Vehicular Technology*, 68(1):856–868, 2018.

[212] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, 2003.

[213] Yonggang Wen, Weiwen Zhang, and Haiyun Luo. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *2012 Proceedings IEEE INFOCOM*, pages 2716–2720. IEEE, 2012.

[214] Xu Chen. Decentralized computation offloading game for mobile cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 26(4):974–983, 2014.

Appendix A

List of Publications

The following is the complete list of publications carried out during the Ph.D.

- Alessio Sacco, Matteo Flocco, Flavio Esposito and Guido Marchetto. "Owl: Congestion Control with Partially Invisible Networks via Reinforcement Learning" in Proc. 2021 IEEE International Conference on Computer Communications (IEEE INFOCOM), 10-13 May 2021, Online Event.
- Alessio Sacco, Flavio Esposito and Guido Marchetto. "On Control and Data Plane Programmability for Data-Driven Networking" in Proc. 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR), 7-10 June 2021, Online Event.
- Alessio Sacco, Flavio Esposito and Guido Marchetto. "Resource Inference for Sustainable and Responsive Task Offloading in Challenged Edge Networks" in IEEE Transactions on Green Communications and Networking (TGCN), vol. 5, no. 3, pp. 1114-1127, September 2021.
- Alessio Sacco, Flavio Esposito, Guido Marchetto and Paolo Montuschi. "Sustainable Task Offloading in UAV Networks via Multi-Agent Reinforcement Learning" in IEEE Transactions on Vehicular Technology (TVT), vol. 7, no. 5, pp. 5003-5015, April 2021.
- Alessio Sacco, Matteo Flocco, Flavio Esposito and Guido Marchetto. "Supporting Sustainable Virtual Network Mutations with Mystique" in IEEE Trans-

actions on Network and Service Management (TNSM), vol.18, no.3, pp. 2714-2727, February 2021.

- Alessio Sacco, Flavio Esposito and Guido Marchetto. “A Distributed Reinforcement Learning Approach for Energy and Congestion-Aware Edge Networks” in Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT), (pp.546-547). Poster. Online Event.
- Alessio Sacco, Flavio Esposito and Guido Marchetto. “Resource Inference for Task Migration in Challenged Edge Networks with RITMO” in Proc. 2020 IEEE 9th International Conference on Cloud Networking (CloudNet), (pp. 1-7). 9-11 November 2020. Online Event.
- Alessio Sacco, Flavio Esposito, Guido Marchetto, Grant Kolar, and Kate Schwetye. “On Edge Computing for Remote Pathology Consultations and Computations” in IEEE Journal of Biomedical and Health Informatics (J-BHI), 24 (9), 2523 – 2534, July 2020.
- Alessio Sacco, Matteo Flocco, Flavio Esposito and Guido Marchetto. “An Architecture for Adaptive Task Planning in Support of IoT-based Machine Learning Applications for Disaster Scenarios” in Computer Communication, 160, 769-778.
- Alessio Sacco, Flavio Esposito and Guido Marchetto. “A Federated Learning Approach to Routing in Challenged SDN-Enabled Edge Networks” in Proc. of IEEE Conference on Network Softwarization (NetSoft), (pp. 150-154). IEEE. Short paper. 29 June-3 July 2020, Online Event.
- Alessio Sacco, Guido Marchetto, Riccardo Sisto and Fulvio Valenza. “Work-in-progress: A Formal Approach to Verify Fault Tolerance in Industrial Network Systems” in Proc. of 16th IEEE International Conference on Factory Communication Systems (WFCS) (pp. 1-4). IEEE. WiP. 27-29 April 2020, Online Event.
- Alessio Sacco, Flavio Esposito and Guido Marchetto. “RoPE: An Architecture for Adaptive Data-Driven Routing Prediction at the Edge” in IEEE Transactions on Network and Service Management (TNSM), vol. 17, no. 2, pp. 986-999, June 2020.

- Agnese Ventrella, Flavio Esposito, Alessio Sacco, Matteo Flocco, Guido Marchetto and Srikanth Gururajan. “APRON: An Architecture for Adaptive Task Planning of Internet of Things in Challenged Edge Networks” in Proc. of the 8th IEEE International Conference on Cloud Networking (CloudNet ’19), 4-6 November 2019, Coimbra, Portugal.
- Alessio Sacco, Flavio Esposito, Princewill Okorie and Guido Marchetto. “LiveMicro: An Edge Computing System for Collaborative Telepathology” in Proc. of the 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge ’19), July 9th, Renton, WA.
- Xu Tao, Flavio Esposito, Alessio Sacco and Guido Marchetto. “A Policy-Based Architecture for Container Migration in Software Defined Infrastructures.” in Proc. of IEEE Conference on Network Softwarization (NetSoft). Short paper. 24-28 June 2019, Paris, France.