ScuDo

Scuola di Dottorato ~ Doctoral School

WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Electronics and Communications Engineering (30^{th}cycle)

Control plane optimization in Software Defined Networking and task allocation for Fog Computing

By

Tianzhu Zhang

******

Supervisor(s):

Prof. Paolo Giaccone, Supervisor

Prof. Marcello Chiaberge, Co-Supervisor

Doctoral Examination Committee:

Prof. Walter Cerroni, Referee, University of Bologna

Prof. Stefano Salsano, Referee, Università degli Studi di Roma Tor Vergata

Politecnico di Torino

2021

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.

Tianzhu Zhang
2021

* 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 mom, who means the whole world to me.

Acknowledgements

First of all, I would like to thank Prof. Paolo Giaccone. Whatever I have achieved in my Ph.D career, I owe them all to Paolo. During the three amazing years working with him, I have learned not only how to do research, but also means to forge ahead and build my career. As a professor with prominent academic achievements, he is always humble and patient. He is also one of the few in this world who always encourages me and tolerates my faults. The most amazing part for Paolo is that he really cares about his students and always thinks for their futures. I will cherish the moment we spent together during the rest of my life.

I would like to thank Prof. Marcello Chiaberge, who granted me the very chance to become a Ph.D candidate. I also appreciate the help and support from the folks of Telecom Italia JOL Swarm group.

Gratitude as well to Prof. Carla Fabiana Chiasserini and Prof. Andrea Bianco, it is a great honor for me to work with them. In addition, I would like to thank my colleagues Ahsan Mahmood and Abubakar Siddique Muqaddas for all the great experiences we spent together, I hope all of us achieve big in the future.

Thank Prof. Stefano Salsano and Prof. Walter Cerroni for reviewing my thesis, your valuable feedbacks and suggestions are highly appreciated.

Abstract

As the next generation of mobile wireless standard, the fifth generation (5G) of cellular/wireless network has drawn worldwide attention during the past few years. Due to its promise of higher performance over the legacy 4G network, an increasing number of IT companies and institutes have started to form partnerships and create 5G products. Emerging techniques such as Software Defined Networking and Mobile Edge Computing are also envisioned as key enabling technologies to augment 5G competence. However, as popular and promising as it is, 5G technology still faces several intrinsic challenges such as (i) the strict requirements in terms of end-to-end delays, (ii) the required reliability in the control plane and (iii) the minimization of the energy consumption. To cope with these daunting issues, we provide the following main contributions.

As first contribution, we address the problem of the optimal placement of SDN controllers. Specifically, we give a detailed analysis of the impact that controller placement imposes on the reactivity of SDN control plane, due to the consistency protocols adopted to manage the data structures that are shared across different controllers. We compute the Pareto frontier, showing all the possible tradeoffs achievable between the inter-controller delays and the switch-to-controller latencies. We define two data-ownership models and formulate the controller placement problem with the goal of minimizing the reaction time of control plane, as perceived by a switch. We propose two evolutionary algorithms, namely EVO-PLACE and BEST-REACTIVITY, to compute the Pareto frontier and the controller placement minimizing the reaction time, respectively. Experimental results show that EVO-PLACE outperforms its random counterpart, and BEST-REACTIVITY can achieve a relative error of $\leq 30\%$ with respect to the optimal algorithm by only sampling less than 10% of the whole solution space.

As second contribution, we propose a stateful SDN approach to improve the scalability of traffic classification in SDN networks. In particular, we leverage the OpenState extension to OpenFlow to deploy state machines inside the switch and minimize the number of packets redirected to the traffic classifier. We experimentally compare two approaches, namely Simple Count-Down (SCD) and Compact Count-Down (CCD), to scale the traffic classifier and minimize the flow table occupancy.

As third contribution, we propose an approach to improve the reliability of SDN controllers. We implement BeCheck, which is a software framework to detect "misbehaving" controllers. BeCheck resides transparently between the control plane and data plane, and monitors the exchanged OpenFlow traffic messages. We implement three policies to detect misbehaving controllers and forward the intercepted messages. BeCheck along with the different policies are validated in a real test-bed.

As fourth contribution, we investigate a mobile gaming scenario in the context of fog computing, denoted as Integrated Mobile Gaming (IMG) scenario. We partition mobile games into individual tasks and cognitively offload them either to the cloud or the neighbor mobile devices, so as to achieve minimal energy consumption. We formulate the IMG model as an ILP problem and propose a heuristic named Task Allocation with Minimal Energy cost (TAME). Experimental results show that TAME approaches the optimal solutions while outperforming two other state-of-the-art task offloading algorithms.

Contents

1 Introduction **1**

1.1 Inter-Controller Consensus and the Placement of Distributed SDN controllers 2

1.2 On-the-fly Traffic Classification and Control with a Stateful SDN approach 3

1.3 Misbehaving SDN controllers 5

1.4 Task Allocation for Integrated Mobile Gaming 6

I Control plane optimization in SDN **9**

2 Inter-Controller Consensus and the Placement of Distributed SDN Controllers **11**

2.1 Distributed SDN controllers 12

2.1.1 Data consistency models 13

2.2 Data-ownership models and delays trade-off 14

2.3 The controller placement problem 15

2.3.1 Results on the placement of controllers in ISP networks . . 16

2.3.2 Tradeoff between Sw-Ctr and Ctr-Ctr delay 16

2.4 Reaction time for the data-ownership models 20

2.4.1 Reactivity model for MDO model 21

2.4.2 Reactivity model for SDO model 21
2.4.3 Experimental results 23
2.5 OpenDaylight validation 25
2.5.1 Flow setup time for reactive forwarding in ODL 25
2.5.2 Experimental validation in a SDWAN 27
2.6 Optimal placement for minimum reaction time 32
2.6.1 Optimization model 32
2.6.2 Numerical results 34
2.7 Evolutionary placement algorithms for large networks 35
2.7.1 An evolutionary algorithm for Pareto-optimal placements . . 37
2.7.2 An evolutionary algorithm to minimize the reaction time . . 38
2.7.3 Performance of EVO-PLACE 40
2.7.4 Performance of BEST-REACTIVITY 44
2.8 Related works 45
2.9 Summary 46

3 On-the-fly Traffic Classification and Control with a Stateful SDN approach 48

3.1 Integrating an SDN controller with a traffic classification engine . . 49
3.1.1 On-the-fly traffic classification 49
3.1.2 Basic integrated approach 50
3.2 Stateful SDN approach for traffic classification 52
3.2.1 Simple CountDown (SCD) scheme 53
3.2.2 Compact CountDown (CCD) scheme 56
3.2.3 Countdown interruption 57
3.2.4 Comparison of approaches 57
3.3 Validation and Experimental Evaluation 58

3.3.1 Experimental comparison with standard OpenFlow switches 61
3.4 Summary 62

4 Misbehaving SDN controllers 64
4.1 Related works 65
4.2 Architecture of our behavioral checker 66
4.2.1 Network handler 69
4.2.2 Controller handler and detection policy 69
4.2.3 Forwarding policy 70
4.3 Validation and Experimental Evaluation 71
4.4 Summary 76

II Integrated Mobile Gaming 78

5 Task Allocation for Integrated Mobile Gaming 80
5.1 System model for IMG 81
5.1.1 Network model 81
5.1.2 Mobile game model 82
5.2 Problem formulation 84
5.2.1 Mobile node energy consumption 85
5.2.2 Response delay 87
5.2.3 Constraints 89
5.2.4 Problem complexity 92
5.3 TAME algorithm 93
5.4 Performance evaluation methodology 95
5.4.1 Benchmark schemes 95
5.4.2 Network scenarios 96

5.4.3 Task graph generation 97
5.5 Numerical results 100
5.5.1 Scenario with Wi-Fi Direct communications 101
5.5.2 Scenario with Bluetooth communications 103
5.5.3 Varying the delay between the PoA and the cloud 104
5.5.4 Multiple objects 105
5.5.5 Real-world games 105
5.6 Related works 106
5.6.1 Offloading for mobile computing 106
5.6.2 Mobile gaming 108
5.7 Summary 109

6 Conclusion **111**

References **114**

Chapter 1

Introduction

In the last few years the interest on the fifth generation (5G) cellular network has been growing. Indeed, with the promise of achieving higher capacity and data rate, shorter end-to-end latency, reduced maintenance cost, massive device connectivity as well as consistent QoE provisioning, 5G is deemed to be the future of communication networks [28]. To fulfil the promise of superior performance with respect to 4G network, 5G network adopts a variety of advanced techniques including small cells, massive MIMO, Device-to-Device (D2D) communications, Software Defined Networking (SDN) and Fog Computing. This thesis presents a handful of approaches to further enhance the performance, specifically with SDN and Fog Computing.

Software Defined Networking is a novel networking paradigm that keeps gaining popularity in both industry and academia since its inception. Unlike traditional networks, SDN breaks the vertical bundled planes, decouples the control logic and data forwarding, advocates centralized control and enables programmable networks [62]. 5G network is believed to benefit massively by integrating SDN technique [85]. For example, optimized resource provisioning is possible through the flexible and programmable control plane. Management of heterogeneous networks and deployment of new services become trivial through the cognitively centralized control, which in return reduces the operational costs. Security policies are also possible to be arranged in finer granularity through the logically centralized control plane. Although the myriad virtues SDN presents, integrating 5G network with SDN is still a challenging task [48].

The thesis is divided in two principle parts, the first devoted to SDN and the second to gaming applications in the context of fog computing. The contributions in each part are complementary.

1.1 Inter-Controller Consensus and the Placement of Distributed SDN controllers

The centralized network control of the Software Defined Networking (SDN) paradigm, which enables the development of advanced network applications, poses two main issues. First, limited reliability, due to the single point-of-failure. Second, the control traffic between the switches and the controller concentrates on a single server, whose processing capability is limited, arising scalability issues.

Distributed SDN controllers are designed to address the above issues, while preserving a logically centralized view of the network state, necessary to ease the development of network applications. In a distributed architecture, multiple controllers are responsible for the interaction with the switches. Thus, the processing load at each controller decreases, because the control traffic between the switches and the controllers is distributed, with a beneficial load balancing effect. Furthermore, resilience mechanisms can be implemented to improve network reliability in case of controller failures.

Distributed controllers adopt coordination protocols and algorithms to synchronize their shared data structures, enabling a centralized view of the network state for the applications. These schemes follow a consensus-based approach in which the coordination information is exchanged among controllers to reach a common network state. This introduces delays that, as discussed in Sec. 2.2 and shown experimentally in 2.5.1, can heavily affect the controller reactivity perceived by the switches. Indeed, each read/write of a shared data structure on one controller can be directed to a potentially different controller acting as “owner” of the data. Thus, the inter-controller delays must be considered along with the switch-controller delays when evaluating the control plane reactivity perceived by the switches. As a result, the optimal placement of the controllers in a given network should contemplate both kinds of delays. However, most of the related literature only concentrate on the delays of switch-controller interactions, while neglecting the delay introduced

by inter-controller communications. Our work, on the other hand, focuses on the controller placement problem by scrutinizing the impact of both delays.

The adoption of distributed SDN controllers in Wide Area Networks (SDWANs) is more challenging than in data center networks. In the latter case, the limited physical distances between network devices permits the installation of a separated network among the controllers (e.g., using dedicated Ethernet or InfiniBand connections), providing an out-of-band control plane. Conversely, SDWANs adopt an in-band control plane: the control messages and data packets share the same network infrastructure. Furthermore, the problem of supporting a responsive inter-controller communication is exacerbated by the geographical extension of SDWANs.

Our work in Chapter 2 provides the following contributions:

1. we discuss Pareto-optimal controller placements considering both controller-switch and inter-controller delays for some real ISP networks, based on the different data-ownership models;
2. we propose a low-complexity algorithm to find the approximated Pareto frontier in large-scale networks;
3. we define a set of formulas to compute the control plane reactivity. The formulas are validated through traffic measurements on an operational SDWAN;
4. we formulate the optimal controller placement problem as a Integer Linear Programming (ILP) problem, with the objective of minimizing the reactivity of control plane;
5. we propose an approximation algorithm to solve the above ILP problem and assess its performance considering topologies of real ISP networks.

1.2 On-the-fly Traffic Classification and Control with a Stateful SDN approach

SDN controllers can execute advanced traffic control policies, implemented as applications, which not only react to slow-varying states of the network (e.g., topology, link costs), but also to fast-varying states (e.g., congestion, incoming traffic). The

switches are responsible to inform the controller about the network state. Thus, applications with slow-varying network states are quite scalable, since the communication overhead to send the actual network state to the controller is negligible. Differently, when network state changes fast, the communication overhead between switches and controllers may become critical in terms of bandwidth and latency, thus posing severe limitations to the scalability of the system. This is particularly true for network applications running in real-time.

One possible way to improve the scalability of real-time network applications is to reduce the interaction between the switches and the SDN controller, by keeping some basic state within the switch. Thus the switch is allowed to take some simple decisions in an autonomous way (e.g. re-routing). This approach is denoted as *stateful* and there has been a growing interest towards it, as discussed in [29]. We remark that it is different from the traditional stateful approaches adopted in Ethernet switches and IP routers, where the state associated to the control plane can be complex (e.g., different level of abstractions in the topology related to the different formats of LSA messages in OSPF), but cannot be programmed in real-time.

In our work we address the integration of traffic control policies, that are specifically driven by a traffic classifier with an SDN approach. The addressed scenario is an SDN network in which flows are classified in real-time and the traffic control application applies some actions to them. E.g., if the traffic is classified as video-streaming, it is sent through a path with better bandwidth and/or delays. Or, if the traffic is classified as file-sharing, it is tagged as low priority. In this chapter we show that a stateful approach reduces the interaction between the switches and the SDN controller, which in turn is no more involved in a continuous interaction with the traffic classifier.

The main contribution of our work is to exploit the stateful approach, enabled by the OpenState [36] extension of OF, to integrate (i) the switches, (ii) the SDN controller and (iii) a traffic classifier in order to minimize the number of packets that are mirrored to the classifier by the switch, without the SDN controller's intervention. As shown in [39], relying on just a few packets (e.g., the first ones of a flow) for flow classification can improve the scalability of the overall system and achieve high data rates (e.g., Gbit/s).

We propose two solutions based on OpenState to configure the flow tables in different ways. The benefit of our approach is not only for the reduced load on

the SDN controller and on the traffic classifier, but also for the switches. Indeed, the memory occupancy of the internal flow tables is minimized. This result is relevant since flow tables are efficiently implemented with TCAMs (Ternary Content Addressable Memories), which are very fast, but much smaller (around 10^5-10^6 bytes) than standard RAM memories. As additional contribution of our work, we validate our solutions in a testbed with a Ryu controller interacting with an OF 1.3 switch (emulated with Mininet) and evaluate experimentally the actual memory occupancy typical of each of our solutions, in function of the number of concurrent flows. Thanks to our results, we can compute the maximum number of concurrent flows compatible with a maximum TCAM memory size.

1.3 Misbehaving SDN controllers

Unlike traditional IP networks, software-defined networking (SDN) decouples the network control logic and the forwarding functions. SDN controllers act as the “brain" of the network by making routing decisions and configuring the underlying simple forwarding devices in a logically centralized fashion. This refinement greatly simplifies network configuration and programmability.

However, just like any complex systems, SDN controllers are susceptible to misbehaviors, exacerbated by the centralized approach. Software bugs (e.g., persistent loops, synchronization failure, inconsistent state, response omission etc.) have been discovered in many popular SDN controllers. According to [51], the load balancer of Floodlight [4] may fail to distribute flows consistently and the POX [23] forwarding modules can delete rules installed by other modules. Furthermore, experiments in [40] detected totally 11 bugs on merely 3 applications of NOX [15]. Even the most practically relevant open source SDN controllers, namely OpenDaylight (ODL) [16] and ONOS [34], are not immune to bugs. According to [67], ODL can face flow deletion and instantiation failure bugs while link detection inconsistency and flow rules pending bugs are spotted in ONOS. These bugs can give high risk to misbehaving controllers. To make things worse, various security attacks to the control plane increases the possibility of misbehaving controllers. According to [49], SDN controllers including POX, Maestro [9], OpenDaylight and Floodlight are all susceptible to a diversity of security attacks on network topology and data forwarding. In addition, malicious network administrators can also misconfigure SDN controllers

to sabotage the network [70]. Most of the previous works improve the reliability and security of SDN control plane either by verifying the controllers' behaviors through complicated analysis such as model checking, or by advocating secure and dependable design of SDN controllers. Few of them provides solutions for real-time detection of misbehaving controllers.

In this paper, we present a behavioral checker, denoted as *BeCheck*, which is a module that relay the OpenFlow (OF) traffic between the controllers and the network switches, and detects misbehaving controllers in real-time, by comparing the instructions received by the controllers, which operate in locksteps. BeCheck resides *transparently* between the SDN controllers and the data plane, and neither the controllers nor the forwarding devices need to be modified for its presence. We propose a misbehavior detection policy, combined with different forwarding policies, and investigate experimentally the possible tradeoffs between the detection reliability and the reactivity of the application as perceived by the network switches.

1.4 Task Allocation for Integrated Mobile Gaming

In the second part of the thesis, we focus on D2D communications and fog computing as 5G enabling technologies. In D2D framework, communication can be guaranteed through cooperation of mobile devices in the vicinity, even without the facilitation of base stations. With the fast growth of mobile game industry, topics such as mobile gaming become increasingly interesting.

Mobile Cloud Gaming (MCG) [90] offers the possibility of running sophisticated games on thin mobile devices by offloading heavy tasks to the cloud. In this way, mobile games can be accessed on any device from anywhere with a simple setup. During the last few years, MCG has sharply motivated the expansion of mobile game industry. According to the report by Newzoo [13], in 2016 game users generated $99.6 billion of revenues, with an increase of 8.5% compared with 2015, and mobile games began to take a larger market share than their PC counterpart for the first time. Additionally, the Asia Pacific cloud gaming market is expected to witness a compound annual growth rate of 22% between 2016-2022 [26]. Finally, as a prevalent gaming model, MCG is widely supported by many famous online gaming platforms including Onlive [17], GaiKai [5] and GamingAnywhere [56].

In spite of the increasing popularity of MCG, several important challenges still need to be faced. First, offloading tasks to the remote cloud imposes extra communication latency, which may degrade users' quality of experience (QoE) and limit user coverage due to strict requirements on the response delay [64, 44]. Second, a large amount of bandwidth is required in order to guarantee high game quality, which in turn increases the costs for gaming service providers [90]. Third, cloud infrastructure requires more and more resources to cater the ever increasing demands of large scale mobile games (e.g., World of Warcraft) [66].

In this paper, we propose a general mobile gaming platform named Integrated Mobile Gaming (IMG), which combines the available resources of both the cloud and the neighbor mobile nodes, denoted as *mobile fog*, to run a game on behalf of the player's device. This new model shares MCG's idea of augmenting mobile devices with computation offloading, whereas it overcomes the intrinsic drawbacks of traffic offloading such as long response latency, wireless bandwidth consumption and limited available energy and computational resources. By partitioning a game into fine-granularity tasks and offloading part of them to either the cloud or neighbor nodes cognitively, not only the player's local device can save energy but also the available resources of the network can be better utilized. Similar to [60, 46], we partition games at both object and method level for the sake of offloading flexibility. Then we determine the offloading strategy by solving an integer linear programming (ILP) problem as well as by devising an approximation algorithm exhibiting excellent scalability.

As an example, consider the IMG scenario depicted in Fig. 1.1, including the player's device (denoted by n_0) and some mobile devices in proximity, a Point of Access (PoA) and a cloud server. Notably, the PoA can be a standard Wi-Fi AP or can be the access node of a cellular network. Beside the connection with the cloud server through the PoA, each of the mobile devices can exploit device-to-device (D2D) communications with the neighboring nodes, enabled by Bluetooth or Wi-Fi Direct technologies. According to the IMG model, the game running on n_0 is firstly partitioned into tasks, part of which can be offloaded to the neighboring devices (e.g., n_1 and n_2) or the cloud server, without degrading gameplay experience. Through task offloading, we aim to minimize the maximum energy consumption across all mobile nodes, instead of just at the player's device, since, as shown in [42], the case of multiple players in the network minimizing unilaterally the energy cost for their own devices may end up with a lose-lose situation.

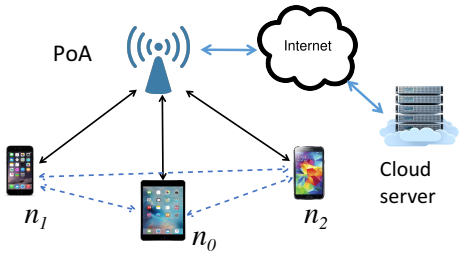

Fig. 1.1: Exemplifying scenario of Integrated Mobile Gaming (IMG).

To the best of our knowledge, this is the first work that exploits both cloud and mobile fog for energy-efficient mobile gaming. In mode detail, we provide the following contributions:

1. we formulate the optimal energy-aware task offloading problem for mobile gaming under the IMG model;
2. we devise an approximate algorithm, named Task Allocation with Minimal Energy cost (TAME), which is able to account for both computation and communication costs;
3. we evaluate the performance of TAME under synthetic and realistic scenarios, and show that it approximates very closely the optimal solution and outperforms other state-of-the-art offloading algorithms.

Part I

Control plane optimization in SDN

Chapter 2

Inter-Controller Consensus and the Placement of Distributed SDN Controllers

In this chapter, we consider a distributed Software Defined Networking (SDN) architecture adopting a cluster of multiple controllers to improve network scalability and reliability. We focus on the control traffic exchanged among the instances of controllers running in a cluster. There are two kinds of control traffic: namely the traffic exchanged between the switches and the controllers, as well as those exchanged between the controllers. The control traffic are exploited by the coordination and consensus algorithms to keep the shared data structures synchronized across all the physically distributed controllers. We advocate a careful placement of the controllers, that should take into account both the two kinds of control traffic. We evaluate, for some real ISP network topologies, the delay trade-offs for the controllers placement problem and we propose a novel evolutionary algorithm to find the corresponding Pareto frontier. Furthermore, we develop a simple model to estimate the reaction time perceived by the switches, which is accurately validated in an operational Software Defined WAN (SDWAN). Last but not least, we also formalize the optimization problem to minimize the reaction time and devise a novel approximation algorithm, whose performance is assessed against the optimal solver in real ISP network topologies.

Our work provides novel quantitative tools to optimize the planning and the design of the network supporting the control plane of SDN networks, especially when the network is very large and in-band control plane is adopted. We also show that for distributed controllers (e.g. OpenDaylight and ONOS), the location of the controller which acts as a leader in the consensus algorithm has a strong impact on the reactivity perceived by switches. The results and conclusions of this chapter are also reported in [99, 102].

This chapter is organized as follows: In Sec. 2.1 we provide an overview of distributed SDN architectures. In particular, we describe the interaction in the control plane, highlighting the importance of the inter-controller communications. In Sec. 2.2 we define two data-ownership models and explain the trade-offs for communication delays. In Sec. 2.3 we describe the controller placement problem and discuss the Pareto optimal solutions in real ISP topologies. In Sec. 2.4 we propose two simple analytical models to evaluate the reaction time based on the adopted data-ownership model. Section 2.5.1 applies the models to a real reactive forwarding application in OpenDaylight and provides an accurate experimental validation in an operational SDWAN. In Sec. 2.6 we formulate the ILP problem to minimize the average reaction time and numerically investigate its effects. To address the limited scalability of the optimal ILP solvers, we devote Sec. 2.7 to an evolutionary algorithm for optimized controller placement, and numerically evaluate their performance in real ISP topologies. In Sec. 2.8 we discuss the related work. In Sec. 2.9 we summarize the chapter.

2.1 Distributed SDN controllers

For a network under the administration of distributed SDN controllers, two control planes can be identified. First, the switch-to-controller plane, denoted as *Sw-Ctr plane*, supports the interaction between any switch and its controller (denoted as *master controller* for a given switch) through the controller "South-bound" interface. This interaction is usually devoted to data plane commands (e.g., through the OpenFlow (OF) [71] protocol) as well as to configuration and management of network switches (e.g. through OF-CONFIG or OVSDB protocols). Second, the controller-to-controller plane, denoted as *Ctr-Ctr plane*, allows the direct interaction among the controllers through the "East-West" interface. Indeed, SDN controllers exchange

heart-beat messages to ensure liveness. Controllers also need to *synchronize the shared data structures* to guarantee a consistent global network view.

2.1.1 Data consistency models

The traffic exchange on the Ctr-Ctr plane is crucial to achieve a consistent shared view of the network state. The network state (e.g., topology graph, the mapping between any switch to its master controller, the list of installed flow rules) is stored in shared data structures, whose consistency across the SDN controllers can be either *strong* or *eventual*. Strong consistency implies that simultaneous reads of some data occurring in different controllers always lead to the same result. Eventual consistency implies that simultaneous reads may eventually lead to different results, for a transient period. Different levels of data consistency heavily affect the reactivity and resilience of the control plane, as highlighted by [37, 80]. In OpenDaylight (ODL) [72] and Open Network Operating System (ONOS) [34], two state-of-the-art SDN controllers, strong consistency for the shared data structures is achieved using the Raft consensus algorithm [78]. For instance, the most recent version of ODL (e.g., Beryllium) provides a clustering service to support multiple instances of the controller, and the clustering module is built with a customized Raft algorithm [76], whose code is available in [25]. ODL clustering service organizes data of different modules into "shards", each of which is replicated to a configurable odd number of ODL instances. Similarly, ONOS release (2015-17) adopts the Raft algorithm for distributed data stores and mastership maintenance [81, 18], according to which data is shared across multiple shards, each of which is managed by an independent instance of Raft algorithm, thus ensures that operations on different shards can proceed independently.

Raft consensus algorithm is based on a logically centralized approach, in which any update is always forwarded to the single controller defined as *leader*. The leader then propagates the update to all the other controllers defined as *followers*. The update is considered committed whenever the majority of the follower controllers acknowledge the update. Sec. 2.4.2 provides a comprehensive description of the adopted protocol, based on the description provided in [78]. Note that the role of leader/follower controller for a data structure is a different concept from the role of master/slave controller for a switch.

In ONOS, some data can also be synchronized according to an eventual consistent model. Eventual consistency is achieved through the so called "anti-entropy" algorithm [18, 75], according to which local updates are propagated periodically in the background with a gossip approach: each controller picks at random another controller and compares the replicas. The differences are reconciled based on timestamps.

2.2 Data-ownership models and delays trade-off

The controller reactivity as perceived by a switch depends on the availability of the data necessary for its master controller. We can identity two distinct operative models.

In a *single data-ownership* (SDO) model, a single controller (denoted as "data owner") is responsible for the actual update of the data structures. Any read/write operation on the data structures performed by other controllers must be forwarded to the data owner. In this case, the Ctr-Ctr plane plays a crucial role for the interactions occurring in the Sw-Ctr plane, because some Sw-Ctr request messages (e.g., `packet-in`) trigger transactions with the data owner on the Ctr-Ctr plane. Thus, the perceived controller reactivity is also affected by the delay in the Ctr-Ctr plane. As discussed in Sec. 2.1.1, the SDO model is currently adopted in ODL and ONOS, for all the strong-consistent data structures managed by Raft algorithm: a local copy of the data structure is stored at each controller, but any read/write operation is always forwarded to the leader. With this centralized approach, data consistency is easily managed and the distributed nature of the data structures is exploited only during failures.

In a *multiple data-ownership* (MDO) model, each controller has a local copy of the data and can run locally read/write operations. A consensus algorithm distributes local updates to all other controllers. This model has the advantage of decoupling the interaction in the Sw-Ctr plane from the one in the Ctr-Ctr plane, thus improving the reactivity perceived by the switches. The main disadvantage is the introduction of possible update conflicts that must be resolved with ad-hoc solutions, and of possible temporary data state inconsistencies leading to network anomalies (e.g. forwarding loops) [80]. This model applies to generic eventual-consistent data structures, as the ones adopted in ONOS.

We now focus on the delay trade-off between Sw-Ctr and Ctr-Ctr planes. For the MDO model, the two planes are decoupled. Thus, small Sw-Ctr delays imply high reactivity of the controllers, whereas small Ctr-Ctr delays imply lower probability of network state inconsistency. But for the SDO model, the Ctr-Ctr delays also affect the perceived reactivity of the controllers, as shown in Sec. 2.4. Thus, reducing Ctr-Ctr delays is as important as reducing Sw-Ctr ones. Due to topological constraints, reducing one kind of delays may imply maximizing the other. The effects of such delays are particularly exacerbated in large networks, where propagation delays are not negligible. These observations motivate the exploration of possible trade-off in the following section.

2.3 The controller placement problem

Let N be the total number of switches in the network and C be the total number of controllers to deploy. The output of any placement algorithm can be represented by the vector π denoted as *placement configuration*:

$$\pi = [\pi_c]_{c=1}^{C} \tag{2.1}$$

where $\pi_c \in \{1,\ldots,N\}$ identifies the switch at which controller c is physically connected to. We assume in-band control traffic and all the controllers should be connected to distinct switches (equivalently, two controllers cannot be connected to the same switch), i.e. $\pi_c \neq \pi_{c'}$ for any $c \neq c'$. For the scope of our work, many controllers connected to a switch is equivalent in terms of delay trade-off as placing just one single controller to the same switch.

Let Ω be the set of all placement configurations; thus, the total number of possible placements is

$$|\Omega| = \binom{N}{C} \tag{2.2}$$

The optimal controller placement problem consists of finding the $\pi \in \Omega$ such that some cost function (e.g. the maximum or average Sw-Ctr delay) is minimized. It is an NP-hard problem for a generic graph, as discussed in [54].

The network topology is described by a weighted graph with N nodes where each node represents a switch; each edge represents the physical connection between the

corresponding switches and is associated with a propagation delay. Each controller is directly connected to a switch. We assume that the master controller of a switch is the one with the minimum Sw-Ctr delay. We also assume that all the communications are routed along the shortest path.

2.3.1 Results on the placement of controllers in ISP networks

To explore all the possible tradeoffs on the Sw-Ctr and Ctr-Ctr planes, we adopt an optimal algorithm (denoted EXA-PLACE) to exhaustively enumerate all possible controller placements and get all Pareto-optimal placements[1] and thus the corresponding Pareto-optimal frontier. For small/moderate values of network nodes N and number of controllers C, as considered in this section, the number of possible placements, evaluated in (2.2), is not so large and thus EXA-PLACE is computationally feasible. In Sec. 2.7.1 we will instead devise an approximated algorithm to find the Pareto frontier for large networks and/or large numbers of controllers.

Coherently with [54], we considered the topologies available in the *Internet topology zoo* repository [27], which collects more than 200 network topologies of ISPs at POP level. For each ISP, the repository provides the network graph, with each node (i.e. switch) labeled with its geographical coordinates. From these, we compute the propagation delay between the nodes and associate it as latency of the corresponding edge. For any given controller placement, we evaluate both the Sw-Ctr delay (as the average delay between the switches and their master controllers) and the Ctr-Ctr delay (as the average delay among controllers).

2.3.2 Tradeoff between Sw-Ctr and Ctr-Ctr delay

We report here the results only for HighWinds ISP, a world-wide network with 18 nodes. A preliminary version of our work [99] shows the detailed results for some other ISPs, qualitatively coherent with HighWinds.

Figs. 2.1-2.2 show the scatter plot with the Sw-Ctr and Ctr-Ctr delays achievable by all possible placements of 3 and 4 controllers, respectively. In total, all the possible

[1] When considering two performance metrics x and y to minimize, a solution (x_p, y_p) is *Pareto optimal* if does not exist any other configuration (x', y') dominating it, i.e. better in terms of both metrics; thus, it cannot be that $x' \leq x_p$ and $y' \leq y_p$. The set of all Pareto-optimal solutions denotes the *Pareto-optimal frontier*.

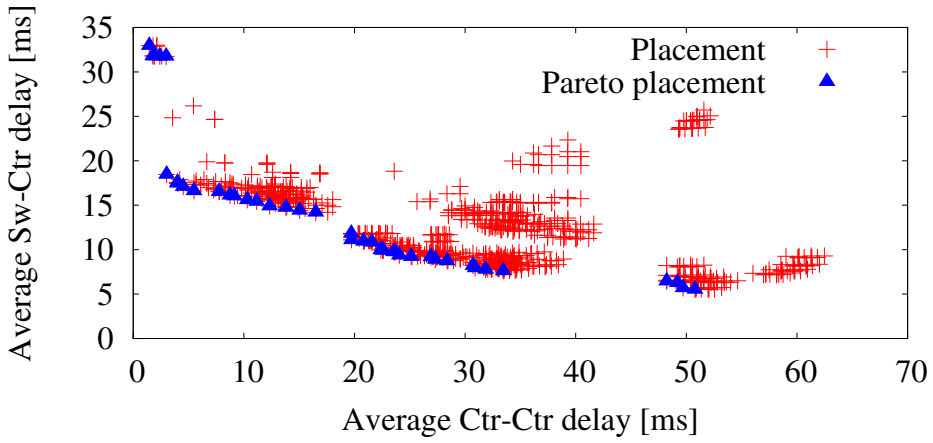

Fig. 2.1: Delay tradeoffs in HighWinds network with 3 controllers

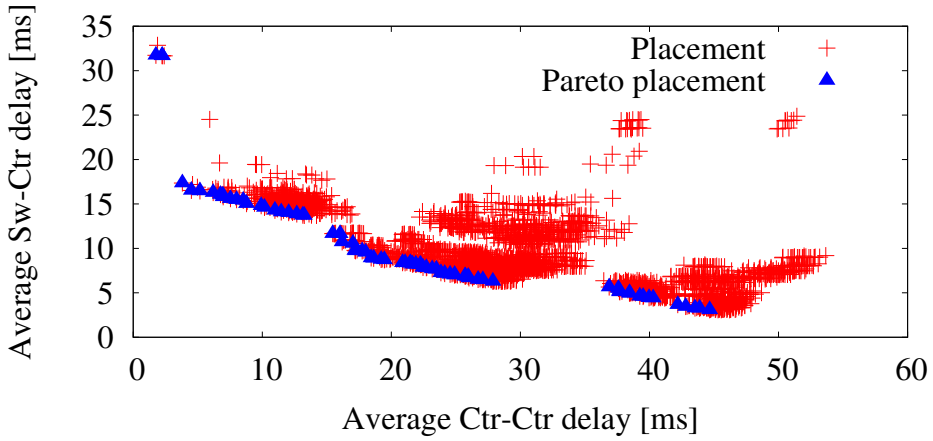

Fig. 2.2: Delay tradeoffs in HighWinds network with 4 controllers

$\binom{18}{3} = 816$ and $\binom{18}{4} = 3060$ different placements are shown; the corresponding Pareto-optimal placements are also highlighted. When comparing the two figures, the delays for Pareto-optimal points are smaller for 4 controllers, thanks to the higher number of controllers.

The fraction of placements corresponding to Pareto points is small, equal to $38/816 = 0.46\%$ and $64/3060 = 0.21\%$ for the two scenarios. This suggests that identifying Pareto points is difficult if using random sampling. To generalize our findings, Fig. 2.3 shows the fraction of Pareto placements for 114 different ISP topologies. When the number of controllers increases, this fraction decreases because of the larger solution space. For some networks, corresponding to small ISPs, the fraction is quite large (around 10%). But for most ISPs the fraction of Pareto

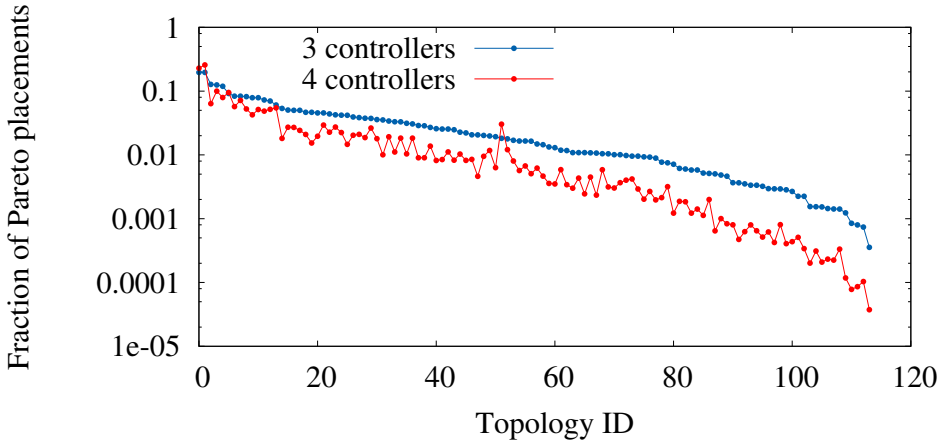

Fig. 2.3: Fraction of Pareto placements for deploying 3 and 4 controllers in 114 ISP networks.

placements becomes very small (less than 1%, and, in some large networks, less than 0.1-0.01%).

When considering the specific shape of the placements in Figs. 2.1-2.2, high (or small) Sw-Ctr delays imply small (or high) Ctr-Ctr delays, respectively. The graphs show the large variety of Pareto-optimal placements. We denote by $P1$ the Pareto point with the minimum Sw-Ctr delay (i.e. the most right-low point), and by $P2$ the one with the minimum Ctr-Ctr delay (i.e. the most left-high point). To understand the relative tradeoffs along the Pareto frontier from $P1$ to $P2$, we compute the *Sw-Ctr delay reduction*, defined as the ratio between the Sw-Ctr delay in $P2$ and the one in $P1$. Similarly, we define the *Ctr-Ctr delay reduction* as ratio between the Ctr-Ctr delay in $P1$ and the one in $P2$. Both reductions are ≥ 1 by construction. According to Fig. 2.1, the Sw-Ctr delay reduction is 6.0 whereas the Ctr-Ctr delay reduction is 34.8. In other words, if we allow the Sw-Ctr delay to increase by 6.0 times, then the Ctr-Ctr delay will decrease by 34.8 times, with strong beneficial effects on the time to reach consistency among the controllers.

To generalize our findings, Fig. 2.4 shows the Sw-Ctr and Ctr-Ctr delay reductions for 114 ISP topologies. Given the same topology, the Sw-Ctr delay reductions for 3 and 4 controllers are quite similar; interestingly, the Ctr-Ctr delay reductions are usually much higher (also 3 order of magnitude) than Sw-Ctr delay reductions. Thus, we can claim that Ctr-Ctr delays corresponding to Pareto points vary much more than Sw-Ctr delays in a generic network. Indeed, Ctr-Ctr delays are by construction

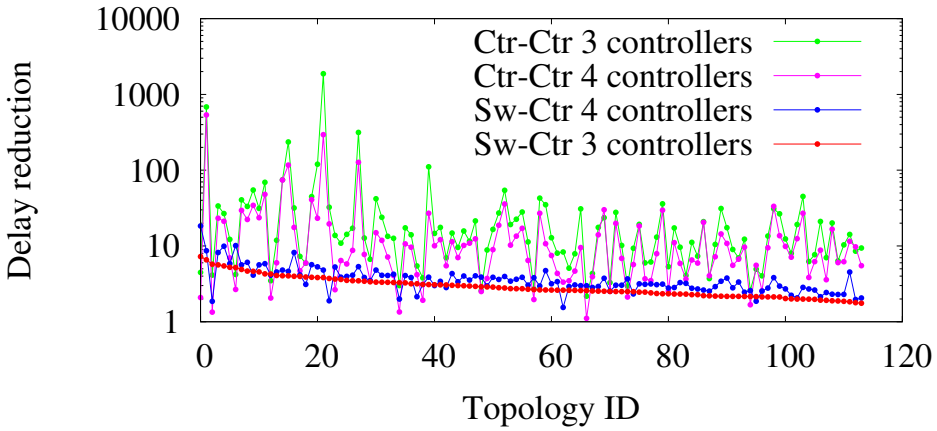

Fig. 2.4: Sw-Ctr and Ctr-Ctr delay reductions for 3 and 4 controllers in 114 ISP networks

between a minimum of 1-2 hops (when all the controllers are at the closest distance) and the maximum equal to the diameter of the network. The gains for the Sw-Ctr delays are lower, since the availability of multiple controllers decreases the maximum distance to reach the master controller from a switch. We can conclude that larger Sw-Ctr delays with respect to the minimum ones are much more compensated by much smaller Ctr-Ctr delays. This highlights the relevant role of the proper design of the Ctr-Ctr plane in SDN networks.

To better highlight the difference with respect to a standard placement problem that minimizes the Sw-Ctr delay, we define a new metric that evaluates the reduction in Ctr-Ctr delay whenever we accept some little increase in the Sw-Ctr delay. Let $P' = (d'_{sc}, d'_{cc})$ be the Pareto placement that minimizes the average delay, where d'_{sc} and d'_{cc} are the corresponding Sw-Ctr and Ctr-Ctr delays. Consider now the specific Pareto placement $P'' = (d''_{sc}, d''_{cc})$ with the minimum Ctr-Ctr delay such that the Sw-Ctr delay increases at most by a factor of 2, i.e. $d''_{sc} \leq 2d'_{sc}$. We define the *Ctr-Ctr reduction factor* as the ratio d'_{cc}/d''_{cc}, which evaluates the relative reduction of Ctr-Ctr delay whenever we accept to double the Sw-Ctr delay.

Figs. 2.5 reports the Ctr-Ctr reduction factors for different network topologies obtained for 3 and 4 controllers. The reduction is larger for 3 controllers, since the average distance between controllers is larger by construction. The gain for 3 controllers depends heavily on the particular topology. For the first 20% topologies, the growth in the Sw-Ctr delay is not compensated by the same reduction in Ctr-Ctr

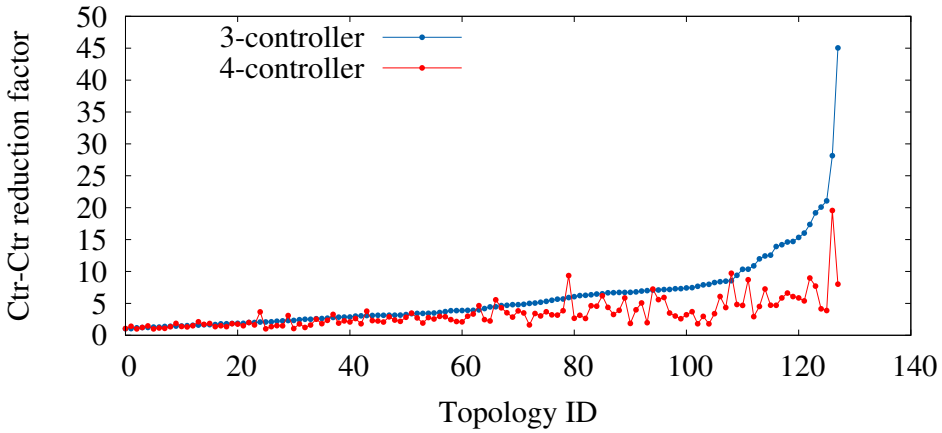

Fig. 2.5: Ctr-Ctr delay reduction when doubling the Sw-Ctr delay for 3 and 4 controllers.

delay. Instead, for the remaining 80% topologies, the reduction in the Ctr-Ctr delay is much higher, achieving also a factor 6; in this case, increasing a little the Sw-Ctr delay has a very strong beneficial effect on the Ctr-Ctr delay.

It is interesting to note that in same cases the reduction decreases from 3 to 4 controllers (as in Highwinds and HiberniaCanada). It can be shown that this is actually due to the peculiar clustered topology of the two ISPs, that are similar to a single star connected to one or two nodes very far (e.g. in Highwinds, we have one star-like cluster in North America and very few nodes in South America and in Europe).

2.4 Reaction time for the data-ownership models

After a global characterization of the Pareto-optimal frontier in terms of Sw-Ctr and Ctr-Ctr delays, we now specifically evaluate the *reaction time of the controller*, defined as the latency perceived by the switch when a new network event is generated. This time depends on the specific data-ownership model adopted by the controllers and on a specific combination of Sw-Ctr and Ctr-Ctr delays.

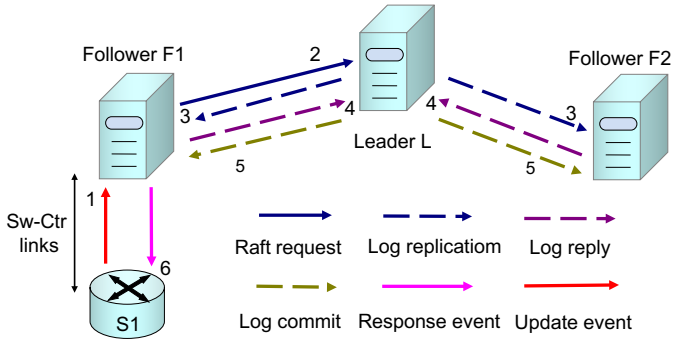

Fig. 2.6: Control traffic due to SDO model for an update event at the switch, coherent with Raft consensus algorithm.

2.4.1 Reactivity model for MDO model

In a MDO scenario, a generic event occurring at the switch (e.g. a miss in the flow table) generates a message (e.g., a `packet-in`) to its master controller, which processes the message locally and eventually sends back a control message to the switch (e.g., `flow-mod` or `packet-out` message). In the meanwhile, in an asynchronous way, the master controller advertises the update to all the other controllers. Thus, we can claim:

Property 1 *In a MDO scenario for distributed SDN controllers, the reaction time* T_R^{MDO} *of the controller perceived at the switch is:*

$$T_R^{MDO} = 2d_{sw\text{-}ctr} \tag{2.3}$$

being $d_{\text{sw-ctr}}$ the delay from the switch to its master controller, i.e. to the switch where its master controller is attached.

2.4.2 Reactivity model for SDO model

In a SDO scenario, we assume the exchange of messages coherent with the detailed description of Raft algorithm available in [78]. According to Raft implementation in ODL, the controller can operate as either leader or as one of the followers, for a specific data store (denoted “shard” in the following). As an example, Fig. 2.6 shows a general message exchange sequence in a cluster with 3 controllers (one leader and two followers), when an update event (e.g. `packet-in` message) for the shard

is generated at some switch (S1 in the figure), which receives a response message from its controller due to the update (e.g. *packet-out* message). The arrows show the exchange of messages in both Sw-Ctr and Ctr-Ctr planes triggered by the update event; the number associated to each arrow shows the temporal sequence of each packet. We have now two cases.

In the first case, S1's master controller is a follower for the shard (as depicted in Fig. 2.6). Thus, the switch sends the update event (message 1) to the master controller, which asks the leader to update the shard through a "Raft request" (message 2). Now the leader sends a "log replication" message to all its followers (message 3) and waits for the acknowledge from the majority of them ("log reply" in messages 4). Only at this point, the update is committed through a "log commit" (message 5) sent to all the followers. Thus, after receiving the commit message, S1's master controller can process the update on the shard and generate the response event (message 6) to the switch.

In the second case, S1's master controller is the leader for the shard. This case is identical to the previous one except for the "Raft request" message 2, now missing.

For both cases, the controller's reaction time perceived by switch S1 is given by the time between the update event and the response event messages. Let $d_{\text{sw-ctr}}$ be the communication delay between the switch and its master controller and $d_{\text{ctr-leader}}$ the communication delay from the master controller and the leader (being zero whenever the master is also leader). Assume a cluster of C controllers. Because of the majority-based selection, let $d_{\text{ctr*-leader}}$ be the communication delay between the leader and the farthest follower belonging to the majority (i.e. corresponding to the $\lfloor (C/2) \rfloor$-th closest follower). Observing Fig. 2.6, the reaction time is obtained by summing twice $d_{\text{sw-ctr}}$, twice $d_{\text{ctr-leader}}$ (only in the first of the above cases) and twice $d_{\text{ctr*-leader}}$. Thus, we can claim:

Property 2 *In a SDO scenario (e.g. adopting Raft consensus algorithm) for distributed SDN controllers, the reaction time T_R^{SDO} of the controller perceived at the switch is:*

$$T_R^{SDO} = 2d_{sw\text{-}ctr} + 2d_{ctr\text{-}leader} + 2d_{ctr*\text{-}leader} \tag{2.4}$$

Thus, the reaction time is identical to the one for MDO model plus either 2 or 4 times the RTT between the controllers, when the master controller is either leader or follower of the shard, respectively. Notably, the delays between controllers may be

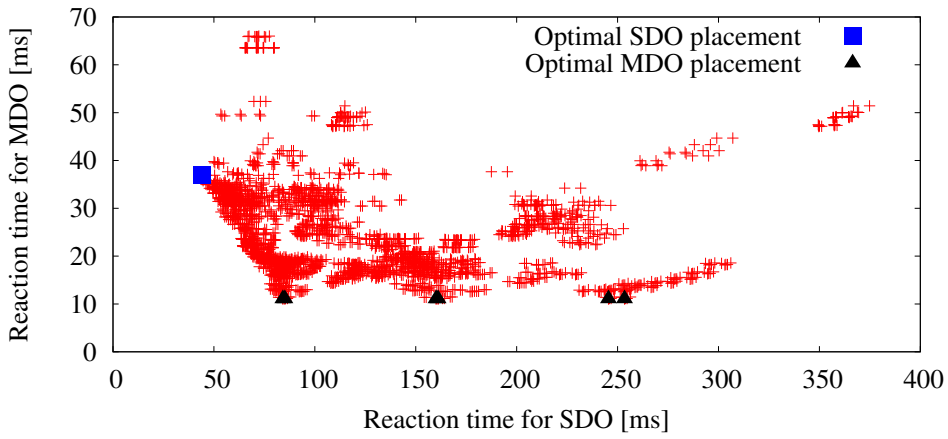

Fig. 2.7: Average reaction times in HighWinds network for all the placements. Optimal placements for the two data-ownership models are highlighted.

dominant for large networks as SDWAN, as also shown experimentally in 2.5.1. The model in (2.4), here obtained in a speculative way, will be tailored to a specific ODL network application and experimentally validated in an operational SDWAN running ODL, as described in details in 2.5.1. Thus, we can claim that the devised model is very accurate.

2.4.3 Experimental results

We investigate the reaction times achievable for different data-ownership models, based on Properties 1 and 2. Given a controller placement, we study the effect of selecting the data owner among the controllers on the perceived controller reactivity. We show the results just for HighWinds ISP, but the results for other ISP networks are available in [99].

In Fig. 2.7, we report the scatter plots of the average reaction times for the SDO and the MDO models when considering all possible controller placements and all possible selections for the data owner, in the case of 3 controllers. Each controller placement appears with 3 points aligned horizontally, one for each data owner, since the data owner selection does not affect the MDO reaction time. In the plots we have highlighted the placements with the minimum reaction time according to the SDO and MDO models. By construction, the minimum reaction time for the MDO is always smaller than the one for SDO model. From these results, the optimal

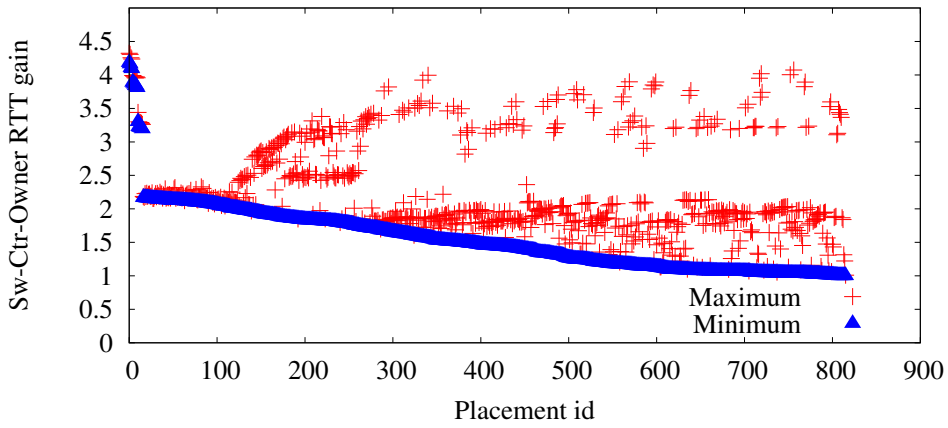

Fig. 2.8: Reaction time reduction in HighWinds network for the optimal selection of the data owner in the SDO model.

placements appear very different for the two data-ownership models and this fact motivates the need for a careful choice not only of the controller placement, but also of the data owner, in the SDO case.

To highlight the role of the proper selection of the data owner for the SDO model, in Fig. 2.8 we investigate the benefit achievable when considering the best data owner among the 3 available controllers, for the three ISPs under consideration. Assume that a given controller placement corresponds to three values of reaction times: d_1, d_2 and d_3, sorted in increasing order. The minimum *reduction factor* is defined as d_2/d_1 and the maximum reduction factor as d_3/d_1. We plot the delay reduction factor due to the optimal choice of the data owner, for any possible placement. For the sake of readability, the placements have been sorted in decreasing order of minimum reduction factor. Fig. 2.8 shows that a careful choice of the data owner in the SDO model decreases the reaction time by a factor around 2 and 4.

These results show that the selection of the data owner in the SDO model has the largest impact on the perceived performance of the controller, and can be easily optimally solved by considering all the possible C cases, after having fixed the controller placement.

2.5 OpenDaylight validation

2.5.1 Flow setup time for reactive forwarding in ODL

To validate and show the practical relevance of the SDO model devised in Sec. 2.4.2, we apply Property 2 to compute the flow setup time for the specific layer-2 forwarding application called "l2-switch" available in ODL, deployed on a generic topology. Notably, even if (2.4) is derived in a speculative way, in the following section we will show that it is *very accurate* from experimental point of view, and thus its relevance is practical. The same methodology can be applied to analyze other applications, given the knowledge of their detailed behavior.

ODL l2-switch application provides the default reactive forwarding capabilities and mimics the learning/forwarding mechanism at layer 2 of standard Ethernet switches. Anytime a new flow enters the first switch of the network, the corresponding ARP-request is flooded to the destination, and only when the ARP-reply is generated, the controller installs a forwarding rule at MAC layer in all the switches involved in the path from the source to the destination, and vice versa. The association of a MAC address to the switch port, needed for the learning phase, is distributed to the other controllers using Raft algorithm.

Assume a generic topology as shown in Fig. 2.9 connecting source host H1 to destination host H2, with every switch s attached to its master controller (denoted as $c(s)$) which can be either a follower or a leader (denoted as L) within the cluster. We assume initially empty flow tables in all the switches. At the beginning, the first ARP-request corresponding to a new flow from H1 is flooded in the whole network (loops are avoided by precomputing a spanning tree on the topology). Anytime the ARP packet is received at a switch, a packet-in is generated and the association [MAC source address, ingress port, switch identifier] is stored in the shared data store, in order to mimic the standard learning process. This means that at each switch, along the path from the source to the destination, a latency is experienced according to formula (2.4). When the ARP-request reaches the destination, H2 sends back an ARP-reply which generates a packet-in from the last switch (denoted as s') to the controller. This event generates another update since the controller learns the port of s' at which H2 is connected. *Only at this point*, the controller installs a flow rule across all the switches in the source-destination path and then the ARP-reply is switched back to the source. Thus the flow setup time can be evaluated

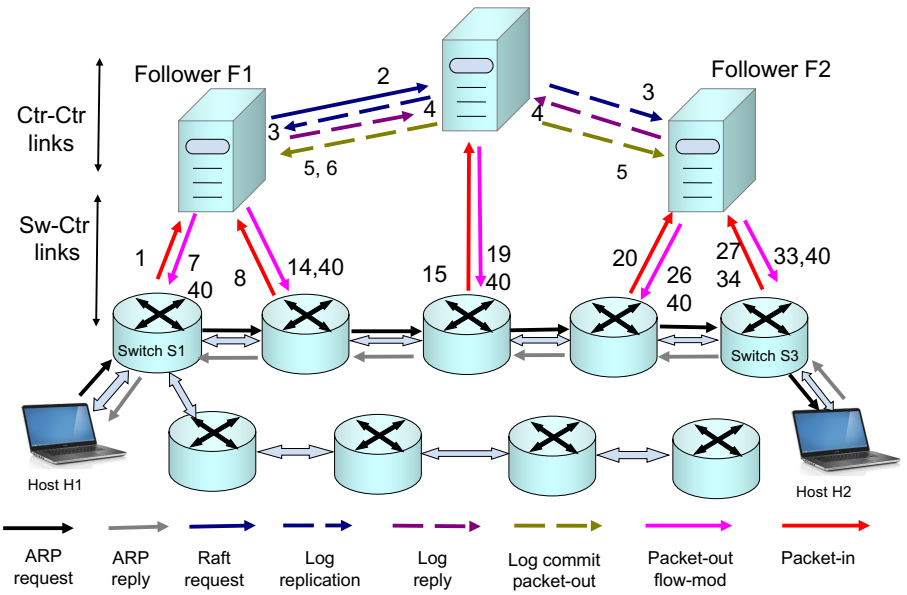

Fig. 2.9: The control traffic for l2-switch application in ODL. For the sake of clarity, we report just some sequence numbers. The packets sent with sequence 2-6 repeat as 9-13, 21-25, 28-32, 35-39. The packets sent with sequence 3-5 repeat as 16-18, since the master controller of the third switch along the path is also the shard leader. Only the messages between the controllers and the switches along the source-destination path are shown.

as *ARP reaction time* $t_{\text{r, ARP}}$, defined as the interval between the time when the H1's switch sends the packet-in message (due to the ARP-request) to its controller and the time when H1's switch receives back the packet-out/flow-mod messages (due to the ARP-reply). Note that the flow setup time depends on the considered application, that installs the flow rules across all the switches involved in the path just after the ARP reply at the destination host is generated.

Let $d_{i,j}$ be the propagation delay between nodes i and j, computed by summing all the contributions along the shortest path from i to j. Let $\mathscr{P}$ be the list of all the nodes involved in routing path from $H1$ to $H2$, in which the last switch s' appears twice. Thus, the total number of updates within a flow is $|\mathscr{P}|$. We can claim:

Property 3 *In OpenDaylight (ODL) running l2-switch application, the flow setup time can be computed as*

$$t_{r,ARP} = 2d_{H1,H2} + \sum_{s\in\mathscr{P}} (2d_{s,c(s)} + 2d_{c(s),L}) + 2|\mathscr{P}|d_{cnt*\text{-}follower} + |\mathscr{P}|t_c \quad (2.5)$$

Indeed, the first term in (2.5) represents the delay to send the ARP request and reply along the routing path, the second term represents the delay occurring for all the switches along the path (the final switch s' is double counted) due to the packet-in and the packet-out/flow-mod ($2d_{s,c(s)}$) and due to the Raft-request and log commit ($2d_{c(s),L}$), the third term represents the delay to get the acknowledgement from the majority for each of the updates, and the fourth term represents the computation time for each update at the controller (assuming to be constant and equal to t_c).

2.5.2 Experimental validation in a SDWAN

We validate Property 3 on a real and operational network. Since (2.5) depends mainly on the formula (2.4) obtained for SDO model, our results validate Property 2.

Specifically, we run a cluster of OpenDaylight (Helium SR3 release) controllers running the default "Simple Forwarding" application. We run our experiments in the JOLNet, which is an experimental SDN network deployed by Telecom Italia Mobile (the major telecom operator in Italy). JOLnet is an Openflow-based SDWAN, with 7 nodes spread across the whole Italy, covering Turin, Milan, Trento, Venice, Pisa and Catania. Each node is equipped with an OF switch and a compute node. The compute node is a server deploying virtual machines (VMs), orchestrated by OpenStack. Network virtualization is achieved though FlowVisor [88] and the logical topology among the OF switches is fully connected.

Due to the limited number of nodes in the JOLNet and the limited flexibility in terms of topology, we augment the topology with an emulated network running on Mininet [92] in one available compute node. We adopt the linear network topology of Fig. 2.10 with a variable number of nodes (from 3 to 36) and with one host attached at each switch. We generate ICMP traffic using the ping command among the different hosts. We run a single controller cluster with multiple ODL instances

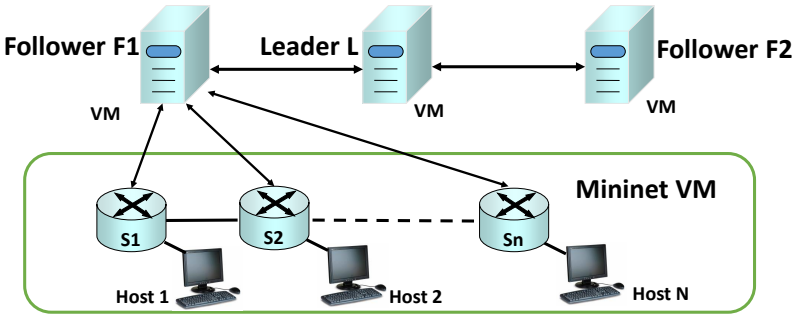

Fig. 2.10: Network configuration for the validation of the SDO model

running in different nodes of the JOLNet, in order to distribute geographically the controllers across Italy. The controllers instances and Mininet run individually on single VMs for a flexible placement across the nodes. Thanks to the large physical extension of the network, we experiment a large variety of scenarios, e.g. with large Sw-Ctr delays (when the VM of the master controller is located in a compute node far from the switch node) and/or large Ctr-Ctr delays (when the VMs of the controller instances are located in nodes far one from each other). By selecting the master controller of the switches, we change the data owner of the shared data structure within the cluster. We consider a cluster of 3 controllers and we measure the flow setup time $t_{\mathrm{r,ARP}}$ by comparing the timestamps of the packets obtained by using Wireshark as network sniffer at the Mininet interface towards the controllers.

As first step to validate Property 3, we evaluate the RTT among each pair of nodes in JOLNet and then we estimate the delay between any pair of nodes i and j as $d_{ij} = \mathrm{RTT}_{ij}/2$, required to apply (2.5). The experiments reported in the following refer to the scenarios using 3 JOLnet nodes, namely Turin, Milan and Pisa, to deploy the VMs. The measured RTT between Turin and Milan is 4 ms, whereas the one between Turin and Pisa is 132 ms.

As second step for the validation, we perform 100 measurements, by clearing the whole forwarding tables and restarting the controllers at each run. According to rigorous methodology, we evaluate the width I_{95} of the 95% confidence interval for the measurements and we compute the *measurement accuracy* as $\lambda = I_{95}/(2\bar{\mu})$, where $\bar{\mu}$ is the average measure. For each scenario and topology, the *relative error of the model* is instead computed as: $\delta = |M_i - T_i|/|T_i|$ where M_i is the average flow setup time according to the experiments and T_i is the flow setup time according to (2.5).

Table 2.1: Placement of the VMs for the experimented scenarios. "L": leader controller. "F1", "F2": follower controllers. "Net": Mininet.

Scenario	Turin VMs	Milan VMs	Pisa VMs
TT	Net, L, F1, F2	-	-
TMC	Net, F1	L, F2	-
TMF	Net	L, F1, F2	-
TPC	Net, F1	-	L, F2
TPF	Net	-	L, F1, F2

Table 2.2: Input parameters for the model, accuracy of measurements and relative error of the model

Scenario	$d_{\text{sw-ctr}}$	$d_{\text{ctr-ctr}}$	t_c	Experimental accuracy (λ)	Model error (δ)
TT	0.25 ms	0.25 ms	20 ms	1.2% - 2.7%	3.2%
TMC	0.25 ms	2.0 ms	20 ms	0.7% - 3.9%	5.2%
TMF	2.0 ms	0.25 ms	20 ms	0.6% - 3.6%	5.1%
TPC	0.25 ms	66 ms	20 ms	0.3% - 1.3%	9.2%
TPF	66 ms	0.25 ms	20 ms	0.6% - 2.3%	0.5%

We consider different scenarios, depending on the placement of the controllers and of Mininet across the different JOLNet nodes. We refer to the physical distance between the network nodes (emulated with Mininet) and the controllers (followers and leader) as "close" when the corresponding VMs are running in the same node, and "far" when on remote nodes. Table 2.1 lists all the experimented scenarios, discussed in the following section. In our cluster of 3 ODL controllers, the leader controller is denoted as "L" and the two followers are denoted as "F1" and "F2". Controller F1 is set to be master controller for all the switches in Mininet network. "Net" represents Mininet emulated network.

Experimental results

Table 2.2 summarizes the input parameters that have been used for the analytical formula in (2.5), and shows also the final experimental results in terms of measurement accuracy and of model error. The input parameters are obtained either by the RTT measurements when the VMs are located in different nodes, or by the steps explained below. In more detail:

- **Scenario TT (Turin-Turin):** We run the VMs of all the controllers and of Mininet in the same node, in order to evaluate the baseline latency due to

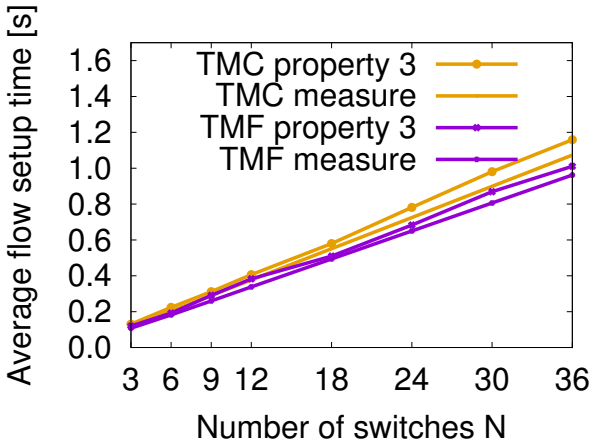

Fig. 2.11: Experimental results with the VMs running either in Turin or Milan

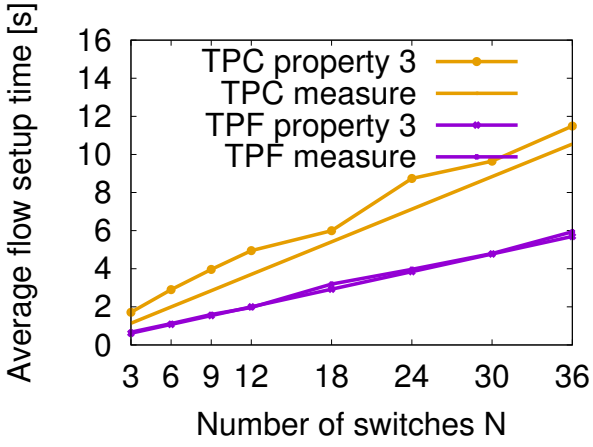

Fig. 2.12: Experimental results with the VMs running either in Turin or Pisa

the controller processing time and to the communication overhead (through the local virtual interfaces). First, we measure the communication delay between VMs, due to the local hypervisor running the different VMs, using ping command. We obtain 0.5 ms, thus we set the delay between the network and the controller, as well as between the controllers, equal to 0.25 ms. By running Mininet and measuring the flow setup time, we estimate an average processing latency of 20 ms, used as reference for all the other experiments. We run many experiments varying n_{sw} in the interval $[3,36]$ and observe a relative error of the model equal to 3.2%, so very small.

- **Scenario TMC (Turin-Milan-Close):** Leader L and Follower F2 of the cluster are located in Milan, whereas follower F1 is co-located with the network

in Turin node, thus all OF switches are close to their master controllers. The dominant term in (2.5) is the delay between controllers, equal to $4/2 = 2$ ms. Fig. 2.11 shows the average flow setup time computed according to (2.5) and the one measured. According to Table 2.1, the experimental results are quite stable for the different values of nodes. The measured value is quite close to the theoretical one, with a relative error 5.2%. Interestingly, the flow setup time in absolute values is always larger than 100 ms and can reach also 1.2 s when the network is quite large. Notably, this is mainly due to the interaction between the master controller and the leader controller.

- **Scenario TMF (Turin-Milan-Far):** All the controllers are located in Milan, thus all OF switches are far from their master controller. Thus, now the dominant term in (2.5) is the delay from the switch to the controller (2 ms). Fig. 2.11 compares the theoretical flow setup time with the measured ones. Now the relative error of the model is 5.1%. As in the previous case, the flow setup time can be very large, due to the latency in the interaction between the network and its master controller.

- **Scenario TPC (Turin-Pisa-Close)** All OF switches and their master controller F1 are located in Turin, whereas the leader controller and the other controller are located in Pisa. The dominant term is the delay between controllers ($132/2 = 66$ ms). As shown in Fig. 2.12, the measured value approaches the theoretical value with a relative error of 9.2%. The measured delays can range from 2 to 12 s as we vary the number of switches. These huge delays are due to the interaction between leader L and follower F1, as well explained by our model.

- **Scenario TPF (Turin-Pisa-Far)** All the controllers are located in Pisa with the network still in Turin. Due to the large delay between Turin and Pisa (66 ms), the dominant term in (2.5) is the delay between follower F1 and the network. In all the results shown in Fig. 2.12, the relative error (0.5%) of the model with respect to the theoretical value is very small. Also in this case, the flow setup time is huge in absolute terms (up to 6 s), due to the large delay between the network and the controllers.

In summary, our experimental results show clearly that the reactivity of controllers, as perceived by the network nodes, is strongly affected by the inter-controller

communications. Furthermore, they validate our analytical models for the SDO model, which appear to be very accurate for RAFT consensus algorithm.

2.6 Optimal placement for minimum reaction time

In this section, we present a mathematic ILP formulation of the optimal controller placement problem, in order to minimize the reaction time at the switches (or equivalently, maximize the network reactivity), for the SDO and the MDO models. Note that in our formulation we aim at finding also the best master controller to allocate to each switch. Indeed, differently from the results shown in the previous sections, we do not assume that the master controller of a switch is the geographically closest controller to that switch. We show that connecting a switch to the closest controller is not always optimal, when considering also the overhead due to coordination traffic among the controllers.

2.6.1 Optimization model

We consider the network graph describing the physical interconnection among the switches. Let $\mathscr{N}$ denote the set of switches; the number of switches is $N = |\mathscr{N}|$. Note that d_{mn} is defined as the delay between switch $m \in \mathscr{N}$ and $n \in \mathscr{N}$ on the shortest path. Let $\mathscr{C}$ denote the set of SDN controllers to be deployed; the number of controllers is $C = |\mathscr{C}|$.

We define the following binary decision variables, for any controller $i \in \mathscr{C}$ and any switch $n \in \mathscr{N}$:

- $X_{in} = 1$ iff controller i is placed at switch n;
- $Y_{ni} = 1$ iff controller i is the master controller of switch n.

According to standard approach, we also define some auxiliary decision variables, introduced to model the product of two binary decision variables while maintaining the problem linear.

- $\varepsilon_{ijmn} = X_{im} \times X_{jn}$, $\forall i, j \in C, \forall m, n \in \mathscr{N}$. Thus $\varepsilon_{ijmn} = 1$ only if controller i is placed at switch m and controller j is placed at switch n. To avoid non-linearities, the product can be equivalently defined as a set of linear constraints:

$$\varepsilon_{ijmn} \leq X_{im}, \quad \varepsilon_{ijmn} \leq X_{jn}, \quad \varepsilon_{ijmn} \geq X_{im} + X_{jn} - 1$$

- $\gamma_{nim} = X_{im} \times Y_{ni}, \forall i \in \mathscr{C}, \forall n,m \in \mathscr{N}$. Thus $\gamma_{nim} = 1$ iff controller i is placed at switch m and is the master controller of switch n. This is equivalent to:

$$\gamma_{nim} \leq Y_{ni}, \quad \gamma_{nim} \leq X_{im}, \quad \gamma_{nim} \geq Y_{ni} + X_{im} - 1$$

We define the following constraints:

- each controller is placed at only one switch:

$$\sum_{n \in \mathscr{N}} X_{in} = 1, \qquad \forall i \in \mathscr{C} \tag{2.6}$$

- each switch can host at most one controller:

$$\sum_{i \in \mathscr{C}} X_{in} \leq 1, \qquad \forall n \in \mathscr{N} \tag{2.7}$$

- each switch has exactly one master controller:

$$\sum_{i \in \mathscr{C}} Y_{ni} = 1, \qquad \forall n \in \mathscr{N} \tag{2.8}$$

We aim at minimizing the average reaction time perceived across the switches. We have two different linear formulations for the objective function, depending of the considered data-ownership model.

According to the MDO model, (2.3) defines the reaction time $T_R^{MDO}(n)$ perceived at switch n as $2d_{\text{sw-ctr}}(n)$. If switch n is connected to controller i as master controller, which is placed at switch m, then $d_{\text{sw-ctr}}$ is equal to $X_{im} \times Y_{ni} \times d_{mn}$. Now the average $d_{\text{sw-ctr}}$ across all the switches can be computed as

$$\frac{1}{N} \sum_{n \in \mathscr{N}} d_{\text{sw-ctr}}(n)$$

and thus, after neglecting constant factors, the objective function for the MDO problem is:

$$\min \sum_{m \in \mathscr{N}} \sum_{i \in \mathscr{C}} \sum_{n \in \mathscr{N}} \gamma_{nim} \times d_{mn} \tag{2.9}$$

For the SDO model, we aim at minimizing the average reaction time T_R^{SDO} computed as (2.4). Note that, as discussed in Sec. 2.4.3, the choice of the leader controller, acting as data owner, affects strongly the reaction time. In order to find the optimal choice of the leader controller we assume, without loss of generality, that controller 1 is also the leader. Observe now that the reaction time $T_R^{SDO}(n)$ perceived at switch n is composed by three terms, according to (2.4). The first term, $d_{\text{sw-ctr}}(n)$, is equal to the MDO case. The second term, $d_{\text{ctr-leader}}$, can be computed setting that switch n has controller i as master controller, which is placed at switch m, and that the leader controller 1 is placed at switch s, i.e. $Y_{ni} \times X_{im} \times X_{1s} \times d_{ms} = \gamma_{nim} \times \varepsilon_{1ism} \times d_{ms}$; then, we average across all the switches. We compute the last term, $d_{\text{ctr*-leader}}$, by approximating the median of the delay between the leader controller and the other controllers with its average value. If the leader controller is placed at switch s and that controller i is placed at switch m, then the delay among them is $X_{im} \times X_{1s} \times d_{ms} = \varepsilon_{1ism} \times d_{ms}$ and the overall average is

$$\frac{1}{C} \sum_{i \in \mathscr{C}} \sum_{m \in \mathscr{N}} \sum_{s \in \mathscr{N}} \varepsilon_{1ism} \times d_{ms}$$

By combining all the above terms, the objective function for the SDO model can be formalized as follows:

$$\begin{aligned} \min \frac{1}{N} \sum_{m \in \mathscr{N}} \sum_{i \in \mathscr{C}} \sum_{n \in \mathscr{N}} \gamma_{nim} \times d_{mn} + \\ \frac{1}{N} \sum_{n \in \mathscr{N}} \sum_{i \in \mathscr{C}} \sum_{m \in \mathscr{N}} \sum_{s \in \mathscr{N}} \gamma_{nim} \times \varepsilon_{1ism} \times d_{ms} + \\ \frac{1}{C} \sum_{i \in \mathscr{C}} \sum_{m \in \mathscr{N}} \sum_{s \in \mathscr{N}} \varepsilon_{1ism} \times d_{ms} \end{aligned} \quad (2.10)$$

Note that the second term appears to be non-linear, but actually it can be converted to a linear relation as we showed at the beginning of Sec. 2.6.1 by defining a new set of auxiliary variables. We omit the details for the sake of space.

2.6.2 Numerical results

We implemented an optimal solver for the optimal placement problem in (2.9) and in (2.10) throughout the Gurobi solver [8].

Figs. 2.13-2.14 show the placement to minimize the reaction times, by solving (2.9) and (2.10) for both data-ownership models, when 3 controllers are deployed in the Highwinds network. As expected, the results obtained for the two models are completely different. For the MDO case, one controller is placed in each continent, in order to minimize the average delay between the switches and their master controllers. For the SDO case, instead, all the controllers are placed close in the continent with the higher number of switches, and specifically the leader is chosen close to the continent (Europe) with the second largest number of switches. This is intuitively the best tradeoff between Sw-Ctr delays and Ctr-Ctr delays.

Fig. 2.15 shows the minimum average reaction time for both SDO and MDO considering 3 and 4 controllers, for the smallest 89 ISP networks available in [27]. We could not run our solver for the largest topologies, because of the limited scalability of the ILP approach. In all the topologies, the optimal reactivity is mainly affected by the adopted data-ownership model, and not by the number of controllers. The MDO model achieves reaction times that are 1-2 orders of magnitude smaller than the SDO model, showing the non-negligible impact of the Ctr-Ctr traffic on the perceived performance at the switches for the SDO model.

Considering the actual master controller chosen for each switch, based on our experiments, in the MDO model all the switches are connected to the closest controller, as expected. But for SDO model, each switch is connected to the first controller along the its shortest paths to the leader controller. Thus, for the SDO model *the closest controller may not be the best solution*. Indeed, it may be convenient for a switch to connect directly to the leader controller, even if farther than the closest controller, in order to reduce the impact of the Ctr-Ctr traffic among the controllers. In the considerer 89 topologies, on average 25% and 35% of the switches were not connected to their closest controller for the 3 controllers and 4 controllers scenarios, respectively.

2.7 Evolutionary placement algorithms for large networks

When commenting the numerical results in Secs. 2.3 and 2.6, we noticed that for the largest ISP topologies (in terms of number of switches and links) we could not run

Fig. 2.13: Placement for MDO in HighWinds.

Fig. 2.14: Placement for SDO in HighWinds.

the exhaustive search (Sec. 2.3.1) and the ILP solver (Sec. 2.6.2) to get the optimal placements, due to the limited scalability of the considered optimal approaches.

In this section, we present two evolutionary algorithms suitable for large networks. The first one, denoted as EVO-PLACE and described in Sec. 2.7.1, finds a set of Pareto controller placements. The second one, denoted as BEST-REACTIVITY and described in Sec. 2.7.2, finds the final placement that minimizes the average reaction time perceived at the switches.

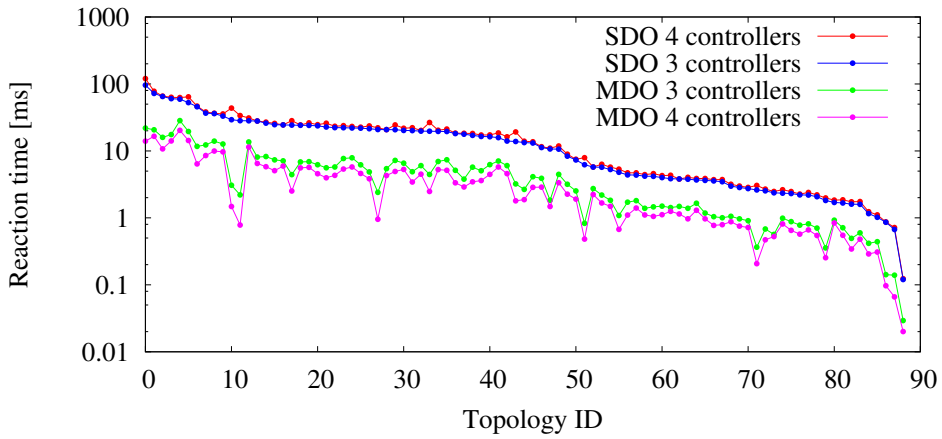

Fig. 2.15: Optimal control plane reactivity obtained with the optimization framework when deploying 3 and 4 controllers, in 89 ISP networks.

2.7.1 An evolutionary algorithm for Pareto-optimal placements

The basic idea of our algorithm is to discover new non-dominated solutions by perturbing the current set of Pareto solutions for the controller placement. Specifically, starting from a given controller placement in the network, we may get a new controller placement with better Ctr-Ctr delay by putting the controllers closer, and a new controller placement with better Sw-Ctr delay by distributing the controllers more evenly in the network. By continuously performing such perturbation, we achieve a good approximation of the Pareto frontier for the placement problem.

As term of comparison for our algorithm, we define a basic randomized algorithm, denoted as RND-PLACE and reported in Algorithm 1, to find a set of Pareto solutions just using a random sampling. The input parameters are the number of controllers C, the number of nodes N and the number of iterations $i_{\max}$. We assume that function RANDOM-PERMUTATION(N,C) (called in step 4) provides the first C elements of a random permutation of size N, with $C \in [1,N]$; its complexity is $O(C)$ thanks to the classical Knuth shuffle algorithm. Let P be the current set of all Pareto (i.e. non-dominated) solutions. At each iteration, a new placement is generated (step 4). Now function ADD-PRUNE eventually adds π to P. More precisely, if π is dominated by any Pareto solution in P, then it not added to P since it is not Pareto (step 10). Instead, if any current solution $p \in P$ is dominated by π, then it is removed (step 12) and then π is added as new Pareto solution (step 13). ADD-PRUNE returns true if π was added successfully, otherwise it returns false.

The set P returned by RND-PLACE at the end of $i_{\max}$ iterations collects all the Pareto placements found by the procedure, and corresponds to an approximation of the optimal Pareto frontier for the controller placement problem. The randomized approach is simple but quite inefficient in terms of complexity, since it takes around $|\Omega| \log |\Omega| + 0.58|\Omega|$ iterations (thanks to the well known results about the coupon collector problem), where $|\Omega|$ is the total number of placements (see (2.2)), to approach the exhaustive search and find the optimal Pareto placements.

We modify RND-PLACE to exploit an evolutionary approach to boost the efficiency of the random sampling. Algorithm 2 reports the pseudocode of our proposed EVO-PLACE. At each iteration, the algorithm selects a random placement π (step 4) and try to add to P, as in RND-PLACE. If the addition is successful (i.e. π is Pareto), then π is perturbed (step 7) and the new placement π' is eventually added to P (step 8). The loop for the perturbation ends when the newly perturbed solution cannot be added to P, since it is dominated by other solutions (steps 6-9). The perturbation phase is described by DECREASE-CTR-CTR-DELAY, whose pseudocode is reported in Algorithm 3.

DECREASE-CTR-CTR-DELAY perturbs the given placement solution π by decreasing the Ctr-Ctr delay. Its main idea is to move the farthest controller closer to the others. Indeed, in steps 2-3 the average distance is computed for each controller to all the others (actually, we omit the division by $C-1$ since it is useless for the following steps). We define d_{ij} as the minimum delay from node i to j, based on the propagation delays in the network topology. Now we choose c' as the controller with the maximum average delay towards the others (step 4) and find c'' as the closest controller to c' (step 5). Now we move c' one hop towards c'' (steps 6) along the shortest path from c' to c''; note that the check that c'' is at least 2 hops far from c' guarantees that the movement is possible. As a result, DECREASE-CTR-CTR-DELAY decreases the average Ctr-Ctr distance most of the times.

2.7.2 An evolutionary algorithm to minimize the reaction time

We adopt the same approach of EVO-PLACE to find the best placement that minimizes the reaction time according to the MDO and SDO models, computed according to Property 1 and 2 respectively. The pseudocode in Algorithm 4 describes the proposed evolutionary approach. At each iteration, a random placement (step 4)

Algorithm 1 Random algorithm for finding Pareto controller placements

procedure RND-PLACE($C, N, i_{\max}$)
  $P = \emptyset$ ▷ Init the set of Pareto solutions
  for $i = 1 \rightarrow i_{\max}$ **do** ▷ For $i_{\max}$ iterations
    π =RANDOM-PERMUTATION(N, C)
    ADD-PRUNE(P, π)
  return P
procedure ADD-PRUNE(P, π)
  for all $p \in P$ **do**
    if π is dominated by p **then**
      return false ▷ No addition of π
    if p is dominated by π **then**
      $P = P \setminus \{p\}$ ▷ Remove p
  $P = P \cup \{\pi\}$ ▷ Add π since not dominated
  return true ▷ Successful addition of π

Algorithm 2 Evolutionary algorithm for finding Pareto controller placements

procedure EVO-PLACE($C, N, i_{\max}$)
  $P = \emptyset$ ▷ Init the set of Pareto solutions
  for $i = 1 \rightarrow i_{\max}$ **do** ▷ For $i_{\max}$ iterations
    π =RANDOM-PERMUTATION(N, C)
    success_flag=ADD-PRUNE(P, π)
    while (success_flag=true) **do**
      π' =DECREASE-CTR-CTR-DELAY(π)
      success_flag=ADD-PRUNE(P, π')
      $\pi = \pi'$
  return P

Algorithm 3 Perturb a given controller placement π to decrease Ctr-Ctr delay

procedure DECREASE-CTR-CTR-DELAY(π)
  for $c = 1 \rightarrow C$ **do**
    $h_c = \sum_{k \neq c} d_{\pi_c \pi_k}$ ▷ Total delay from c
  $c' = \arg\max_c \{h_c\}$ ▷ Farthest controller
  $c'' = \arg\min_{c \neq c'} \{d_{\pi_c \pi_{c'}}\}$ ▷ c''s closest cnt.
  n=**find** first node in the shortest path from c' to c''
  if $n \neq \pi_{c''}$ **then**
    $\pi_{c'} = n$ ▷ Move c' into n
  return π

Algorithm 4 Evolutionary algorithm to find Placement with minimum reaction time

```
1: procedure BEST-REACTIVITY(C, N, i_max)
2:     π_best = {}                                   ▷ Init the best solution
3:     for i = 1 → i_max do                          ▷ For i_max iterations
4:         π = RANDOM-PERMUTATION(N, C)
5:         UPDATE-BEST(π, π_best)
6:         π = DECREASE-CTR-CTR-DELAY(π_best)
7:         UPDATE-BEST(π, π_best)
8:     return π_best
9: procedure UPDATE-BEST(π, π_best)
10:    if REACTION-TIME(π)<REACTION-TIME(π_best) then
11:        π_best = π                                ▷ Found better placement
```

is generated in order to possibly escape from local minima. Another candidate placement is obtained (step 6) by perturbing the optimal candidate solution π_{BEST} found adopting DECREASE-CTR-CTR-DELAY. The reaction time for each new candidate solution is evaluated in step 7 by calling UPDATE-BEST. This function (step 9) exploits the analytical formulas to compute the reaction time to check whether the given placement has a smaller reaction time than the current candidate optimal. As an example, the formula adopted in Property 3 (see 2.5.1) can be exploited to compute the reactivity for the standard reactive layer-2 forwarding application in ODL controller.

2.7.3 Performance of EVO-PLACE

We compare the performance of EXA-PLACE, RND-PLACE and EVO-PLACE on different networks with varying number of controllers. For a fair comparison, we run EVO-PLACE after having fixed $i_{\max}$ and record the actual total number of analyzed placements. Then we use such value to set the number of placements considered by RND-PLACE. Thus, all the results comparing EVO-PLACE with RND-PLACE are obtained with the same number of analyzed placements.

In Fig. 2.16, we show the results for the Garr network, a nation-wide Italian ISP, taken from [27], with 35 nodes, for the case of 3 controllers. Thus, $N = 35$, $C = 3$ and thus $|\Omega| = \binom{35}{3} = 6,545$ are all the possible placements evaluated by EXA-PLACE. The corresponding Pareto points represent the optimal Pareto frontier, used as a reference for the frontiers obtained with the other algorithms. The graphs show

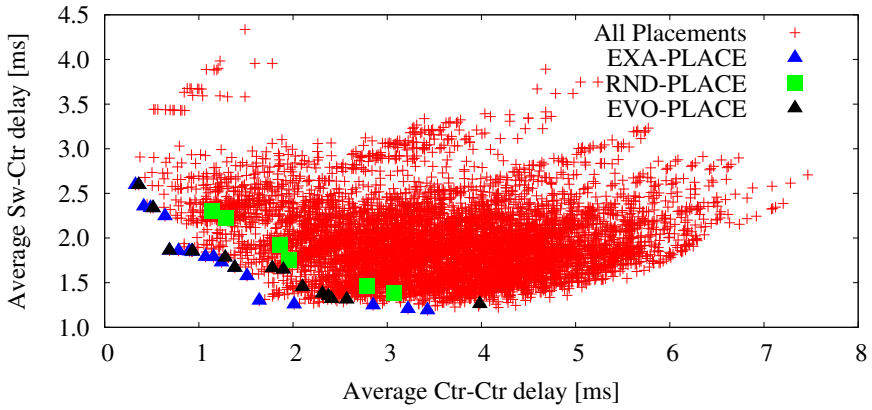

Fig. 2.16: Optimal Pareto frontier (EXA-PLACE) and its approximations (RND-PLACE, EVO-PLACE with $i_{\max} = 50$) for the placement of 3 controllers in Garr network

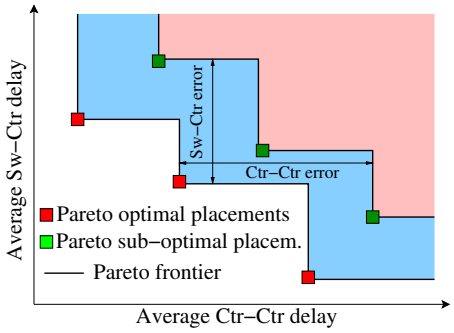

Fig. 2.17: Definition of Ctr-Ctr error and of Sw-Ctr error with respect to the optimal Pareto frontier

the sub-optimal Pareto points obtained by RND-PLACE and EVO-PLACE running for $i_{\max} = 50$ iterations, corresponding to a sampling fraction equal to 0.9% of all possible solutions. From the figure, the Pareto placements computed by EVO-PLACE appear to approximate much better the optimal ones than RND-PLACE, given the same number of iterations.

In order to evaluate in a quantitative way the “distance" between the optimal Pareto frontier computed by EXA-PLACE and the approximated ones obtained by RND-PLACE and EVO-PLACE, we define two error indexes, as depicted in Fig. 2.17, derived from classical volume based performance indexes for Pareto sets [77]: (i) the *average Sw-Ctr error*, computed as the average vertical distance between the optimal Pareto frontier and the approximated Pareto frontier, (ii) the *average Ctr-Ctr error*, computed as the average horizontal distance between the two frontiers.

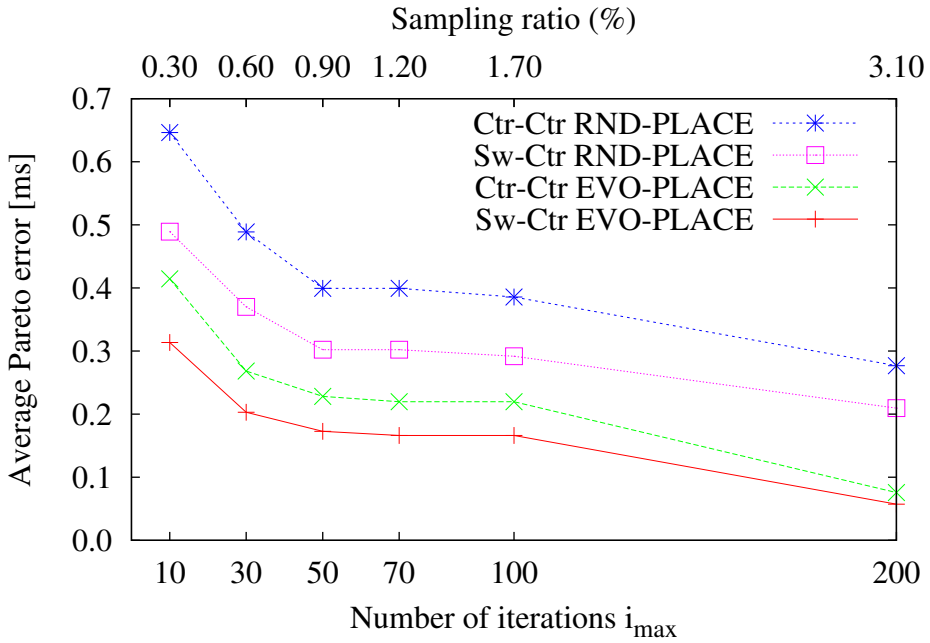

Fig. 2.18: Pareto frontier error with 3 controllers for Garr.

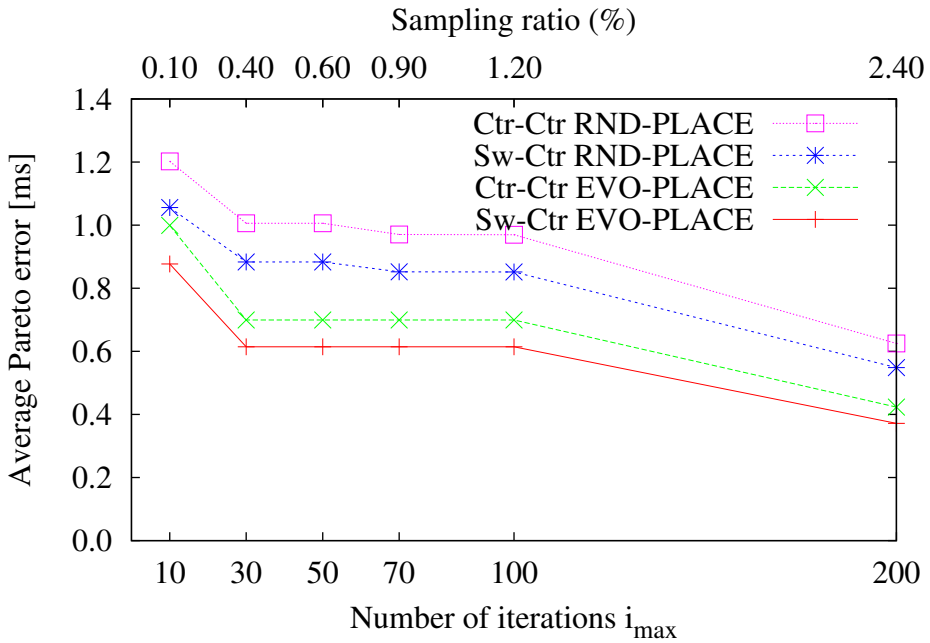

Fig. 2.19: Pareto frontier error with 3 controllers for China-Telecom.

Fig. 2.18 shows the behavior of the two errors in function of the number of iterations, in the same scenario considered in Fig. 2.16. Each experiment, for a given

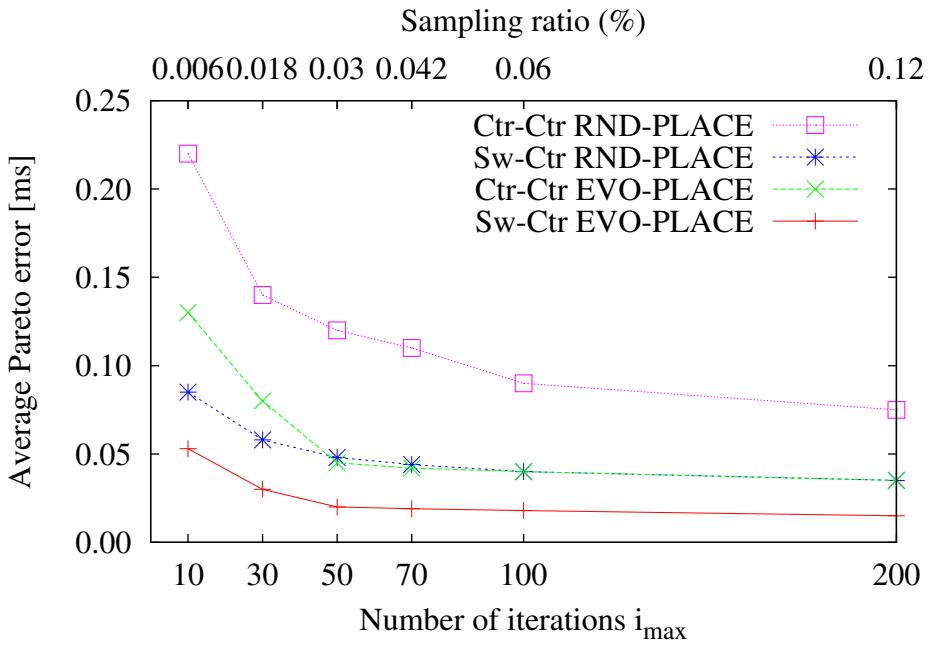

Fig. 2.20: Pareto frontier error with 3 controllers for ITC-Deltacom.

number of iterations, is repeated multiple times to get an accurate estimation of the error. When increasing the number of iterations, the Pareto errors decrease, thanks to the larger space of considered solutions. As already observed, we expect that the Ctr-Ctr delays are larger than the Sw-Ctr delays in absolute terms, and thus the corresponding errors are following the same behavior. After 200 iterations both the Sw-Ctr error and the Ctr-Ctr error are around 3 times smaller than the errors obtained for 10 iterations, and in absolute terms very small (less than 0.1 ms) for EVO-PLACE. The advantage of EVO-PLACE with respect to RND-PLACE tends to increase with the number of iterations, indeed the errors for EVO-PLACE are between 1.5× (for low number of iterations) and 3× (for high number of iterations) smaller than RND-PLACE. In general, an accurate estimation of the Pareto frontier can be achieved with a sampling ratio equal to 1-3% of the total solution space.

We extend our investigation to other larger topologies, for which EVO-PLACE is much faster than EXA-PLACE. Figs. 2.19 and 2.20 show the errors in the Pareto frontiers obtained for China-Telecom and ITC-Deltacom networks, respectively, taken from [27]. In China-Telecom (38 nodes), the Pareto errors decreases by a factor 2.5 from 10 to 200 iterations, and the relative gain of EVO-PLACE with respect to RND-PLACE is around 1.2-1.5, decreasing for larger sampling space. Also in this

case, around 1-2% of the sampling space is enough to obtain an accurate estimation of the Pareto region. Fig. 2.20 shows the errors for ITC-Deltacom network, which is a large USA ISP with 100 nodes. Despite the large size of the network, after 50 iterations (corresponding to 0.03% of sampling ratio) all the errors tend to stabilize for EVO-PLACE, showing a relative gain with respect to RND-PLACE which is always more than 2.

The results obtained for the above 3 ISPs show the effectiveness of the evolutionary approach with respect to the simple random sampling. Notably, also in absolute terms all the errors are small, even if their actual values depend on the geographical area covered by each network.

We have also evaluated the scenario with Colt-Telecom from [27], an Europe-wide ISP covering 149 nodes, in the case of 10 controllers. In this scenario EXA-PLACE cannot run since the total number of possible placements is larger than 10^{15} and thus we cannot evaluate the average errors with respect to the optimal Pareto points. We instead observe that EVO-PLACE is always outperforming RND-PLACE by reducing the average Sw-Ctr and Ctr-Ctr delays of $0.25 - 1$ ms.

In conclusion, for all the scenarios we investigated, we have observed a better Pareto frontier obtained by EVO-PLACE with respect to RND-PLACE, given the same number of considered placements and thus the same computation complexity. Thus, the evolutionary approach adopted in EVO-PLACE appears efficient in finding the Pareto placements for a given network topology, especially when the network is large and an exhaustive approach is not anymore feasible.

2.7.4 Performance of BEST-REACTIVITY

We evaluate the performance of BEST-REACTIVITY algorithm applied to the MDO and SDO models to find the single optimal placement that minimize the average reaction time. For the SDO model, we evaluate the performance on 58 middle-size ISP networks, again taken from [27], for 3 controllers. We compare BEST-REACTIVITY with a random sampling of the controller placement. For the sake of space, we do not report the detailed results. The ratio between the reaction time obtained with BEST-REACTIVITY is on average $2.1\times$ (at most $4.4\times$) smaller than the random sampling, given the same number of considered placements. Our results

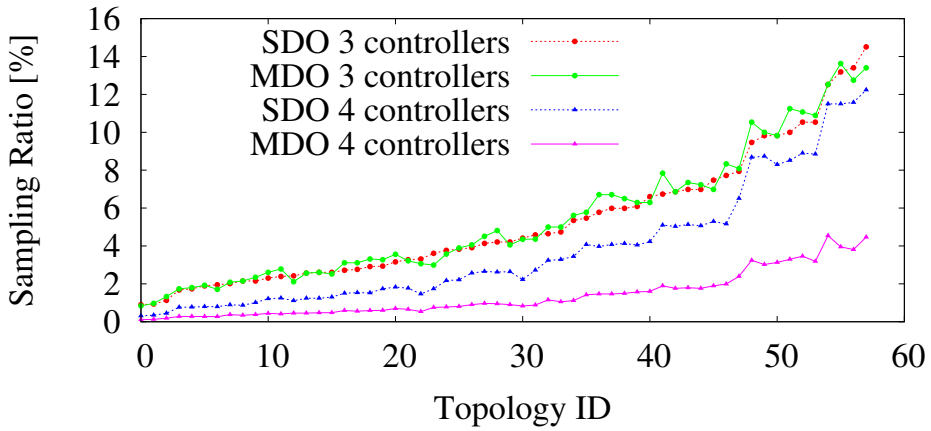

Fig. 2.21: Complexity for BEST-REACTIVITY to achieve a reactivity $\leq 30\%$ larger than the minimum reaction time.

prove the effectiveness of adopting DECREASE-CTR-CTR-DELAY to decrease the reaction time of the candidate for the optimal placement.

We also compare BEST-REACTIVITY with the solution obtained through Gurobi [8] implementing the optimization model described in Sec. 2.6.1, under the same 58 ISP topologies considered above. According to standard methodology, we evaluated the approximation ratio with respect to the optimal algorithm, i.e. the ratio between the average reaction time obtained by BEST-REACTIVITY and the one obtained by Gurobi solver. Fig. 2.21 shows the number of iterations, measured as sampling ratio with respect to the exhaustive search, to achieve 1.3 approximation ratio, i.e. to obtain a reaction time which is $\leq 30\%$ larger than the minimum one obtained by Gurobi. We investigate both SDO and MDO scenarios, for 3 and 4 controllers. Our results show that the actual solution space to consider depends on the actual topology, and a reasonable good solution (i.e. $\leq 30\%$ error with respect to the optimal one) can be obtained by sampling around 1-10% of the solution space. Interestingly, increasing the number of controllers improves the efficiency of BEST-REACTIVITY thus making us more confident about the robustness of the proposed approach.

2.8 Related works

The work in [65] emphasizes the importance of the network state consistency, and indicates that inconsistent network states degrade the performance of network ap-

plications. Thus, [65] motivates our work, since we devise the controller placement problem to target small Ctr-Ctr delays in the MDO model, thus improving the resilience of the network to possible state inconsistencies between controllers.

Many works address the controller placement problem in SDN, but with slightly different objectives. The works [84, 55, 74] target fault tolerance, whereas [96, 83] aim at balancing the loads on the controllers. [30] strives for network resource minimization in mobile cellular networks. [54, 33, 94] focus on the optimal controller placement by considering only the minimization of Sw-Ctr delays (average or maximum). Differently from us, they neglect completely the interaction among controllers and thus the Ctr-Ctr delays. In the case of SDO model, [54, 33, 94] neglect the relevant role of the data owner. In the case of MDO, we have shown in Sec. 2.4.3 that by relaxing the minimum switch-to-controller delay target, it is possible to significantly reduce the Ctr-Ctr delays and improve the convergence to a consistent network state.

Interestingly, [63] addresses the controller placement problem considering a wide combination of metrics: average/maximal Sw-Ctr and Ctr-Ctr delays, the level of load balancing, and the number of isolated switches in case of network partitioning. The latter metric is tailored to a resilient controller placement. Notably, [63] does not consider the combined effect of Sw-Ctr and Ctr-Ctr delays in the reactivity perceived at the switches as in our work. From the algorithmic point of view, [63] adopts exhaustive search to find the optimal Pareto controller placements for small size networks, exactly as EXA-PLACE (Sec. 2.3.1), and proposes a simulated annealing approach for large networks. Differently from EVO-PLACE, this approach requires careful tuning of many parameters and obtains the solution with a number of iterations around 1-10% of the sample space, similar to the results obtained by EVO-PLACE. Finally, [86] provides a general mathematical framework to compute the optimal controller placement, under generic cost functions, but it neglects the role of Ctr-Ctr delays.

2.9 Summary

We consider a distributed architecture of SDN controllers, with an in-band control plane. We investigate the performance issues related to the placement of the controllers across the network nodes. Different from previous work, we highlight the

importance of the coordination among the controllers to synchronize their shared data structures. We distinguish two possible models for the shared data structures: the single (SDO) and the multiple (MDO) data-ownership models, which are both currently implemented in state-of-the-art controllers for strong consistent and eventual consistent data structures.

As first step, we study the optimal controller placement problem by considering all the communications occurring in the control plane. We investigate the Pareto frontier that characterizes all the possible tradeoffs between switch-controller delays and inter-controller delays. For the SDO model, we prove the crucial role of the placement of the data owner controller, since the reactivity perceived by any switch depends heavily on the interaction between the controllers and the leader controller. We compute the Pareto frontier for many realistic ISP topologies, based on an optimal algorithm (EXA-PLACE). To overcome the limited scalability of the optimal solver for large networks, we devise an evolutionary algorithm (EVO-PLACE).

As second step, we focus on the reaction time as perceived by the switches, which provides a single target to optimize, and depends on the network application running on the controller. We devise some analytical formulas for the two data-ownership models models, and validate experimentally their accuracy in a SDWAN. Thanks to these formulas, we formalize an ILP problem to minimize the average reaction and we devise an approximated algorithm, still based on evolutionary algorithm (BEST-REACTIVITY), able to scale to large networks, whose performance is compared against the optimal solver.

We believe that our investigation provides a solid methodology not only to place the controllers but also to design the network supporting the control plane in large networks, as in the scenario of large SDN networks or SDWANs.

Chapter 3

On-the-fly Traffic Classification and Control with a Stateful SDN approach

The novel "stateful" approach in Software Defined Networking (SDN) provides programmable processing capabilities within the switches to reduce the interaction with the SDN controller, thus improves both the scalability and the performance of the network. In our work, we consider specifically a stateful extension of OpenFlow that was recently proposed, called OpenState, that allows to program simple state machines in almost-standard OpenFlow switches.

We consider a reactive traffic control application that reacts to the traffic flows which are identified in real-time by a generic traffic classification engine. We devise an architecture in which an OpenState-enabled switch sends the minimum number of packets to the traffic classifier, in order to minimize the load on the classifier and improve the scalability of the approach. We design two stateful approaches to minimize the memory occupancy in the flow tables of the switches. Finally, we validate experimentally our solutions and estimate the required memory for the flow tables. The content of this chapter is also reported by [100].

3.1 Integrating an SDN controller with a traffic classification engine

We consider a scenario in which the incoming traffic is mirrored to a traffic classifier (TC), so that traffic flows are eventually identified. Based on the classification outcome, a traffic control application operates a specific policy on the flow, e.g. re-routing the traffic to a different port, tagging the traffic or dropping it.

3.1.1 On-the-fly traffic classification

We address the scenario in which traffic is classified in real-time based on the actual sequence of packets switched across the network. To identify a flow, only the initial sequence of packets of a flow are required by the TC. Let C_p be the minimum number of packets to identify protocol p. Each TC engine is characterized by different values of C_p, depending on the adopted technology and the level of accuracy. For some protocols, the basic classification based on transport layer information (e.g. TCP/UDP ports), allows an immediate identification and thus $C_p = 1$. For more advanced identifications, this number can be larger and may depend on the required accuracy.

The traffic control application is supposed to react only to a specific set, denoted as $\mathscr{A}$, of protocols. Let C be the minimum number of packets sufficient to identify any protocol in $\mathscr{A}$ for a given TC engine. Thus,

$$C = \min_{p \in A} C_p$$

We are now discussing some technologies for the TC engine. One technique to classify traffic on-the-fly is Deep Packet Inspection (DPI), which can be implemented in different ways. The first approach is denoted as pattern-matching DPI (aka, pure DPI), which identifies the flow by matching the whole layer-7 payload with a set of predefined signatures. All the signatures are collected in a dictionary defining a set of classification rules, and then checked against the current packet payload until either a match is found or all the signatures have been tested. The second approach is based on Finite State Machines (FSM-DPI) which are used to verify that message exchanges are conform to the expected protocol behavior. For example, for the

OpenDPI engine [19], $C_p \in [1,25]$ to achieve the maximum accuracy [58]; a smaller accuracy can be achieved for $C_p \in [1,10]$. The third approach is based on Behavioral Classifiers (BC) which leverage some statistical properties of the traffic. For instance, the distribution of packet sizes or of inter-arrival times may allow to identify the application generating the traffic. This approach avoids the payload inspection and is not affected by encryption mechanism. However, statistical estimators usually require a large number of packets per flow to achieve a good accuracy.

The definition of the signatures and matching rules implemented by above approaches can be either achieved manually, i.e., by studying or reverse-engineering the protocols to classify, or, automatically, i.e., by adopting Machine Learning (ML). ML learns the peculiar features of given traffic flows, and provides the knowledge to classify on-the-fly the traffic [52]. The disadvantage is that the results depend mostly on the training data which should be up-to-date and accurate, and may not be as accurate as other techniques. As example, the ML-based classification scheme proposed in [35] is able to detect the application generating the traffic with at most 5 packets, thus $C_p \in [1,5]$.

Each classifier offers a different tradeoff between accuracy and processing speed. Our investigation is independent from the actual classification engine, provided that only the first C packets are required for the flow identification, given the specific set $\mathscr{A}$ of protocols for which the traffic control application is supposed to react. Whenever the engine is not able to classify a flow after receiving C packets, it is clearly useless and counterproductive to keep injecting packets of the same flow to the TC. Thus, we aim at designing solutions satisfying the following design constraint: *no more than C packets of the same flow are sent from the switch to the traffic classifier*.

3.1.2 Basic integrated approach

A standard approach to integrate an SDN controller with the TC which satisfies the above design constraint is shown in Fig. 3.1. The traffic control application instructs the switch to forward all the packets of a new flow to the controller through legacy OF packet-out messages. Then the SDN controller provides a copy of each received packet to the traffic control application (usually through the northbound interface of the controller), which does a *countdown* from C to 0 for each flow, by

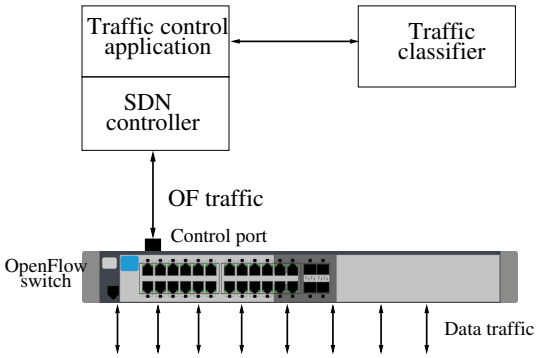

Fig. 3.1: Basic approach for integrating traffic classification with an SDN controller

.

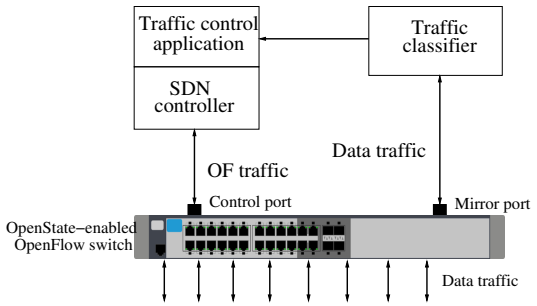

Fig. 3.2: Proposed stateful approach for integrating traffic classification with an SDN controller

counting the number of packets for each flow. As soon as the TC classifies the flow, the network application stops the countdown and programs the switch based on the given traffic control policy. In the case the countdown reaches 0, i.e. the number of forwarded packets is C, then the traffic control stops sending packets to the TC (since it is useless to identify any protocol in $\mathscr{A}$) and programs the switch (typically, through flow-mod messages) to stop sending the packets to the controller. This approach poses severe scalability issues caused by the exchange of packets and control messages between (i) the switch, (ii) the controller/application and (iii) the TC, and the consequent communication and processing overhead.

An alternative solution to reduce the communication overhead from the switch to the controller is to install a forwarding rule within the switch that mirrors all the traffic with a time limit. This approach does not require a stateful extension to the OF switch, but requires a hard-timeout which is difficult to tune, since the minimum time corresponding to C packets depends on the actual traffic arrival process, which is usually unknown. Notably, hard-timeouts, differently from soft-timeouts, expire

after a predefined time, independently from the actual traffic arrival process and have been available since OF version 1.0. Nevertheless, an hard-timeout can be safely set assuming a worst-case behavior of the flow, but in typical cases this would imply a much larger number of mirrored packets than C, with a useless waste of resources.

In the following section, we present our approach which overcomes the limitations of the techniques described above.

3.2 Stateful SDN approach for traffic classification

Adopting a stateful approach in OF switch allows for a very efficient mirroring of the first C packets of a flow. Indeed, the switch can *autonomously* mirror the first C packets of each flow to the TC engine, without the involvement of the traffic control application or the SDN controller in the countdown process.

We consider OpenState [36] as an enabling technology for stateful SDN. OpenState supports Mealy Machine as abstraction for extended finite-state machine (XFSM), which enables programmability of a stateful data plane in a quite flexible way, with switches whose hardware is (almost) the same as standard OF switches. OpenState is implemented with two main tables. The *state table* maps each active flow to its current state (i.e., an integer value). Instead the *XFSM table* is an extension of a standard OF *flow table* that maps a match field to an action. Indeed, in the XFSM table the match field includes also a possible value for the current state, and the action can also be updated on the fly. In such a way, we can implement state machines in which packet arrival events trigger transitions and states evolve as described by the XFSM table. Notably, we can implement XFSM tables directly in TCAM memories, as currently done for flow tables in commercial OF switches. In the following, to remark their common nature, we will refer to the XFSM table as flow table.

Leveraging this technology, we can adopt the approach described as follows. Whenever a packet arrives, the state table identifies the current state of the corresponding flow, the switch processor accesses the flow table, and based on the match fields on the packet header and on the current state, it takes an action on the data

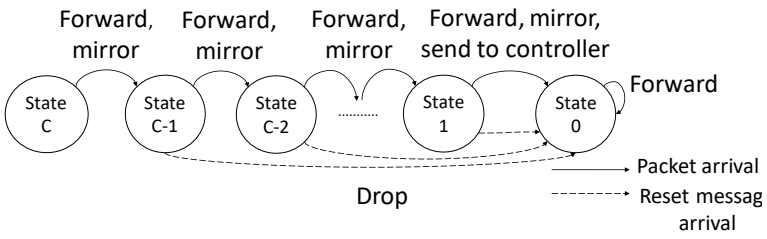

Fig. 3.3: Finite state machine programmed in the OpenState switch for each new flow. The transitions are triggered by packet arrivals and associated with the actions to apply on the packet.

plane (e.g., forward, drop) and updates the state of the flow in the state table.[1] The implementation details of OpenState are available in [36].

To exploit the stateful approach provided by OpenState, we propose the architecture shown in Fig. 3.2, based on OF switches supporting OpenState extension.

We program the switch to run the finite state machine (FSM) illustrated in Fig. 3.3 for each new flow, in order to operate the countdown from C to 0 within the switch, and not in the traffic control application as in the basic solution described in Sec. 3.1.2. Each packet arrival triggers a transition in the FSM. Whenever a new packet arrives, the switch decrements the state, forwards the packet to the required destination port, and in the meanwhile mirrors it to the TC. When the countdown reaches zero, the switch disables the mirror operation. The transitions triggered by a "reset" message are not required for the basic countdown process, and will be discussed in Sec. 3.2.3.

In the following, we propose two approaches to implement the state machine mechanism described above. Our goal then is to minimize the number of flow entries and the size of the tables used by such approaches.

3.2.1 Simple CountDown (SCD) scheme

The first approach to implement the state machine in Fig.3.3 is denoted as SCD (Simple CountDown). The main idea is to maintain the state equal to the current countdown value and the flow table describing the update of the state based on

[1]Notably, OpenState is flexible and provides more operations than those described. For instance, it allows to define different "lookup" and "update" scopes to access and update the state table.

Match fields		Action	
Header	Current state	Data plane	New state
flow-id$_1$	C	forward and mirror	$C-1$
flow-id$_1$	$C-1$	forward and mirror	$C-2$
...	...	...	...
flow-id$_1$	1	forward, (send to controller) and mirror	0
flow-id$_1$	0	forward	0
*	default	send to controller	-

Table 3.1: Flow table for SCD approach when the first packet of flow "flow-id$_1$" is received

Match fields		Action	
Header	Current state	Data plane	New state
flow-id$_1$	0	forward	0
*	default	send to controller	-

Table 3.2: Flow table for SCD approach after the countdown ends

the flow identifier and the current state. The behavior of the proposed scheme is described in Fig. 3.4, according to which the switch mirrors only the first C packets to the TC.

Table 3.1 shows the flow entries installed in the flow table when the first packet of a new flow reaches the controller (through a packet-in message). We assume that the flow is identified by a specific matching rule denoted as "flow-id$_1$" (e.g. IP source/destination and TCP ports). In the first C states (from C to 1) the switch mirrors the traffic to the TC (through the mirror port), while it forwards the traffic according to the standard routing. The final state of the countdown is 0 that means that the switch has mirrored C packets to the TC, and must disable the mirroring for the corresponding flow. By construction, the total number of entries is $C+1$ for each flow, thus the total memory occupancy of the table is $F(C+1)$ entries, if F is the concurrent number of flows traversing the switch. After the installation of the entries in the state table, the switch processes new packets belonging to the same flow locally without the intervention of the controller.

In addition to the flow rules to update the countdown process, we add the standard default rule for any new flow, which must be sent to the controller (through a packet-in message). In addition, we also add some basic rules (not shown here for briefness)

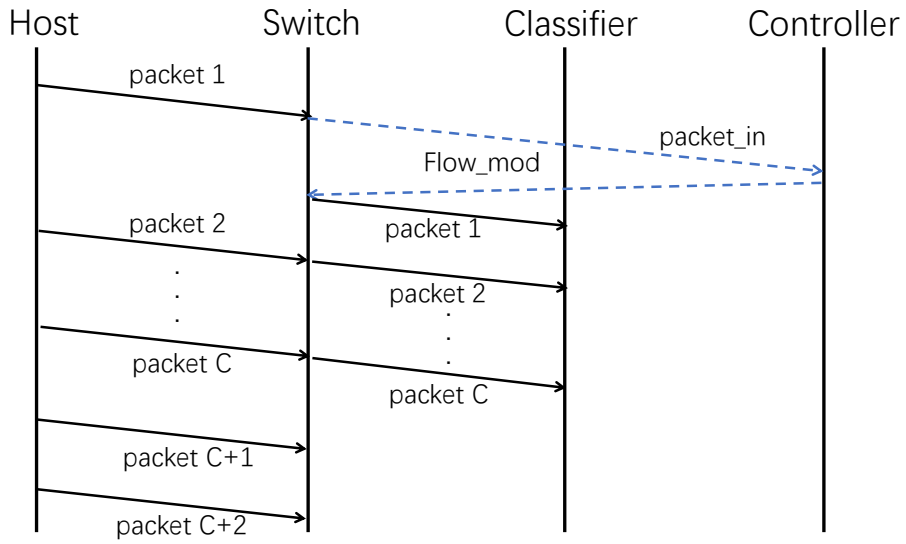

Fig. 3.4: Exchange of messages for SCD and CCD schemes

to manage ARP packets and avoid sending them to the TC. In the following, we will not consider the impact of this couple of rules on the size of the flow tables.

In order to minimize the memory occupancy, we devise an optional *memory purging scheme* to delete the C entries associated to a flow as the countdown ends. Indeed, when the flow state becomes 0 (i.e. the countdown has terminated), the packet is sent also to the controller through a packet-in message (not shown in Fig. 3.4). Since in OpenState a packet-in carries also the current value of the state, the controller can understand that the countdown has terminated and issues an OF delete message to remove all the entries regarding the corresponding flow and add a entry with the final forwarding rule to apply. At the end, the flow table corresponding to a specific flow is shown in Table 3.2. The proposed purging scheme is complementary to the standard idle timeouts of the entries in the flow tables. The main advantage of our approach is that it does not require a careful setting of the timeouts, which depend on some worst-case arrival pattern for a flow, which is practically very difficult to know in advance.

Match fields		Action	
Header	Current state	Data plane	New state
flow-id$_1$	*	forward and goto table 2	*
*	default	send to controller	-

Table 3.3: OpenState flow table FT1 for CCD approach

Match fields		Action	
Header	State	Data plane	New state
*	C	mirror	$C-1$
*	$C-1$	mirror	$C-2$
...	...	...	...
*	1	mirror	0
*	0	-	0

Table 3.4: OpenState flow table FT2 for CCD approach

3.2.2 Compact CountDown (CCD) scheme

The second approach we propose aims at reducing the size of the flow tables, and thus we denote it as Compact CountDown (CCD). The approach exploits a cascade of two flow tables, as shown in Tables 3.3 and 3.4. The entries corresponding to each flow in both tables are installed when the first packet of a flow reaches the controller, as in SCD scheme. The first table (FT1) programs the required forwarding action and imposes that the second table (FT2) must be processed, in cascade, independently from the actual state. Instead, FT2 stores the countdown values, independently from the flow.

In this way, we achieve the same behavior as SCD (shown in Fig. 3.4) but with a reduced number of state entries. We have 1 entry in FT1 for each flow and $C+1$ entries in FT2 for all the flows. Thus, for F concurrent flows, the total number of entries is $F+C+1$.

Differently from SCD, the memory purging scheme at the end of the countdown is not necessary in CCD since only one entry for each flow is stored in the flow tables and must be kept for the entire life of the flow. Thus, in addition to the reduced memory occupancy, SCD does not require the switch to interact with the controller for the purging, with a beneficial effect of load reduction on the controller.

3.2.3 Countdown interruption

As soon as the TC identifies the flow, it is useless to keep mirroring the traffic to the TC. Thus, we propose a scheme to interrupt the countdown in order to minimize the load on the TC. We devise an in-band signaling scheme based on a "reset" message sent directly from TC to the switch with the same flow identifier of the just classified flow. Fig. 3.3 shows how this message is integrated in the countdown FSM and Fig. 3.5 shows the network behavior due to the interruption. The state machine

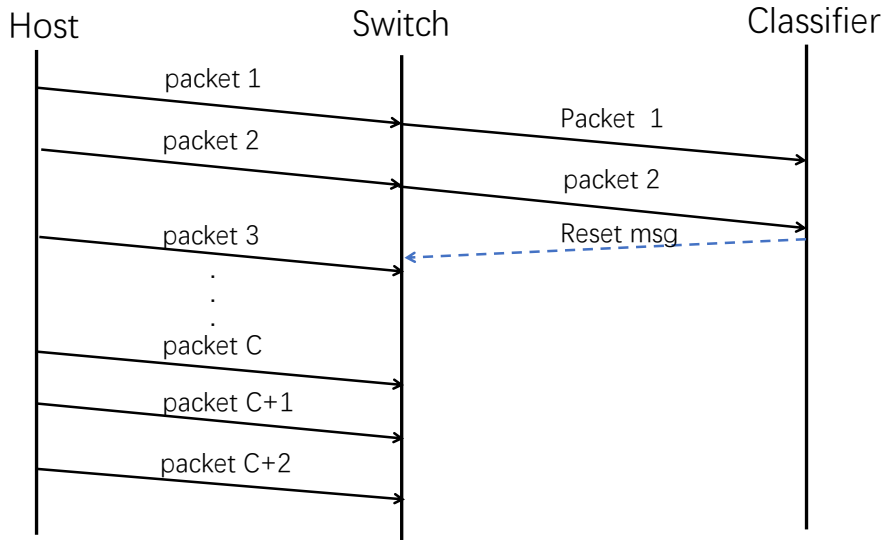

Fig. 3.5: Protocol behavior for an interruption

changes in a way that anytime the TC sends a packet to the switch on the mirror port, the new state of the flow becomes 0, i.e., the countdown is interrupted. This behavior is obtained by adding one flow entry as shown in Table 3.5. The priority of such entry is set higher than the other entries to be sure that it works properly.

For SCD the interruption mechanism is integrated with the proposed memory purging scheme in order to minimize the memory occupancy.

3.2.4 Comparison of approaches

Table 3.6 summarizes the differences between our two proposed approaches and the number of installed entries for F concurrent flows, according to the discussion above

Match fields			Action	
Header	Input port	Current state	Data plane	New state
...	...	...	...	...
flow-id$_1$	mirror port	*	drop	0

Table 3.5: Additional entry in SCD and CCD to interrupt the countdown

	SCD	CCD
Number of flow tables	1	2
Memory purging	Yes	Not needed
Countdown interruption	Yes	Yes
Flow entries during countdown	$F(C+1)$	$F+C+1$
Flow entries after countdown	F	$F+C+1$

Table 3.6: Comparison between the two approaches for F concurrent flows

(we have omitted the default rule for unknown flows and the rules related to ARP packets). From both tables, CCD appears the most convenient because of its mild growth in the memory occupancy. In Sec. 5.4 we also evaluate the actual occupancy in bytes.

3.3 Validation and Experimental Evaluation

We validate the behavior of both SCD and CCD approaches in the testing Ubuntu 14.04 VM provided in OpenState website [1]. The VM provides a modified version of Mininet 2.2.1 with OpenState-enabled switches and Ryu controller is available to issue OpenState-specific flow-mod commands and configure the state machine internal to the switch.

We develope a Python script running in Ryu that programs the switch according to either SCD or CCD schemes. To verify the correct behavior of our implementation for both schemes, we configured the topology of Fig. 3.6 in Mininet, with 2 hosts, the controller, the TC module and one switch. We run `tcpdump` in all the hosts to capture the detailed exchange of packets destined to the hosts and to verify the correct behavior of our implementation for different values of C.

We perform the validation as follows. We program the OpenState FSM to send the traffic arriving from host 1 to host 2 and to mirror the first C packets to the host corresponding to the traffic classifier, using SCD or CCD approach. We generate

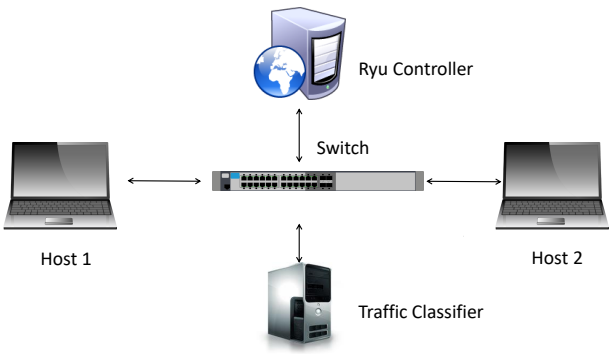

Fig. 3.6: Topology for testing

Approach		Flow table occupancy [bytes]
SCD	min	$18F$
	max	$22F(C+1)+17F$
CCD	min	$17F+14C+12$
	max	$34F+14C+12$

Table 3.7: Total memory occupancy for F concurrent flows and countdown from C

the ICMP packets from host 1 to host 2 with the `ping` command to verify that only the first C packets are forwarded correctly also to TC. Then, by sending an appropriate flow-mod packet from the SDN controller, we verify that the memory purging scheme works as expected in SCD. Finally, to verify the correct behavior of the countdown interruption, explained in Sec. 3.2.3, we run `netcat` command in the TC host to generate a packet with the same flow-id (at IP level) of the flow from host 1 to host 2 and thus interrupt the countdown.

We evaluate experimentally the actual memory occupancy in bytes for the two approaches. Notably, it is not immediate to infer the memory occupancy because of the different match fields in SCD and CCD schemes. Furthermore, our estimation is based on the memory occupancy of the flow tables in Mininet with the OpenState extension, which provides a reasonable approximation of the memory required for a real hardware implementation based on TCAM.

To evaluate the actual size of the flow tables, we exploited the standard OpenFlow "FLOW_STATS" request and reply messages. The reply contains a field representing the length in bytes of the entries installed in the tables of the switch. This length comprises the match fields (including the current state) and the actions (including the

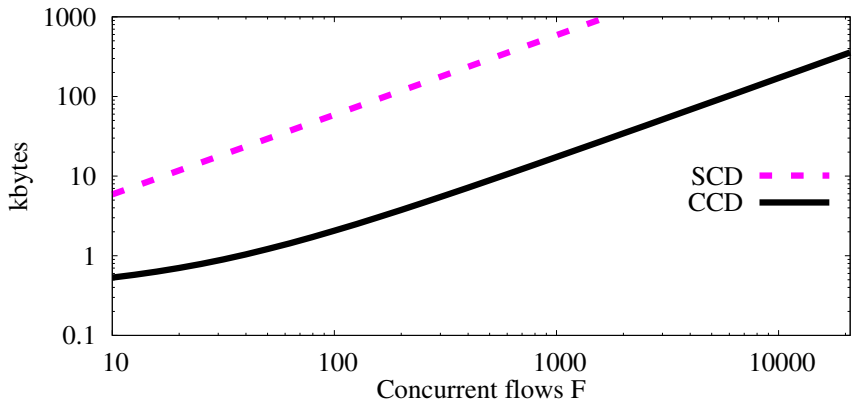

Fig. 3.7: Total memory occupancy in the flow tables

new state) that must be applied over the packets. We sample the table sizes after each installation of the XFSM for a new flow, for different values of F and C and obtain the empirical formulas in Table 3.7. We show two bounds for SCD and CCD. "SCD max" provides an upper bound on the occupancy, due to the $C+1$ rules installed for each flow at the beginning plus the rule to manage the countdown interruption. "SCD min" provides instead a lower bound on the occupancy, due to the final entry left in the table after the memory purging operation. Both bounds are strict, and we expect that the actual occupancy is between the two bounds. For CCD the two bound differs only of 17 bytes, equivalent to the size of the interruption entry.

Fig. 3.7 shows the total occupancy in function of F and for two values of C. All the curves show the expected growing proportional to F. SCD in the worst case requires around $1.5C$ times the amount of memory than CCD, but in the best case it can also outperform CCD, when the number of flows is less than 50. This is due to the fixed overhead of CCD to store the flow table FT2.

Fig. 3.7 allows to assess the maximum scalability of each approach in a real setting. If we consider a maximum size for the TCAM equal to 250 kbytes, which is a typical value according to [22], SCD is able to sustain around 2,500 concurrent flows, whereas CCD can sustain more than 80,000 concurrent flows, thus with a gain of almost two orders of magnitude.

3.3.1 Experimental comparison with standard OpenFlow switches

We tested experimentally the traffic monitoring architecture in Fig. 3.2 in two different scenarios: the first one using the CCD stateful approach implemented with OpenState, and the second one using a standard approach (without countdown) implemented in a standard OF switch. By comparing the actual data traffic from the switch to the traffic classifier, we will evaluate quantitatively the gain in terms of scalability of our proposed stateful approach.

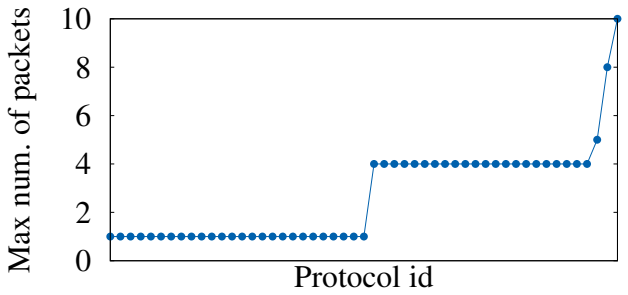

Fig. 3.8: Maximum number of packets needed to identify the protocols

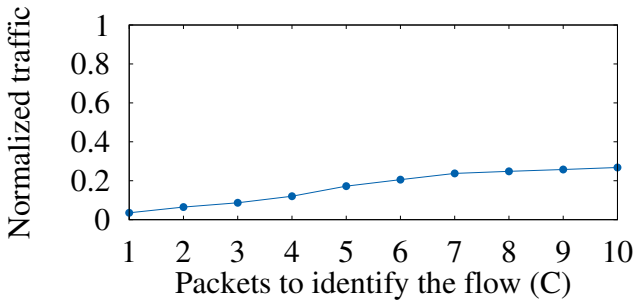

Fig. 3.9: Average ratio of traffic received by nDPI for a stateful CCD approach, with respect to a standard OF approach

In our testbed, we connected directly Ryu controller to the traffic classifier through a TCP socket. The traffic classifier was implemented in a standalone module by adapting the open-source code of nDPI [12], which allows to identify a large set of applications analyzing the IP packets. The classifier was programmed to send a message to the SDN controller whenever a flow was identified. The network application running on the SDN controller was designed to stop mirroring the traffic of a flow anytime the flow was identified by nDPI and to steer such flow to another port of the switch.

We created a real-traffic trace by capturing the traffic of a single user for 53 minutes while accessing multiple services on the Internet (e.g. web browsing, video streaming, VoIP, cloud services, etc). The total number of packets in the trace was 645,720, with a total number of flows equal to 14,807; the average generated traffic was around 200 packets/s. Fig. 3.8 shows the maximum number of packets required to identify the flows in our specific trace, obtained by feeding directly the real-traffic trace to nDPI. In our experiment, 51 different kinds of flows were identified, in particular all the DNS queries were identified with just the one packet (as expected), whereas between 4 and 10 packets were needed to identify all the remaining flows.

We used `tcpreplay` to feed the trace of the traffic from a host to the switch. We measured through a packet sniffer the data traffic sent from the switch to the classifier for identification. We varied C to evaluate the effect of the countdown procedure in CCD on the data traffic sent by the switch to the traffic classifier.

Fig. 3.9 shows the traffic received by the traffic classifier using a stateful CCD approach and a standard approach in an OpenFlow switch. In the latter case, the average traffic sent to the classifier is always 13.37 packets/s since C does not have any effect. This traffic is much lower than the average offered load to the switch (around 200 packets/s), because of the truncated mirroring of any new identified flow. Notably, in a scenario with multiple users to monitor, we would expect an increase in the traffic to the classifier proportional to the number of active users.

As we can see from Fig. 3.9, the CCD approach reduces always the load of classifier between 73% (for $C = 10$) and 96% (for $C = 1$), with respect to the standard OF approach, thanks to the countdown interruption mechanism described in Sec. 3.2.3. This allows to increase the number of users monitored by the same traffic classifier, e.g. by a factor of 28 when $C = 1$ and by a factor of 3.7 when $C = 10$.

3.4 Summary

In this chapter, we considered an SDN traffic control application that reacts in real-time to the traffic, based on the analysis of a traffic classifier, towards which the traffic is mirrored. To improve the scalability of the approach, and, thus, of the overall system, we addressed the problem of minimizing the interaction between the

main three players of the system, i.e., the switch, the SDN controller and the traffic classifier.

We leveraged OpenState, a novel extension of OpenFlow, to implement a state machine directly in the switches in order to mirror just the minimum number of packets to the packet classifier. We designed two solutions based on OpenState, aimed at minimizing the total amount of memory required for the flow tables. Finally, we evaluated experimentally the actual memory in bytes to assess precisely the maximum scalability of each solution, given the size of the TCAM memory through which the flow tables are typically implemented in OpenFlow switches.

Our results show that our proposed CCD solution outperforms SCD solution in terms of memory footprint by almost two orders of magnitude, thus allowing us to execute on-the-fly traffic classification, while guaranteeing a satisfactory scalability degree.

Chapter 4

Misbehaving SDN controllers

The logical centralized approach in the control of SDN networks allows an unprecedented level of programmability in the network, but also implies the vulnerability in the case of misbehavior of the controller, due for example to software bugs, hardware problems or hacker attacks. In our work we propose to exploit the diversity offered by multiple controllers to manage the network switches and detect misbehaviors whenever one controller issues different OpenFlow instructions for the data plane with respect to the others. We design a behavioral checker, denoted as BeCheck, that acts as a transparent relay in the interaction between the network switches and the controllers. We propose and investigate different policies to relay the messages and to detect the controller misbehavior. We implement and validate our approach in a simple testbed, showing the possible tradeoff between detection reliability and controller reactivity perceived at the switches.The results and obervations in this chapter have also been reported in [101].

This chapter is organized as follows. Sec. 4.1 discusses the previous work. Sec. 4.2 introduces the architecture of the proposed solution and describes different detection policies. Sec. 4.3 describes the implementation of the prototype and the testbed adopted for the experimental validation. The experimental results highlight the different tradeoffs between detection reliability and the controller reactivity as perceived by the network switches. Finally, in Sec. 4.4 we draw our conclusions.

4.1 Related works

Many recent works [32, 59, 51, 69, 87, 40] focused on checking software bugs and verifying the correctness of SDN control plane. In particular, [32] proposed a system that deductively verifies if an SDN program is correct on all feasible network topologies. Similarly, [51] implemented a dynamic analyzer to identify software bugs and prevent a network from concurrent violations. The work [69] developed a distributed model checker to verify some security properties related to the network state. In [87] a troubleshooting technique was implemented that can automatically detect the minimal sequence of random inputs responsible for bugs in the SDN control plane. The work in [40] presented a model checking technique with symbolic execution to test the applications running on top of SDN controllers. On the other hand, the studies [61, 70] focused on the security aspects of SDN. In particular, [61] analyzed the threats related to the SDN paradigm and advocated the design of secure and dependable SDN controllers. Finally, [70] presented a first prototype of secure SDN controller designed to deal with malicious SDN administrators. However, all the above works adopted complex analysis processes, such as model checking, which are very time-consuming and difficult to run in real-time. Unlike them, our work detects misbehaving controllers in real-time by comparing the behavior of independent controllers, in a transparent way for both the controllers and the network switches.

More similar to our work, [59] proposed to check some network-wide invariants in real time (e.g. loop free routing) by deploying a software layer between the control and data plane and dynamically inspecting each flow installation rules. It was based on a single controller, whereas BeCheck operates with multiple controllers. The advantage of our approach is that BeCheck runs completely oblivious of the network application. Finally, [67] aims at validating the behaviors of distributed SDN controllers. Similar to us, a consensus process is employed to determine the correct actions and detect misbehaving controllers. However, the detection is based on replicating the network events on the other controllers within the cluster, thus requires some modification of the internal management of the cluster. Instead, BeCheck is completely transparent with respect to the controllers, which do not require any modification.

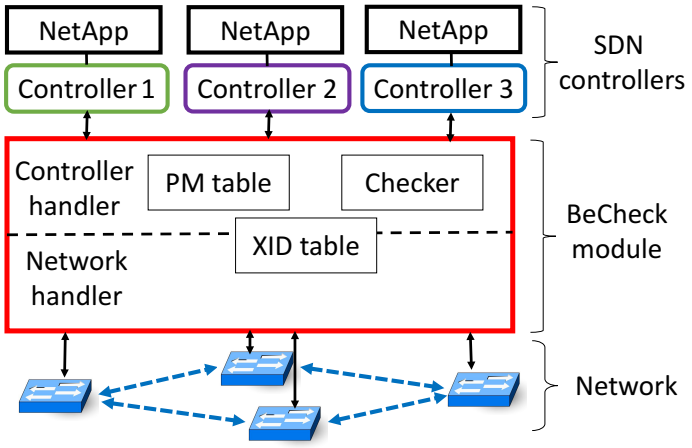

Fig. 4.1: The architecture of BeCheck with $C = 3$ controllers

4.2 Architecture of our behavioral checker

Fig. 4.1 shows the general architecture of our behavioral checker, which exploits the diversity offered by multiple controllers running in lockstep the same network application. Let C be the number of controllers, with $C \geq 2$. All C controllers operate on the same network, and each controller is responsible for managing all the switches, i.e. acting as master for OpenFlow (OF) switches. The controllers run the same network application; thus, if they all behave correctly, they will show the same sequence of messages sent to the switches. Thus, misbehaviors can be detected by comparing the messages received by the controllers and check if inconsistent messages are sent to the switches. Our behavioral checking module, denoted as *BeCheck*, is responsible to digest the OF instructions arriving from the controllers, check their consistency and replicate a copy of the instructions to the destined switch. At the same time, BeCheck replicates all the network events to all the controllers, to ensure that the network applications proceed in lockstep with coherent states. BeCheck is connected to the switches as it was their master controller, but actually its role is just to detect misbehaviors and relay the messages in both directions between switches and controllers. Thus, BeCheck does not substitute the controller and does not take any decision for the data plane, being completely oblivious of the specific network application running on the controllers.

It is worth to highlight that our approach is different from a cluster of distributed controllers, where each controller acts as master only for a subset of switches. Distributed controllers indeed are aimed at improving reliability and scalability of

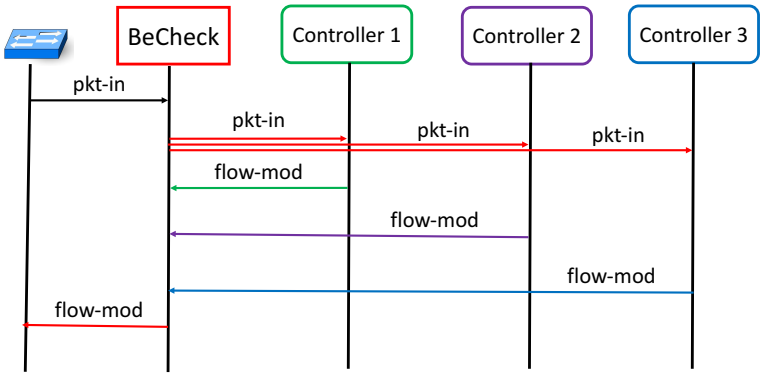

Fig. 4.2: Example of BeCheck behavior with $C = 3$ controllers

the SDN control plane. Instead, in our case we exploit the diversity provided by multiple controllers, in order to detect misbehaviors. It would be also possible to extend our approach to cluster of controllers, i.e. each controller in Fig. 4.1 might be a different cluster of distributed controllers, but we have left this extension for future work.

BeCheck acts as a relay for the OF messages between the controllers and the switches. Not all the OF messages are actually processed by BeCheck, since some of them are not meaningful to detect misbehaviors in the data plane. In OF standard [20], three kinds of messages are defined: (i) controller-to-switch messages, which are initiated by the controller and allow to manage the switches; (ii) asynchronous messages, which are initiated by the switch and allow to update the controller about some local events (e.g. changes in the switch state); (iii) symmetric messages, which are initiated by either the switch or the controller and sent without solicitation. In our case, we consider just the main OF messages dictating the forwarding behavior of the data plane in the network, i.e. `pkt-in` (which is an asynchronous message), `flow-mod` (which is a controller-to-switch message) and `pkt-out` (which is a controller-to-switch message). All the other kinds of messages are instead transparently forwarded by BeCheck.

As example of BeCheck behavior, consider the scenario of a reactive forwarding network application. Assume now that a switch receives a packet from the network, generates a `pkt-in` message and its master controller reacts by sending to the

switch a `flow-mod` message. As shown in Fig. 4.2, the switch sends the `pkt-in` to BeCheck, which replicates it to all the three controllers. Now each controller reacts to the `pkt-in` independently, and sends a `flow-mod` message to BeCheck. If now BeCheck finds out that all the 3 messages received by the controllers are coherent, then it sends to the switch one copy of the `flow-mod`. Otherwise, in the case of inconsistency, i.e. one controller behaving incorrectly, BeCheck can react in different ways: e.g., generates an alarm towards the controllers, or disables the interaction with the misbehaving controller. The details on how to manage misbehaving controllers are outside the scope of the current paper.

There are three critical challenges about implementing BeCheck. First, to guarantee the communication between BeCheck and all the controllers, the transaction id ("xid") for each single controller must be managed correctly. Since BeCheck module interacts with multiple controllers which may react to a network event using different xids, OF reply messages with wrong xids from the network can result in connection refusal by the controllers. BeCheck implements a scheme to map the xids used to interact with the controller and the xids to interact with the switches. Second, to detect misbehaving controllers, the action/s associated with each OF controller-to-switch message (i.e. `pkt-out`, `flow-mod`) must be recorded and the messages are checked in terms of coherence between the different controllers. BeCheck records the messages from the controllers throughout a Pending Message (PM) table. Third, the policy according to which the behavior check is preformed as well as the timing to send the required OF messages to the switches have different impacts on the detection reliability and on the reactivity of BeCheck. BeCheck implements one of the three detection policies described in Sec. 4.2.3.

The software architecture of BeCheck, as shown in Fig 4.1, mainly consists of two components: *controller handler* and *network handler*. The controller handler manages the message exchange with the controllers whereas the network handler manages the message exchange with the network switches. The *XID table* is shared between the two components and it implements a one-to-many table mapping for the xids used for the network switches and for the controllers. The *PM table* buffers the instructions of the OF messages received from the controllers and it is updated by the controller handler. The Checker submodule is in charge of running the policy to verify the behavioral consistency among the messages received by the controllers and to detect possible misbehaviors.

OF message type	Datapath-id	Switch xid	Controllers xid
`pkt-out`	...:0A	1001	{101, 201, 301}
`flow-mod`	...:FB	1002	{102, 202, 302}

Table 4.1: Example of XID table for 3 controllers

OF message type	Datapath-id	Action	Controller bitmask *B*
`pkt-out`	...:0A	output 1	100
`flow-mod`	...:FB	output 3	110

Table 4.2: Example of PM table for 3 controllers

4.2.1 Network handler

The network handler plays the role of forwarding OF messages between the controller handler and the network. Whenever an OF message is received from the network, the network handler adds the corresponding xid field in the XID table and replicates the message to all the controllers, storing the corresponding values of xids used for the interaction with the controllers. The XID mapping is required since multiple controllers may send OF messages with different xids to react the same event, and OF reply messages with unexpected xids are discarded by the controllers.

The XID table consists of 4 fields, mainly based on the 8-byte header of OF. The first two fields are aimed at identifying the message, while the last two store the xids mapping. The message is identified through the OF 1-byte "message type" and the corresponding switch is identified through the 8 byte "datapath-id" (based on the switch MAC address). Finally, the xid is detected by the 4-byte xid present in the OF header of the messages exchanged with the network and with the controllers. An example of XID table is shown in Table 4.1.

4.2.2 Controller handler and detection policy

The controller handler is the central part of BeCheck. It maintains the connections with the SDN controllers and forwards OF messages between the controllers and the network handler. It also collects and inspects the messages from the controllers' side, so as to detect possible misbehaviors. All the OF messages, except `flow-mod` and `pkt-out`, are sent directly to the switches in a seamless fashion.

If all the controllers behave identically, BeCheck expects exactly the same OF message received from all the three controllers in some random order. A message is considered "pending" if it has been only confirmed by a subset of controllers. The handler operates on the Pending Message (PM) table, based on the OF messages (either `pkt-out` or `flow-mod`) received from the controller.

The *detection policy* is based on the full consensus on the messages received by the controllers and works as follows. Only when a message is confirmed by all the controllers, then it is considered correct and removed from the PM table. Otherwise, after a fixed timeout, an entry is removed and a misbehavior event is generated, since the corresponding command has not been confirmed by all the controllers.

The PM table contains a message identifier identical to the one in XID table, based on the message type and the datapath-id. Now the particular action associated to such message is stored in the table and a controller bitmask $B = [b_i]_{i=1}^{C}$ is updated to keep track of the controllers that sent the message; the ith bit is defined as:

$$b_i = \begin{cases} 1 & \text{if the message was received from controller } i \\ 0 & \text{else} \end{cases}$$

By construction, $\sum_{i=1}^{C} b_i$ gives the number of controllers from which the message from received. Whenever $\sum_{i=1}^{C} b_i = C$ (i.e. the same message has been received from all the C controller), then the corresponding entry is removed from the PM table. A sample PM is shown in Table 4.2, which refers to the two messages of Table 4.1. According to it, the `pkt-out` message has been already confirmed by controller 1, whereas the `flow-mod` message by controllers 1 and 2.

4.2.3 Forwarding policy

The Checker submodule in Fig. 4.1 reacts on changes in the PM table and triggers two kinds of events: (i) the transmission to the switch of an OF message, (ii) the misbehavior detection. We define reaction time as the latency experienced by the controllers to react to a new event in the network (e.g. `pkt-in` due to a new flow) as perceived by the network switches. E.g., in the toy example of a reactive forwarding application, the reaction time is the interval of time between the generation of the `pkt-in` for a new flow and the reception of the `flow-mod`/`pkt-out` at the switch,

generated by BeCheck. We propose three different forwarding policies, each of them with a distinct trade-off between detection reliability and reactivity:

- **Consensus** policy (CO) sends the OF message to the switch only when the message has been received by all the controllers, i.e. only when $\sum_{i=1}^{C} b_i = C$. This approach introduces the largest reaction time, since it depends on the slowest controller, but it is the most reliable policy since it is able to detect immediately misbehavior of just 1 controller and sends to the switches only fully correct messages. In the case of a misbehavior, BeCheck will not relay the messages from the controllers to the switches, since they are unconfirmed by all the controllers, and this results into a network outage.

- **First Response** policy (FR) sends the OF message to the switch just after the first message has been received by any controller, i.e. as soon as $\sum_{i=1}^{C} b_i = 1$. In the case of a misbehavior, BeCheck will keep relay the messages until the timeout of the corresponding entries in the PM table expires and the detection occurs. This approach introduces the smallest reaction time, due to the fastest controller, but it is the least reliable policy since incorrect messages may be sent to the switches (e.g. when the fastest controller is misbehaving) and the misbehavior detection takes longer.

- **Majority** policy (MA) sends the OF message to the switch just when the message has been received by the majority of the controllers, i.e. when $\sum_{i=1}^{C} b_i > C/2$. The behavior of this approach is intermediate in terms of reaction time and reliability with respect to CO and FR policies.

The performance of these three policies are evaluated in the following section. We argue that BeCheck is not susceptible to software bugs mainly due to its intrinsic simplicity.

4.3 Validation and Experimental Evaluation

We evaluate the performance of BeCheck and compare the different forwarding policies running in the Checker submodule. For easy reference, we introduce the following notation to identify the variants of our approach: "BeCheck-X(C)" where

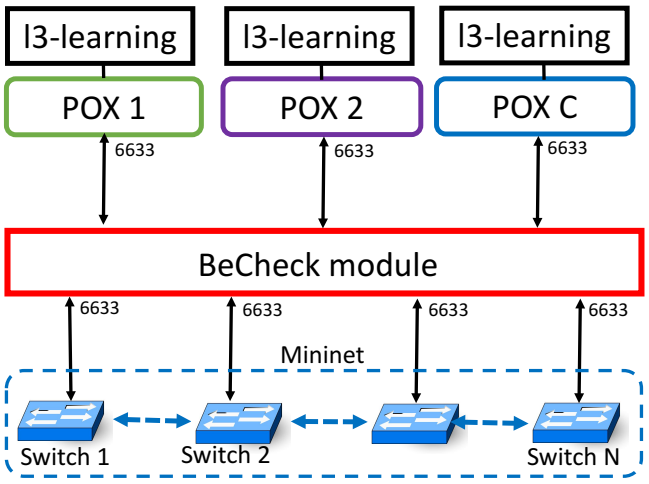

Fig. 4.3: Experimental setup

X is either CO, FR or MA depending on the forwarding policy, and C is the number of controllers.

We implement BeCheck module using OpenFlowJ [21] and NIO.2 libraries [14]; the final source code is around 1500 lines. The experiments are performed in the scenario shown in Fig. 4.3, based on C controllers and N switches in the network. We run BeCheck instance directly in a server, together with the controller instances and the network, emulated with Mininet [92]. The switches communicates with BeCheck through port 6633, thus BeCheck appears as a classic master controller. BeCheck, in turn, is connected to each controller through their predefined port 6633.

We evaluate the time-average occupancy of the size of the PM table, since its small size is crucial for the scalability of the proposed approach, in particular due to misbehaving controllers. We do not report the size of the XID table since its occupancy (never larger than PM table) is not affected by possible misbehaving controllers.

We perform our tests with C independent instances of POX controller [23], each of them running the default `l3-learning` application for reactive forwarding. We show the results for $C \in \{1,2,3,4\}$. Note that for $C = 1$ the forwarding policy does not have any effect, since acts just as a message relay, and this case is used as term of comparison for the evaluation of the computation overhead of the bare BeCheck application (e.g. due to the socket management for the transmission and reception of

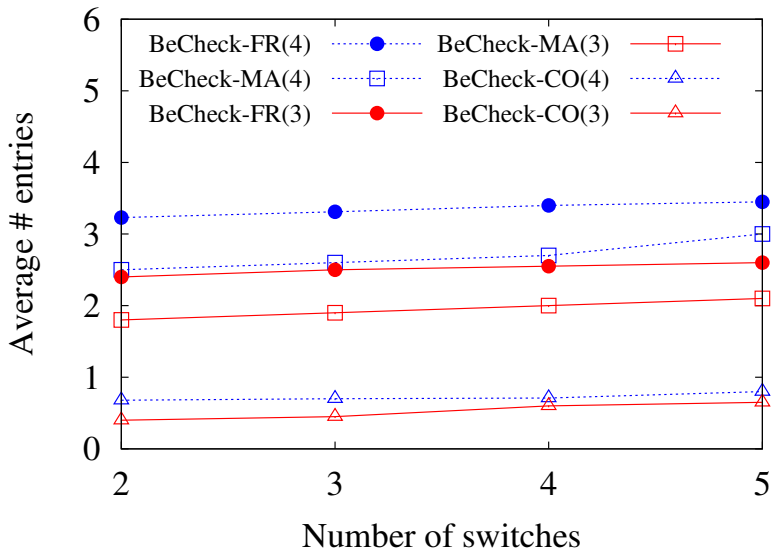

Fig. 4.4: Average size of the PM table

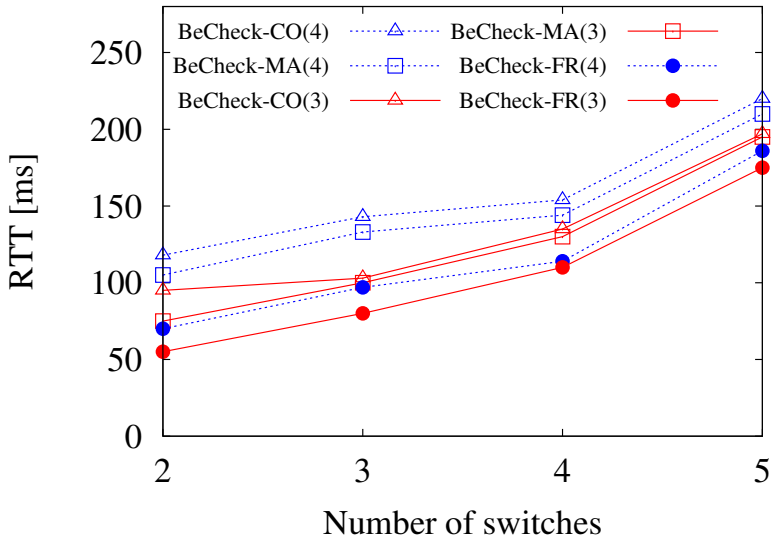

Fig. 4.5: Response time obtained with ping command

OF messages). In all our experiments, the network topology is linear, connecting N switches, with $N \in \{1, \ldots, 10\}$.

We also evaluate the reactivity perceived by the switches by running the `ping -c 1` command at the terminal connected at the first switch, generating one single

ICMP packet towards the Nth switch, and recording the corresponding Round Trip Time (RTT) for the first ICMP request/reply exchange in the switch.

Fig. 4.4 shows the table size for $C \in \{3,4\}$ and $N \in \{2,3,4,5\}$, for different forwarding policies. Consensus (CO) policy always presents the minimal average number of entries, whereas First Response (FR) policy shows the largest number of entries. The reason can be easily understood assuming that one controller is much slower than the others to process the `pkt-in` messages received by the network, relayed by BeCheck. Indeed, for CO only when all the OF `flow-mod` messages are received by BeCheck for a target switch, the message is sent to the switch, which forwards the ICMP packet one hop further along the path. Thanks to the deletion policy in PM table, the corresponding entry is deleted and thus the maximum number of entries is one independently from the number of controllers. On the contrary, FR policy sends the `flow-mod` to the destined switch as soon as it receives the message from the fastest controller, without waiting for the slowest controller. Thus each switch can forward the ICMP packet to the next switch in the path, without waiting for the `flow-mod` generated by the slow controller. But this implies that a large number of entries will be present in the table, which will be deleted only when the slowest controller sends the `flow-mod` message. So in a specific time instance, FR policy results in a larger number of entries in the PM table than that of CO, since FR allows PM to contain entries for messages originated from multiple switches. Majority (MA) policy instead, by construction, behaves in an intermediate way with respect to CO and FR.

Our intuitive explanation is corroborated by observing the reaction time perceived at the network switches, as shown in Fig. 4.5. FR achieves always the best reactivity (i.e. the smallest RTT), CO the worst, whereas MA is in the middle of the two. The graph shows also that the RTT increases with respect to the network size, due to the larger number of hops in the network.

We now consider the scenario in which a controller is misbehaving. We modify the `l3-learning` application in one controller to send always the ICMP packet towards an unused port of the switch. Fig. 4.6 shows the occupancy of the PM table. The number of entries for CO is now bounded by two, independently from the number of switches, since CO receives a coherent OF message from N-1 controllers and another one from the misbehaving controller. CO detects at once the inconsistency. The other two policies are also able to detect the misbehavior but, differently from

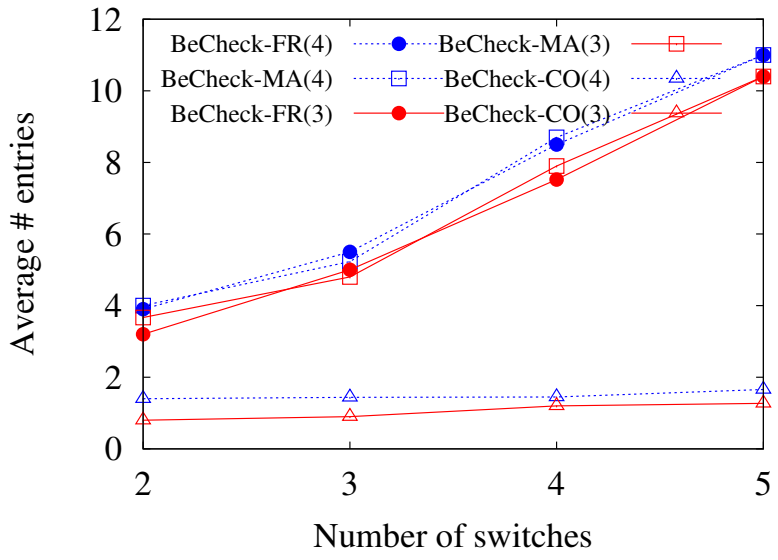

Fig. 4.6: Average size of the PM table in the case of one misbehaving controller

CO, allow the network application run for the non-misbehaving controllers. The main difference between FR and MA is that MA assures always a correct behavior (since the majority of the controllers are behaving correctly in our scenario), whereas FR generates wrong instructions on the data plane whenever the misbehaving controller is the fastest. In all these cases, the occupancy of PM tables grows proportionally to $2N$, since around 2 pending messages are stored for each switch.

To evaluate the actual overhead due to BeCheck, we keep the same configuration as in Fig. 4.3 with all the controller behaving correctly, but we use Cbench to emulate the network with N switches instead of Mininet. Cbench is not able to emulate a real network topology, but just a set of N switches flooding the controller with `pkt-in` messages. We run now `l2-learning` application on POX controller. We record the "throughput" of BeCheck in terms of maximum number of responses per second, as measured by Cbench. The results are obtained by averaging the results for 100 tests, each of them lasting for 1 sec. The results of the tests are shown in Fig. 4.7. As expected, the throughput decreases with increasing number of switches, since in Cbench the switches generate `pkt-in` messages simultaneously. As term of comparison, we report also the scenario, denoted as "No BeCheck", in which BeCheck is not present between the network and the single controller. Thus, from Fig. 4.7 it is clear that the reduction of throughput for larger networks is due to the POX controller. Furthermore, BeCheck(1) (i.e. with just one controller) shows a

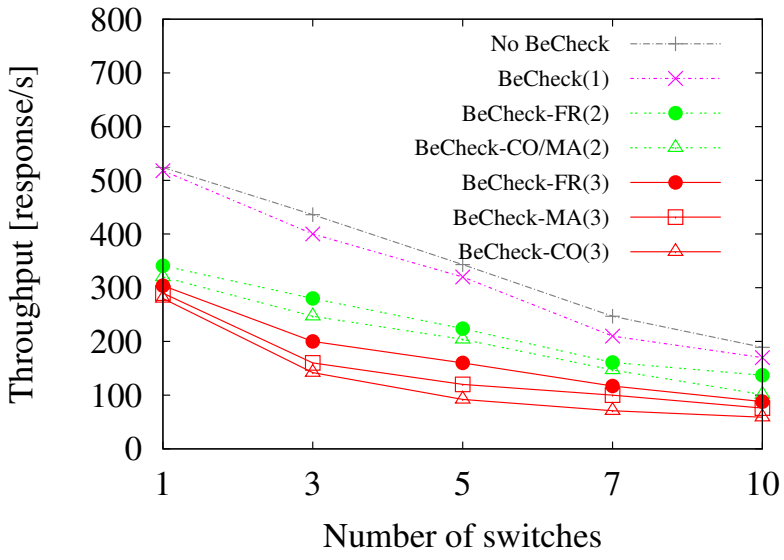

Fig. 4.7: Performance evaluated with Cbench tool

minimum degradation in terms of performance, due to the basic operation of relaying the messages between the controller and the switches. Instead, when the forwarding policies are effective, the performance degrades almost proportionally with respect to the number of controllers. This result is quite expected since each POX controller is able to answer to the flooding requests of Cbench without almost any processing, and thus the main bottleneck becomes the BeCheck module, on which the processing of multiple controllers converges.

4.4 Summary

In this chapter, we propose a behavioral checker, denoted as BeCheck, to detect in real-time possible misbehaviors of SDN controllers. We assume that multiple, independent controllers are running the same network application, and thus their behavior must be coherent. BeCheck is based on a module which relay the OpenFlow traffic between the controllers and the network switches, and compares the OpenFlow instructions to detect possible misbehaviors. We propose a detection policy based on the full consensus on the messages from the controllers. Combined with the detection policy, we consider the effect of different forwarding policies, First Response (FR), Majority (MA) and Consensus (CO), which offer different tradeoffs between the

detection reliability and the latency introduced by the BeCheck module. BeCheck runs completely transparent from the point of view of the network switches and the SDN controllers, and operates obviously from the specific network application running on the SDN controllers.

We implement BeCheck as a standalone module and investigated its performance for the specific case of a basic reactive forwarding application running in multiple instances of POX controller. We show the possible tradeoff between the detection reliability and the controller reactivity as perceived by the switches, for the different detection policies. We also evaluate the throughput degradation due to BeCheck.

Our results, even if preliminary, are promising, and can be extended to consider applications, coherent in terms of behavior, running on completely different controllers, in order to detect misbehaviors in a more reliable way. We leave this extension for future work.

Part II

Integrated Mobile Gaming

Chapter 5

Task Allocation for Integrated Mobile Gaming

In this chapter, we consider a combined Integrated Mobile Gaming (IMG) platform, in which some tasks, i.e., software modules of a game, can be offloaded either to the cloud or to the neighbor mobile nodes. We formalize the optimal offloading problem that minimizes the maximum energy consumption among the mobile nodes, under the constraints of a maximum response time and a limited availability of computation, communication and storage resources. We also propose a heuristic, called TAME, which closely approximates the optimal solution and outperforms other state-of-the-art algorithms under both synthetic and realistic test scenarios. The findings and results of the chapter have also been reported in [98].

The remainder of the chapter is organized as follows. Sec. 5.1 presents the system model of IMG, and describes the available computation and communication network resources as well as the game structure in terms of tasks and their interaction. Sec. 5.2 introduces the optimal energy-efficient task offloading problem, while Sec. 5.3 describes our approximate algorithm TAME. Sec. 5.4 presents the methodology used to investigate the system performance, which is then shown in Sec. 5.5. Finally, Sec. 5.6 discusses related work highlighting the novelty of our contribution, and Sec. 5.7 draws some conclusions.

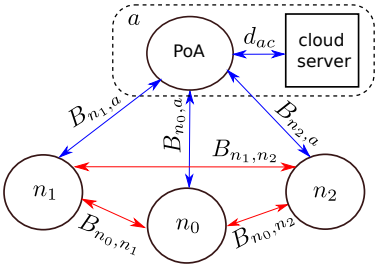

Fig. 5.1: A sample network graph $\mathscr{G}_N$ with three mobile nodes: one player's node n_0 and two neighbor nodes n_1 and n_2.

5.1 System model for IMG

We first introduce the network graph $\mathscr{G}_N$, which describes the network topology and all its computation, communication and storage resources. Then we define the task graph $\mathscr{G}_T$, characterizing the game tasks and their dependency.

5.1.1 Network model

We consider the network topology depicted in Fig. 1.1. Each mobile device can communicate with a Point of Access (PoA), e.g., a Wi-Fi access point or a base station, and with its neighbor mobile nodes through any device-to-device communication technology. For simplicity, we assume that the cloud server is directly connected to the PoA through a wired connection. Let $\mathscr{N}$ be the set of all network nodes, comprising the mobile nodes and the PoA. We denote by n a generic network node; with an abuse of notation, we will use a, with $a \in \mathscr{N}$, to refer to both the PoA and the cloud server. We assume that the network evolves through a sequence of *temporal epochs*, each with duration of the order of tens of seconds or minutes. During each epoch, the communication capacity between pairs of nodes remains constant; indeed, we include in the $\mathscr{N}$ set only those neighbor nodes whose movement with respect to the player's device is negligible during an epoch.

Let $\mathscr{G}_N$ be the *network graph* representing our network during an epoch: each vertex corresponds to a network node while edges represent communication links between nodes in radio visibility. An example of $\mathscr{G}_N$ in the case of a network including three mobile users is depicted in Fig. 5.1, while Table 5.1 summarizes the notation we use to describe the network. Let n_0 be the game player's device. A generic mobile node n is characterized by computation capacity C_n, available

Table 5.1: Network model notation

Symbol	Description
$\mathscr{G}_N$	Network graph
$\mathscr{N}$	Set of all network nodes
m, n	Generic node in the network, $m,n \in \mathscr{N}$
a	PoA and/or cloud server, $a \in \mathscr{N}$
n_0	Player's mobile device
$B_{n,m}$	Throughput from node n to m [bit/s]
$B_{n,a}$	Throughput between node n and the PoA [bit/s]
D_{ac}	Propagation delay between the PoA and the edge node [s]
$E^T_{n,m}$	Energy consumption for node n to transmit data to m [J/bit]
$E^R_{n,m}$	Energy consumption at node m to receive data from n [J/bit]
E^C_n	Processing energy consumption at node n [J/cycle]
C_n	Computation capacity at node n [Hz]
S_n	Storage availability at node n [bit]
E_n	Available energy at node n in a frame period [J]

storage S_n, and available energy E_n. The cloud node a has computation capacity C_a, while its available storage and energy are considered as unbounded. Let $B_{n,m}$ be the throughput between node n and m, and $B_{n,a}$ the throughput between node n and the PoA. We assume that the propagation delay from any mobile node to the PoA is negligible, and that the available bandwidth between the PoA and the cloud server is so high that the only contribution to the communication delay between the PoA and the cloud server is due to the propagation delay, denoted with D_{ac}. Regarding the energy consumption, we define E^C_n as the per-clock-cycle energy cost due to computing at node n. With regard to the transmission from node n to node m, let $E^T_{n,m}$ and $E^R_{n,m}$ be the per-bit energy cost for the transmission and the reception of data, respectively, similarly to the models adopted in [42, 93].

5.1.2 Mobile game model

The software of a game can be partitioned into several tasks. Different levels of granularity for the definition of the tasks can be adopted, i.e., at method level, at object/class level, or at component level (such as artificial intelligence module, collision detection module and so on). In real-time games, the software manages the calls of the tasks within a main event loop, which is responsible to update the entire state of the game, given the players' inputs, and to render the scene. The duration of

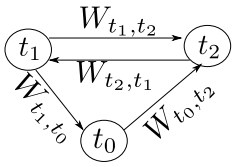

Fig. 5.2: A sample task graph $\mathscr{G}_T$, composed of one local task t_0 and two generic tasks t_1, t_2.

Table 5.2: Game model notation

Symbol	Description
$\mathscr{G}_T$	Task graph
$\mathscr{T}$	Set of all game tasks
$t,\ \tau$	Generic task of a game, $t, \tau \in \mathscr{T}$
t_0	Local task on n_0
F	Frame period duration [s]
s_t	Size of task t (code + state) [bit]
$\mathscr{O}$	Set of all the objects
s_o	Size of object o [bit]
c_t	CPU required by task t [cycles]
$W_{t,\tau}$	Data from task t to task τ [bit]

the event loop is typically bounded by the scene *frame period* (e.g., 33 ms for 30 fps) with duration F.

For the sake of generality, we describe the tasks and their dependencies through a directed graph, called *task graph* and denoted by $\mathscr{G}_T$. Each vertex corresponds to a task, and an edge connects two vertices if there exists a dependency between the corresponding tasks, e.g., the output of the first task is used as input to the second one, or the second task runs only after the first has been completed. Fig. 5.2 shows a sample graph with 3 tasks; in Sec. 5.4.3, we will then discuss a methodology to obtain such task graph in the case of real-world games.

The notation we use for the task graph is summarized in Table 5.2. Let $\mathscr{T}$ be the set of all tasks, assumed to run in real-time. Let t_0 be a special task (possibly comprising a set of specific sub-tasks) that must run locally on the player's device n_0 (e.g., I/O processing, video rendering or decoding). Each task t is characterized by a computation requirement c_t, expressed as number of CPU cycles in a frame period, coherently with standard practice [93, 42]. Let $W_{t,\tau}$ be the average amount of data, measured in bits, sent from task t to task τ during a frame period. Some tasks may need some objects for their execution (e.g., the textures shown in the game, or the

Table 5.3: Decision and auxiliary variables in the optimization problem

Symbol	Description
$X_{t,n}$	1 if t is offloaded to n, 0 else
$\pi_{n,m}^{t,\tau}$	1 if task t on n sends data to task τ on m, 0 else
$\eta_{o,m,n}$	1 if n downloads object o from m, 0 else
$\nu_{n,o}$	1 if some task on n needs object o, 0 else

sounds), which must be stored at the nodes running the tasks. We assume that the cloud has a copy of every object and that, at the beginning of each frame period, both the availability of objects at each node and the set of objects required by a task are known. With an abuse of notation, we say that $o \in n$ if object o is locally available at node n, and $o \in t$ if object o is required by task t.

5.2 Problem formulation

We formulate the problem of optimal energy-aware task offloading, under the system constraints presented in Sec. 5.1. Aim of the problem is to allocate each task to the most suitable node, so that the maximum energy consumption across the mobile nodes, including the player's one, is minimized. This ensures fairness in the energy toll requested to the mobile nodes for the game execution, and it can also be easily adapted to the case where the player's device is supposed to pay a higher energy toll than its neighbor devices.

We agree that an alternative cost function would be the total energy consumption; notably, our models adapts immediately to this modification. But this cost may lead to unfair energy consumption (few nodes very loaded and others idling) that could be incompatible with cooperation incentives. Thus, we preferred to focus only on the proposed cost function.

Input to the optimization problem are the network model, described by $\mathscr{G}_N$, and the game model, described by $\mathscr{G}_T$. The main decision variable is the binary variable $X_{t,n}$, which takes 1 iff task t is offloaded to node n and zero otherwise, with $X_{t,a} = 1$ denoting that the task is allocated to the cloud server. The related decision and auxiliary variables are summarized in Table 5.3. The objective function is given by:

$$\min \max_{n \in \mathscr{N} \setminus \{a\}} \varepsilon_n \tag{5.1}$$

where ε_n is the energy cost on node n (excluding the PoA) due to the game execution, during a given frame period. As detailed in Sec. 5.2.1, ε_n accounts for the energy consumption due to task processing as well as to the communication required for task offloading, retrieval of non-local objects and data exchange between tasks. In addition, our optimization framework accounts for the performance perceived by the user, defined as response delay of the game software. Indeed, a game with high response delay or low frame rate cannot guarantee accurate and fluent game scenes, thus in our optimization problem, we constraint the response time to be below a given threshold. This performance metric is defined in detail in Sec. 5.2.2, while the system constraints under which the optimization problem should be solved are reported in Sec. 5.2.3.

5.2.1 Mobile node energy consumption

The total energy cost for any mobile node n, with $n \in \mathcal{N} \setminus \{a\}$, in the given frame period is the summation of four terms:

$$\varepsilon_n = \varepsilon_n^P + \varepsilon_n^C + \varepsilon_n^M + \varepsilon_n^O \tag{5.2}$$

Each term is defined below.

Task processing energy consumption ε_n^P

The total energy consumption of mobile node n, due to the computation of the tasks that node n hosts, is given by:

$$\varepsilon_n^P = \sum_{t \in \mathcal{T}} X_{t,n} \cdot E_n^C \cdot c_t \tag{5.3}$$

where E_n^C denotes the per-clock-cycle energy cost of node n and c_t is the required number of CPU cycles for task t. Thus, $X_{t,n} \cdot E_n^C \cdot c_t$ is the energy consumption due to t being offloaded to n.

Task communication energy consumption ε_n^C

Each mobile node transmits and receives data from other nodes on behalf of the hosted tasks. The corresponding energy consumption at the generic node n is evaluated as follows:

$$\varepsilon_n^C = \sum_{\substack{t,\tau \in \mathscr{T} \\ t \neq \tau}} \sum_{m \in \mathscr{N} \setminus \{n\}} W_{t,\tau} \cdot (\pi_{n,m}^{t,\tau} \cdot E_{n,m}^T + \pi_{m,n}^{t,\tau} \cdot E_{m,n}^R) \tag{5.4}$$

where we recall that $W_{t,\tau}$ represents the amount of data that task t sends to task τ. The first term within the summation denotes the energy used by node n to transmit to m (i.e., $E_{n,m}^T$), whenever t is running on n and τ on m (i.e., $\pi_{n,m}^{t,\tau} = 1$). Similarly, the second term is the energy used by n, running τ, to receive data from task t running on m (i.e., $E_{m,n}^R$). Notably, we consider also the reception of data from the PoA (i.e., $m = a$).

Task migration energy consumption ε_n^M

In the case of task migration from the player's node n_0 to another node m (possibly including the cloud server a), the energy cost at n_0 due to the task transmission is:

$$\varepsilon_{n_0}^M = \sum_{t \in \mathscr{T}} \sum_{m \in N \setminus \{n_0\}} s_t \cdot X_{t,m} \cdot E_{n_0,m}^T \tag{5.5}$$

where s_t denotes the size of task t. Similarly, the energy cost at a generic destination node n receiving a task from n_0 is:

$$\varepsilon_n^M = \sum_{t \in \mathscr{T}} s_t \cdot X_{t,n} \cdot E_{n_0,n}^R \qquad n \neq n_0, a \tag{5.6}$$

Note that we excluded the energy consumption at the PoA (i.e., $n \neq a$), since the PoA operates without energy limitations.

Object retrieval energy consumption ε_n^O

Since tasks may need to load some objects as input for execution, the nodes hosting the tasks are in charge of downloading the required objects from other nodes if not locally available. As a result, a node may need to transfer its locally available

objects to others, or receive objects from other nodes. Thus, any $n \in \mathcal{N} \setminus \{a\}$ may experience the following energy consumption:

$$\varepsilon_n^O = \sum_{o \in \mathcal{O}} \sum_{m \in \mathcal{N} \setminus \{n\}} s_o \cdot (\eta_{o,m,n} \cdot E_{m,n}^R + \eta_{o,n,m} \cdot E_{n,m}^T) \tag{5.7}$$

where s_o denotes the size of object o and $\eta_{o,m,n}$ indicates whether n needs to download object o from m or not. In particular, the two terms in the above equation represent the total energy cost for node n to, respectively, retrieve objects from others and transmit objects to others.

5.2.2 Response delay

The response delay δ experienced by the player in a given frame period is given by:

$$\delta = \delta^M + \delta^R + \delta^P + \delta^E \tag{5.8}$$

where each term is described below.

Task migration delay δ^M

The tasks that are offloaded from n_0 to other nodes or to the cloud for remote execution, require to be transmitted; thus, they experience some migration latency given by:

$$\delta^M = \sum_{t \in \mathcal{T}} \sum_{n \in \mathcal{N} \setminus \{n_0, a\}} \left(\frac{X_{t,n} \cdot s_t}{B_{n_0,n}} + X_{t,a} \cdot \left(\frac{s_t}{B_{n_0,a}} + D_{ac} \right) \right) \tag{5.9}$$

The first term in (5.9) is the transmission time of task t, of size s_t, from n_0 to another mobile node n, given that the expected throughput between the two nodes is $B_{n_0,n}$. The second term in (5.9) is instead the migration delay from n_0 to the cloud, given the transmission time to the PoA (i.e., $s_t/B_{n_o,a}$) and the propagation delay D_{ac} from the PoA to the cloud server.

Task processing delay δ^P

The processing time of task t, requiring c_t cycles and running on node n with C_n processing capability, is c_t/C_n. Notably, in the case the task runs in the cloud, this turns out to be c_t/C_a. The overall execution time depends on the degree of parallelism allowed to run the tasks. If we assume that all tasks are executed sequentially, the total task processing delay δ_{Seq} is given by:

$$\delta_{\text{Seq}} = \sum_{t \in \mathscr{T}} \sum_{n \in \mathscr{N}} \frac{X_{t,n} \cdot c_t}{C_n}$$

Instead, if all tasks are executed in parallel, the total task delay δ_{Par} is the maximum among all the processing nodes:

$$\delta_{\text{Par}} = \max_{n \in \mathscr{N}} \sum_{t \in \mathscr{T}} \frac{X_{t,n} \cdot c_t}{C_n}$$

We expect that in realistic scenarios some tasks can be executed in parallel and others sequentially. As a result, the actual task processing time is bounded as follows:

$$\delta_{\text{Par}} \leq \delta^P \leq \delta_{\text{Seq}} \tag{5.10}$$

In the following, for a worst case design, we will consider $\delta^P = \delta_{\text{Seq}}$.

Task communication delay δ^E

As shown in Fig. 5.2, tasks may need to exchange data, introducing communication latency. For a worst case design, we assume sequential communications, and thus the task communication delay can be formulated as follows:

$$\delta^E = \sum_{\substack{t,\tau \in \mathscr{T} \\ t \neq \tau}} \left(\sum_{\substack{n,m \in \mathscr{N} \\ n \neq m,a}} \frac{W_{t,\tau} \cdot \pi_{n,m}^{t,\tau}}{B_{n,m}} + \sum_{n \in \mathscr{N} \setminus \{a\}} \left(\pi_{a,n}^{t,\tau} + \pi_{n,a}^{t,\tau} \right) \cdot \left(\frac{W_{t,\tau}}{B_{n,a}} + D_{ac} \right) \right) \tag{5.11}$$

The first term in (5.11) includes the duration of the transmission from task t to task τ occurring from mobile node n to mobile node m, and equal to $W_{t,\tau}/B_{n,m}$, whenever such transmission occurs (i.e., $\pi_{n,m}^{t,\tau} = 1$). The second term in (5.11) considers the communications between the cloud server and a mobile node. It includes the

transmission time $W_{t,\tau}/B_{n,a}$ from/to the PoA and the propagation delay D_{ac} between PoA and the cloud server whenever such communication occurs, either from the mobile node to the cloud server (i.e., $\pi_{n,a}^{t,\tau} = 1$) or vice versa (i.e., $\pi_{a,n}^{t,\tau} = 1$).

Object retrieval delay δ^R

We assume that objects, if needed, are retrieved sequentially, thus the total object retrieval delay is formulated as follows:

$$\delta_R \;=\; \sum_{o\in\mathscr{O}} \sum_{n\in\mathscr{N}\setminus\{a\}} \sum_{m\in\mathscr{N}\setminus\{n,a\}} \left(\frac{\eta_{o,m,n}\cdot s_o}{B_{m,n}} \;+\; \eta_{o,a,n} \;\cdot\; \left(\frac{s_o}{B_{n,a}} \;+\; D_{ac} \right) \right) \tag{5.12}$$

The first term in (5.12) represents the total delay $s_o/B_{m,n}$ for mobile node n to retrieve object o from mobile node m whenever convenient (i.e., $\eta_{o,m,n} = 1$). The second term refers to the download time of o from the cloud, i.e., between the cloud and the PoA (i.e., D_{ac}) and between the PoA and node n (i.e., $s_o/B_{n,a}$), whenever this case happens (i.e., $\eta_{o,a,n} = 1$). We recall that the cloud has a copy of any object, thus objects are never uploaded to the cloud.

5.2.3 Constraints

Given the variables defined above, the IMG system is subject to the constraints listed below.

Maximum response delay

The execution of all tasks related to a given frame period needs to be completed within the frame itself:

$$\delta \leq F \tag{5.13}$$

Task mapping

Each task can only be offloaded to either a mobile node or the cloud, i.e.,

$$\sum_{n\in\mathscr{N}} X_{t,n} = 1 \qquad \forall t \in \mathscr{T} \tag{5.14}$$

Additionally, since t_0 is the local task that must run on node n_0, we force $X_{t_0,n_0} = 1$.

Task communication

The auxiliary binary variable $\pi_{n,m}^{t,\tau}$ is related to the main decision variable $X_{t,n}$ as follows:

$$\pi_{n,m}^{t,\tau} = \begin{cases} X_{t,n} \cdot X_{\tau,m} & \text{if } W_{t,\tau} > 0 \\ 0 & \text{else} \end{cases} \tag{5.15}$$

i.e., a data transfer between tasks t and τ running, respectively, on nodes n and m, occurs only if τ gets as input $t's$ output, and t and τ are actually assigned to the two mobile nodes. In order to obtain linear constraints, we can equivalently express (5.15), when $W_{t,\tau} > 0$, as:

$$\pi_{n,m}^{t,\tau} \geq (X_{t,n} + X_{\tau,m} - 1), \quad \pi_{n,m}^{t,\tau} \leq X_{t,n}, \quad \pi_{n,m}^{t,\tau} \leq X_{\tau,m}$$

Object demand

Let $\nu_{n,o}$ be a binary variable such that $\nu_{n,o} = 1$ iff one or more tasks on node n need object o to be executed. Thus, the value of $\nu_{n,o}$ should be such that:

$$\sum_{t|o \in t} X_{t,n} \leq K \cdot \nu_{n,o} \tag{5.16}$$

$$\sum_{t|o \in t} X_{t,n} \geq \nu_{n,o} \tag{5.17}$$

where K is a large enough constant. Indeed, if at least one task t running on node n needs object o (thus, $X_{t,n} = 1$ and $o \in t$) then the summation across all tasks in (5.16) implies $\nu_{n,o} = 1$; otherwise, (5.17) imposes $\nu_{n,o} = 0$. We remark that (5.17) is superfluous, since the energy minimization will prevent the case $\nu_{n,o} = 1$ to occur whenever task t does not require object o.

Object retrieval

If node n hosts tasks requiring an object that is not locally available, then one copy of the object must be downloaded from other nodes, i.e., for any $o \notin n$:

$$K \cdot \sum_{m \in \mathcal{N} \setminus \{n\}} \eta_{o,m,n} \geq \nu_{n,o} \tag{5.18}$$

An object is downloaded by n from other nodes (i.e., $\sum_{m \in \mathcal{N} \setminus \{n\}} \eta_{o,m,n} = 1$) iff it is not locally available (i.e., $o \notin n$) and it is needed by the tasks running on node n (i.e., $\nu_{n,o} = 1$), as modeled in (5.18). Similarly to (5.17), we remark that $\nu_{n,o}$ will take 0 zero unless (5.18) is satisfied.

Resource capacity

Since mobile devices have limited battery and computation capabilities, they must have enough resources in order to host tasks. For each mobile device, we define four kinds of resource capacity constraints: CPU, bandwidth, storage and energy constraints, as described below.

Node CPU constraint. Each node n (either a mobile node or the cloud) must have enough CPU to satisfy the computation requirements of all the tasks it hosts in the given frame period. Thus, for any $n \in \mathcal{N}$,

$$\sum_{t \in \mathcal{T}} X_{t,n} \cdot c_t \leq F \cdot C_n \tag{5.19}$$

The left side is the summation of CPU cycles required by all tasks running on node n, while the right side is the total number of CPU cycles available at the node.

Node bandwidth constraint. In the given frame period, each node n (either a mobile node or the PoA/cloud) must have enough bandwidth to support the data exchange between the local tasks and the remote tasks running on other nodes. Hence, for any $n, m \in \mathcal{N}$:

$$\sum_{\substack{t,\tau \in \mathcal{T} \\ t \neq \tau}} W_{t,\tau} \cdot (\pi^{t,\tau}_{n,m} + \pi^{t,\tau}_{m,n}) \leq F \cdot B_{n,m} \tag{5.20}$$

The left side of (5.20) is the total amount of data that node n transmits/receives to/from m (i.e., $\pi_{n,m}^{t,\tau}$ and/or $\pi_{m,n}^{t,\tau}$ are equal to 1). The right side of (5.20) is the total amount of data node n can exchange with m, in the given frame period.

Node storage constraint. In a given frame period, each mobile node n has to store the objects needed by its local tasks, i.e., for any $n \in \mathcal{N} \setminus \{a\}$, the total size of stored objects cannot be larger than the available storage at n:

$$\sum_{o \in \mathcal{O}} \nu_{n,o} \cdot s_o \leq S_n \tag{5.21}$$

Note that we implicitly assume non-persistent storage, and we expect that non-needed objects can be deleted by node n at the beginning of a frame period whenever $\nu_{n,o} = 0$ and the storage is full. We stress however that the set of objects stored by a node at the beginning of one frame period is given, and used as input to the optimization problem for the current frame period.

Node energy constraint. The energy consumption ε_n cannot be larger than the available energy E_n of a mobile node n, during the frame period, i.e., for any $n \in \mathcal{N} \setminus \{a\}$:

$$\varepsilon_n \leq E_n \tag{5.22}$$

Recall that the cloud server has CPU and bandwidth limitations only.

5.2.4 Problem complexity

The above ILP formulation involves four kinds of decision variables and six kinds of constraints. It can be shown that the total number of variables grows as $O(|N|^2|\mathcal{O}|, |T|^2|N|^2)$, and the total number of constraints grows as $O(|T||N|, |\mathcal{O}||N|)$. According to [73], for a linear programming problem with v variables and α constraints, the complexity of an ILP solver is $O(2^{2^v}\alpha)$. Thus, the final complexity to solve our optimization problem is:

$$\begin{cases} O(2^{2^{|\mathcal{T}|^2 \cdot |\mathcal{N}|^2}} |\mathcal{T}||\mathcal{N}|) & \text{for } |\mathcal{T}| > |\mathcal{O}| \\ O(2^{2^{|\mathcal{O}| \cdot |\mathcal{N}|^2}} |\mathcal{O}||\mathcal{N}|) & \text{for } |\mathcal{T}| \leq |\mathcal{O}| \end{cases}$$

which underlines that the problem complexity greatly increases with the number of tasks and mobile nodes.

5.3 TAME algorithm

The optimization problem presented in Sec. 5.2 can be solved with ILP solvers in the case of small problem instances, but, due to its high complexity, it cannot scale to large network/game instances. We therefore devise a low-complexity, greedy algorithm, named Task Allocation with Minimum Energy (TAME), which closely approximates the solution obtained with the optimal ILP solver.

At each iteration, TAME assigns a task to the node that minimizes the energy cost. This cost depends on both communication and computation. To minimize the complexity when evaluating the energy costs, TAME identifies the dominant factor (either computation or communication) contributing to the total energy cost, based on all unallocated tasks. The task to offload is chosen according to the dominant factor, while ensuring compatibility with the available resources (computation, communication and storage). In other words, at each iteration TAME adapts its "energy-awareness" to the most relevant energy contribution.

The pseudocode of TAME is presented in Algorithm 5, which takes both the task graph $\mathscr{G}_T$ and the network graph $\mathscr{G}_N$ as input and returns the task allocation $\mathscr{X} = \{X_{t,n}\}$. After initialization, TAME allocates the local tasks t_0 to the player's device n_0 (line 2). Then it considers the remaining tasks iteratively, until all of them have been allocated (lines 4-26). At each iteration, TAME estimates the total energy cost due to computation and communication, for all the unallocated tasks. This allows identifying the major contribution to the energy cost, and, based on that, the optimal task allocation. In more detail, let $\hat{E}^C$ be the energy consumption corresponding to one computation unit, averaged across all the mobile nodes, then ε_{cpu} is the estimated total energy cost due to computation for all unallocated tasks (line 5). Likewise, let $\hat{E}^T$ and $\hat{E}^R$ be the energy consumption due to the transmission and reception, respectively, averaged across all possible pairs of mobile nodes. Then ε_{com} is the estimated total energy cost due to communication between all unallocated tasks (line 6). If the dominant energy contribution is due to computation, the unallocated task with the maximum energy (t^*) is selected (line 8). Otherwise, the dominant energy is due to communication, thus the pair of unallocated tasks with the maximum communication cost (t^* and τ^*) are selected. After having selected the task (or the pair of tasks) to allocate, all the mobile nodes are considered as candidates to host such task/s (line 12-17). When a generic candidate node n is considered, the constraints on the maximum response delay (as detailed in Sec. 5.2.3)

Algorithm 5 TAME algorithm

Require: $\mathscr{G}_T, \mathscr{G}_N$
1: $X_{t,n} = 0, \forall t \in \mathscr{T}, \forall n \in \mathscr{N}$ ▷ Init with no allocated task
2: $X_{t_0,n_0} = 1$ ▷ Allocate local task/s to n_0
3: $\Omega = \mathscr{T} \setminus \{t_0\}$ ▷ Init set of unallocated tasks
4: **while** $\Omega \neq \emptyset$ **do** ▷ For all unallocated tasks
5: $\varepsilon_{\text{cpu}} = \hat{E}^C \cdot (\sum_{t \in \Omega} c_t)$ ▷ Energy due to computation
6: $\varepsilon_{\text{com}} = (\hat{E}^T + \hat{E}^R) \cdot (\sum_{t,\tau \in \Omega, \tau \neq t} W_{t,\tau})$ ▷ Energy due to comm.
7: **if** $\varepsilon_{\text{cpu}} \geq \varepsilon_{\text{com}}$ **then** ▷ Computation energy is dominant
8: $t^* \leftarrow \arg\max_{t \in \Omega} \{c_t\}$ ▷ Task with highest computation requirement
9: $\tau^* = \{\}$ ▷ Task τ^* is undefined
10: **else** ▷ Communication energy is dominant
11: $t^*, \tau^* \leftarrow \arg\max_{t,\tau \in \Omega} \{W_{t,\tau} + W_{\tau,t}\}$ ▷ Task pair with highest communication requirement
12: **for all** $n \in \mathscr{N}$ **do** ▷ Find most energy-convenient node
13: assume t^* (and, possibly, τ^*) is offloaded to n
14: **if** all the constraints are satisfied **then**
15: $\hat{e}_n \leftarrow$ maximum energy cost across the mobile nodes if t^* (and, possibly, τ^*) is offloaded to n
16: **else**
17: $\hat{e}_n \leftarrow \infty$ ▷ n is not suitable
18: $n^* = \arg\min_{n \in \mathscr{N}} \{\hat{e}_n\}$ ▷ Find the node with minimum energy
19: **if** $\varepsilon_{n^*} < \infty$ **then** ▷ A feasible choice exists
20: $X_{t^*,n^*} = 1$ ▷ Offload t^* on n^*
21: $\Omega = \Omega \setminus \{t^*\}$ ▷ t^* will not be further considered
22: **if** $\tau^* \neq \{\}$ **then** ▷ If τ^* is defined
23: $X_{\tau^*,n^*} = 1$ ▷ Offload τ^* on n^*
24: $\Omega = \Omega \setminus \{\tau^*\}$ ▷ t^* will not be further considered
25: **else**
26: **return** ▷ Task offloading is not possible
27: **return** X

and on CPU, bandwidth, storage and energy resources (as detailed in Sec. 5.2.3) are evaluated. If all constraints are satisfied (line 14), then $\hat{e}_n$ records the maximum energy cost among all the nodes, assuming that t^* (and possibly τ^*) is allocated on the node under consideration. Among all nodes for which the allocation is feasible, TAME selects the node n^* for which the cost $\hat{e}_{n^*}$ is minimum (line 18). Finally, t^*, and possibly τ^*, are offloaded to node n^* (lines 20-24).

Note that, since TAME greedily allocates tasks in each iteration, at some point it may happen that the response delay constraint cannot be met under the current partial task allocation thus resulting into an allocation failure. To solve this issue, TAME adopts a worst case prediction approach: when it verifies the response delay constraint (line 14), it assumes that for the future iterations the unallocated tasks

will run in the player's node n_0. All nodes that fail to satisfy this response delay constraint are excluded from further inspection. So doing, TAME minimizes the probability of task offloading failure.

5.4 Performance evaluation methodology

We now introduce the methodology adopted to assess the performance of our scheme and to compare it against state-of-the-art solutions. In particular, Sec. 5.4.1 describes the two algorithms we use as benchmarks for the TAME algorithm, while Sec. 5.4.2 and Sec. 5.4.3 present, respectively, the network scenarios and the network task graphs used in our experiments.

5.4.1 Benchmark schemes

We evaluate the performance of our TAME algorithm against the following approaches:

- BESTFIT considers the tasks in decreasing order of CPU requirements and offloads each task to the node with the minimum available CPU resource, thus consolidating the tasks into the minimum number of mobile nodes such that the system constraints are met. As an example, assume that all mobile nodes have the same CPU capacity: since the local task t_0 is initially allocated to node n_0, all other tasks will be allocated into n_0 until possible. Then BESTFIT will assign tasks to the node with maximum available CPU till its capacity is saturated, and it will proceed in this way till no further allocation is possible. It follows that often BESTFIT does not offload any task.

- GRAPHMERGE [79] is based on the idea of combining tasks with low computation requirements and high communication cost into a "super-task", which is then allocated to the most suitable node. This is equivalent to merging nodes in the task graph, and it has the advantage of nullifying the energy cost due to communication between tasks that fall within the same super-task since they will be co-located in the same physical node.

- OPTIMAL solves optimally the ILP optimization problem formulated in Sec. 5.2.

Table 5.4: Experimental setting for the network model $\mathscr{G}_N$

Parameter	Value
$\lvert\mathscr{N}\rvert$	$\{4,6,8,10,12\}$
$B_{n,a}$	$\{10,20,100\}$ Mbps
$B_{n,m}$	1 Mbps (Bluetooth), 20 Mbps (Wi-Fi Direct)
S_n	$\{16,32,64,128\}$ GB
C_n	0.8 - 2.7 GHz, for $n \neq a$
C_a	3 GHz
E_n	18000 J
s_o	1000 bytes
E_n^C	Randomly chosen from Table 5.5
$E_{n,m}^T$, $E_{n,m}^R$	Randomly chosen from Table 5.6
D_{ac}	$\{1,20,50\}$ ms
$\lvert\mathscr{O}\rvert$	$\{0,10\}$ objects

TAME, BESTFIT and GRAPHMERGE are implemented in Python, while in OPTIMAL the solution is obtained by using the Gurobi solver [8]. We evaluate the *approximation ratio* of each algorithm defined as the ratio of the cost function $\max_{n\in\mathscr{N}\setminus\{a\}} \varepsilon_n$ (as defined in (5.1)) obtained through the algorithm, to that of OPTIMAL. Clearly, by construction, the approximation ratio is always equal or greater than one.

5.4.2 Network scenarios

To generate the network model described by $\mathscr{G}_N$, we consider a network scenario characterized by the parameters listed in Table 5.4. We vary the total number of network nodes $\lvert\mathscr{N}\rvert$ (which we recall it includes the player's node n_0 and the cloud server a) from 4 to 12; this corresponds to a number of neighbor mobile nodes for n_0 varying from 2 to 10. Node n_0 is connected to the neighbor nodes through a Wi-Fi Direct or Bluetooth interface, while it communicates with the PoA through a Wi-Fi interface.

The values of the energy consumption for computation (E_n^C) and communication ($E_{n,m}^T, E_{n,m}^R$) are all derived from real-world mobile processors and wireless modules, as detailed below.

Table 5.5: Energy cost due to computation

Processor name	Max power [W]	Max frequency [GHz]	E_n^C [nJ/cycle]
Exynos 5433	0.22	1.5	0.147
Exynos 7420	1.29	2.1	0.614

Table 5.6: Energy cost due to communication.

Technology	Module name	V [V]	$I_T; I_R$ [mA]	$E^T; E^R$ [nJ/bit]
WiFi-Direct	SPWF01SA	3.3	243.0; 105.0	14.8; 6.4
	QFM-2202	3.3	201.2; 66.7	12.3; 4.1
	QCA6234	3.3	250.0; 69.0	15.3; 4.2
	WGM110	4.8	246.0; 81.0	22.0; 7.2
Bluetooth	RN-42(N)	3.3	30.0; 30.0	99.0; 99.0
	HC-06	3.3	8.0; 8.0	26.4; 26.4
	BT4 BLE	3.3	8.5; 8.5	28.0; 28.0
	BT24	3.3	29.0; 29.0	95.0; 95.0
	SPBT2632	2.5	23.0; 23.0	57.5; 57.5

Computation energy cost E_n^C

We estimate the energy consumption per clock cycle by dividing the nominal maximum power of a mobile processor by its maximum CPU frequency. Specifically, we consider two mobile processors: the Exynos 5433 equipping the Samsung Note 4 [6], and the Exynos 7420 equipping the Samsung Galaxy S6 [7]. The resulting values are shown in Table 5.5.

Communication energy costs $E_{n,m}^T$ and $E_{n,m}^R$

We estimate the energy cost to transmit ($E_{n,m}^T$) and receive ($E_{n,m}^R$) data using the formula $V \cdot I/B$ applied to the considered communication chipset, where V is the voltage supply and I is the current when transmitting/receiving at rate B. We consider different Wi-Fi and Bluetooth chipsets, and we assume $B = 54$ Mbps for Wi-Fi and $B = 1$ Mbps for Bluetooth. Note that the Wi-Fi chipsets support also Wi-Fi Direct since the two technologies share the same physical layer. The results are summarized in Table 5.6 and are derived from the chipset datasheets.

5.4.3 Task graph generation

To describe the game graph $\mathscr{G}_T$, we take two approaches. First, we build synthetic task graphs, which emulate mobile games and allow us to easily vary the system

Table 5.7: Experimental setting for synthetic task graphs

Parameter	Value
$\lvert\mathscr{T}\rvert$	$\{4,6,8,10\}$ tasks
$d_{\mathscr{T}}$	2
c_t	Uniformly distributed in $[10^5, 10^6]$ clock cycles
$W_{t,\tau}$	Exponentially distributed with average 220 bytes
F	$\{33,150\}$ ms

Fig. 5.3: Screenshots of MICshooter (left) and Minecraft clone (right)

parameters. Then we profile two real-world games and create the corresponding task graphs, so as to further verify the algorithms performance in real-world scenarios.

Synthetic task graphs

We generate random task graphs with a given number of tasks $\lvert\mathscr{T}\rvert$ and a given average out-degree of each task $d_{\mathscr{T}}$. The graph generation process starts with a graph of $\lvert\mathscr{T}\rvert$ isolated nodes. Then an ordered pair of tasks (t_1, t_2) is randomly selected and a directed edge from t_1 to t_2 is added to the graph. Such select-and-add operation continues until the number of distinct edges in the graph equals $\lvert\mathscr{T}\rvert \cdot d_{\mathscr{T}}$, so as to achieve the expected average degree. The amount of data exchanged between each pair of tasks is exponentially distributed with 500 bytes as the mean value. The frame period is fixed to either 33 ms (for a frame rate equal to 30 fps), or 150 ms (for a generic real-time application). All the settings for this scenario are reported in Table 5.7.

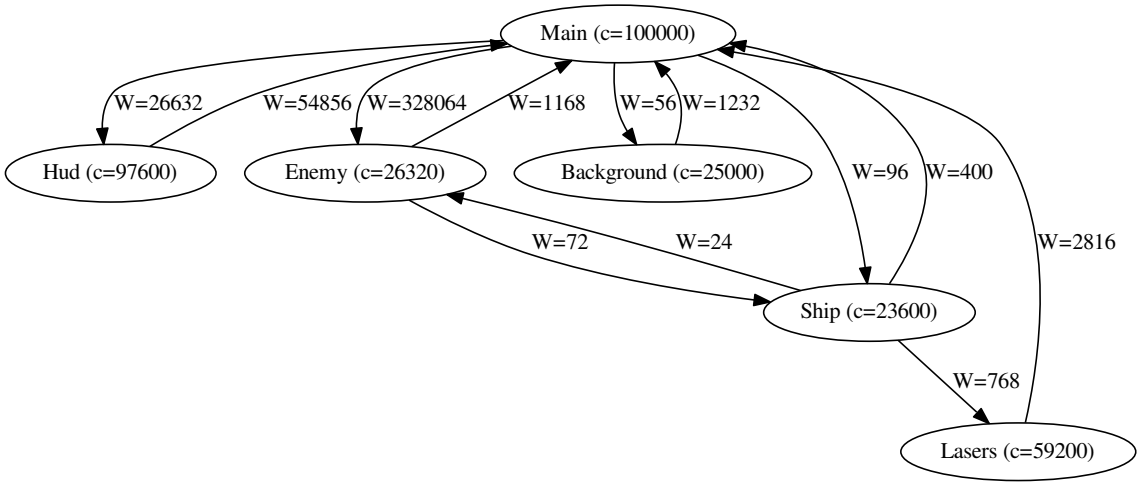

Fig. 5.4: Experimental task graph $\mathscr{G}_T$ for MICshooter: c is expressed in CPU cycles and W in bytes.

Real task graphs

We select two open-source `python` games: MICshooter [10] and Minecraft clone [11], whose screenshots are shown in Fig. 5.3. MICshooter is a classic single-person arcade space shooting game, while Minecraft is a clone of the popular multi-player sandbox game. For both games, we first analyze the source code and partition the game into relevant tasks at the class level. In our experiment, we partition MICshooter into 6 tasks and Minecraft clone into 5 tasks. Then we run the game and get the real task graphs for both games as follows. First, the game call graph can be obtained in real-time using the `pycallgraph` module [24]. We then process the call graph by categorizing the method calls at class level to get the actual task graph $\mathscr{G}_T$. The average CPU requirement c_t of each task t in a frame period can be approximated by the total CPU time provided by `cProfile` [3] divided by the number of frames. The average amount of exchanged data between tasks is obtained by parsing the source code and setting up interception points to measure the size of exchanged data, considering the input variables and the output ones of each call. Notably, to obtain the actual size of the whole data structure referred by a variable, we adopt the python module `asizeof` [2], which recursively measures the referents of a data structure.

Fig. 5.4 and 5.5 depict the resulting task graphs, highlighting the computation requirement (c) and the communication requirement (W) of each task.

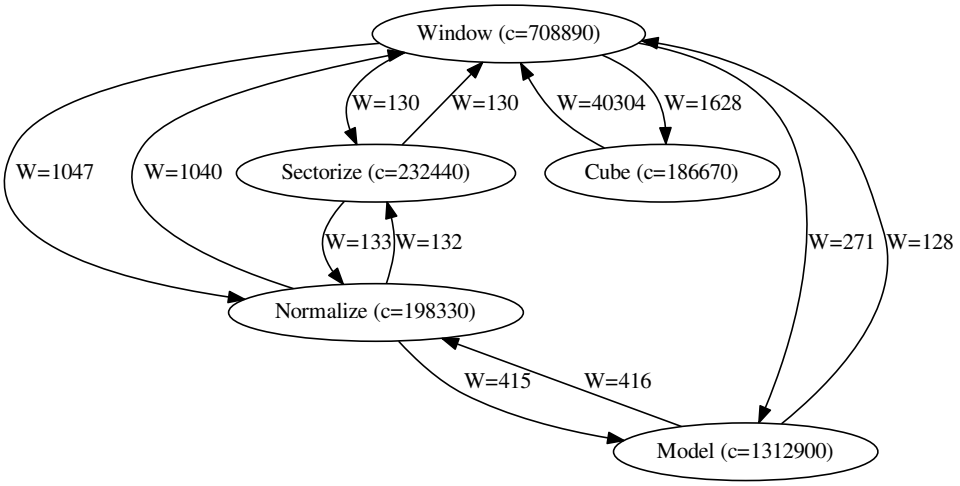

Fig. 5.5: Experimental task graph $\mathscr{G}_T$ for Minecraft clone: c is expressed in CPU cycles and W in bytes.

Table 5.8: Average approximation ratio for Wi-Fi Direct (top) and Bluetooth (bottom) connections, $|\mathscr{N}| \in \{4,6,8,10,12\}$ and $|\mathscr{O}| = 0$

Alg. \ $\lvert\mathscr{T}\rvert$	4	6	8	10
TAME	1.01-1.01	1.01-1.02	1.01-1.02	1.02-1.03
BESTFIT	3.16-3.40	4.59-4.77	5.81-6.04	6.80-6.92
MERGEGRAPH	1.23-1.30	1.45-1.48	1.46-1.51	1.48-1.56

Alg. \ $\lvert\mathscr{T}\rvert$	4	6	8	10
TAME	1.01-1.01	1.01-1.01	1.02-1.04	1.09-1.12
BESTFIT	3.21-3.46	4.52-4.80	5.78-5.93	6.53-6.61
MERGEGRAPH	1.21-1.32	1.54-1.59	2.25-2.33	4.75-5.01

5.5 Numerical results

We first compare the performance of the algorithms for the synthetic task graph scenarios, when the type of connectivity between the network nodes and the number of objects required to run the game vary. Then we show the performance in the case of real-world games.

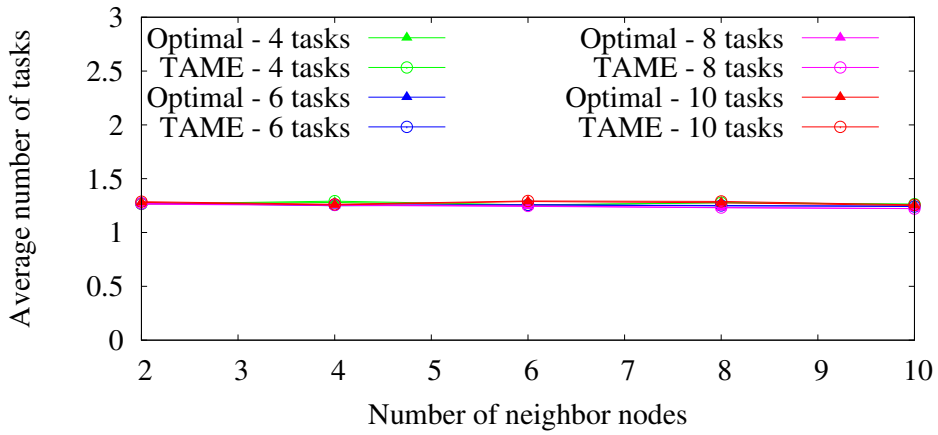

Fig. 5.6: Tasks running locally on player's node, for Wi-Fi Direct communications.

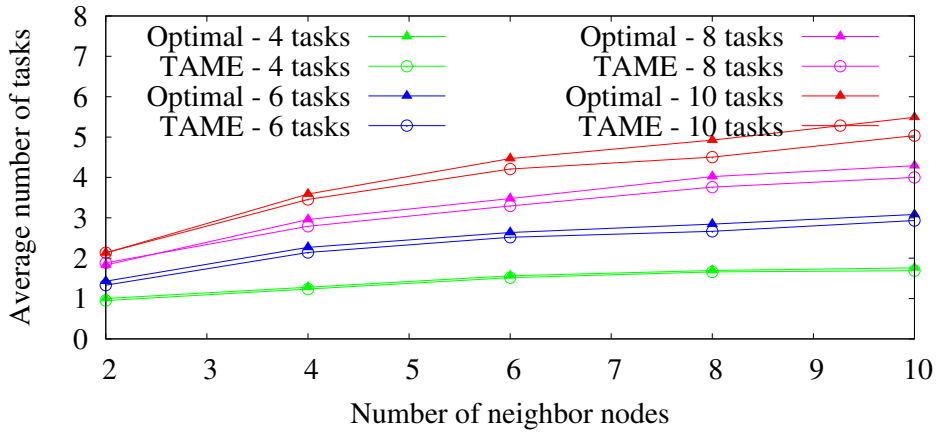

Fig. 5.7: Tasks offloaded to neighbor nodes, for Wi-Fi Direct communications.

5.5.1 Scenario with Wi-Fi Direct communications

Initially, we neglect the process of object retrieval by setting $|\mathscr{O}| = 0$ and assume that the cloud is "close" to the PoA, i.e., we set $D_{ac} = 1$ ms. We fix $F = 33$ ms, corresponding to the common 30 fps refresh rate. Table 5.8(top) reports the average approximation ratio of TAME, BESTFIT and GRAPHMERGE, assuming only Wi-Fi Direct D2D communications. Given a number of tasks $|\mathscr{T}|$ and a number of nodes $|\mathscr{N}|$, we run 1000 different experiments and evaluate the approximation ratio averaged over 1000 instances. In Table 5.8, we fix $|\mathscr{T}|$ and vary $|\mathscr{N}| \in \{4,6,8,10,12\}$; in each cell, for each $|\mathscr{T}|$, we report the minimum and maximum (average) approximation ratio obtained by varying $|\mathscr{N}|$.

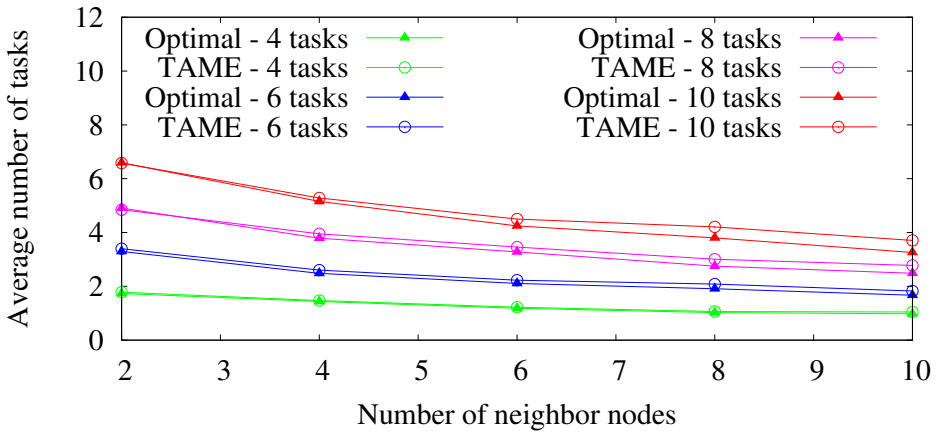

Fig. 5.8: Tasks offloaded to the cloud, for Wi-Fi Direct communications.

Observe that the TAME algorithm is always very close to the optimal solution, with a maximum approximation ratio of 1.03, and always outperforms the other algorithms. On the contrary, BESTFIT gives the worst performance, since it concentrates the tasks in one (n_0) or few nodes thus increasing their energy consumption, while the other algorithms tend to balance the task load across multiple nodes. As expected, this problem is exacerbated as $|\mathscr{T}|$ increases. With regard to GRAPHMERGE, it behaves worse than TAME, because it assumes that the problem is mainly dominated by communication energy costs (which is not the case in this scenario), hence it tends to co-locate different tasks on the same node, regardless their computation energy cost. TAME instead is able to adapt its choices to the dominant energy contribution.

To better understand the behavior of TAME and OPTIMAL, we show how the tasks are distributed across the local node (Fig. 5.6), the neighbor nodes (Fig. 5.7) and the cloud (Fig. 5.8), in the same scenario as for Table 5.8(top). We report the results as functions of the number of neighbor nodes (which is equal to $|\mathscr{N}|-2$). TAME behaves almost identically to OPTIMAL in terms of number of offloaded tasks toward the neighbor nodes and the cloud, and this justifies the approximation ratio very close to 1 reported in Table 5.8(top). In particular, according to Fig. 5.6, the number of local tasks is 1 most of the times, and 2 in all other cases, independently from the number of neighbor nodes, thus TAME is very effective in offloading tasks. By comparing Fig. 5.7 to Fig. 5.8 as the number of neighbor mobile nodes increases, we note that fewer tasks are offloaded to the cloud while more tasks are delegated to the neighbor nodes. This is because a higher number of neighbor mobile nodes provides more choices to offload tasks. Notably, the number of offloaded tasks per

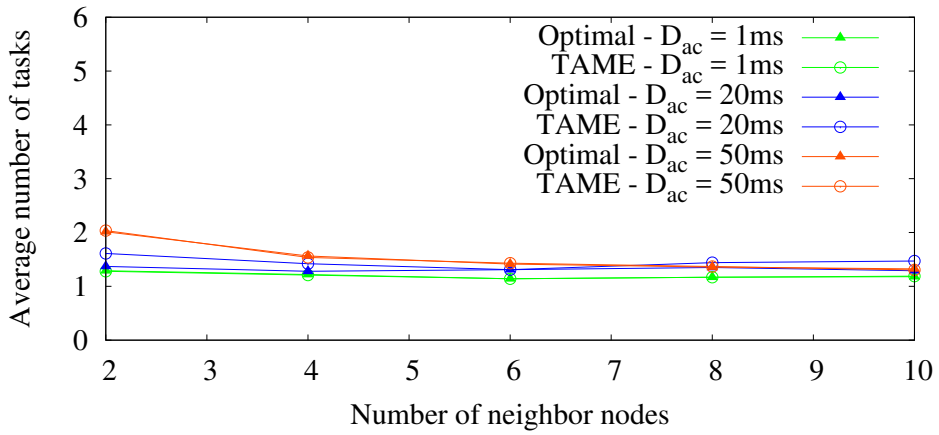

Fig. 5.9: Number of tasks running locally on player's node, for $|\mathscr{T}| = 6$

node is on average at most one, thus TAME tends to distribute equally the load across all neighbor nodes, although some of them do not host any task since offloading is not convenient. All other tasks are offloaded to the cloud.

5.5.2 Scenario with Bluetooth communications

Table 5.8(bottom) shows the average approximation ratio when mobile nodes communicate via Bluetooth, and Wi-Fi is used to communicate with the PoA. In this case too, TAME greatly outperforms the other algorithms. Note that TAME performs a little worse for 10 tasks, compared to the Wi-Fi Direct case in Table 5.8(top). Indeed, Wi-Fi Direct and Bluetooth have quite different link speeds. Since in our experiment the link speed of Wi-Fi Direct (20 Mbps) is 20 times higher than that of Bluetooth (1 Mbps), the response delay constraint becomes more critical in the latter case. As a result, TAME is reluctant to offload tasks to the neighbor nodes and inclined to keep more tasks locally on the player's node n_0. This behavior is exacerbated by a larger number of tasks, since the bandwidth required for the communication between tasks increases. Similarly, GRAPHMERGE performs worse when adopting Bluetooth than in the case of Wi-Fi Direct communications.

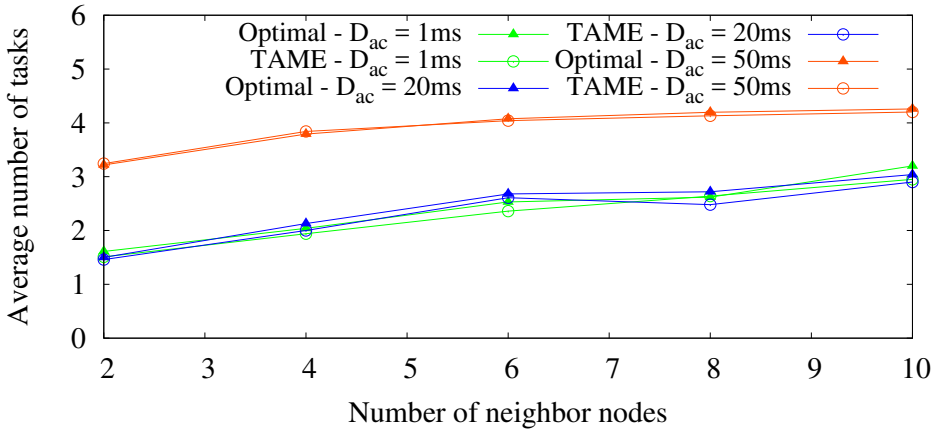

Fig. 5.10: Number of tasks offloaded to the neighbor nodes, for $|\mathscr{T}| = 6$

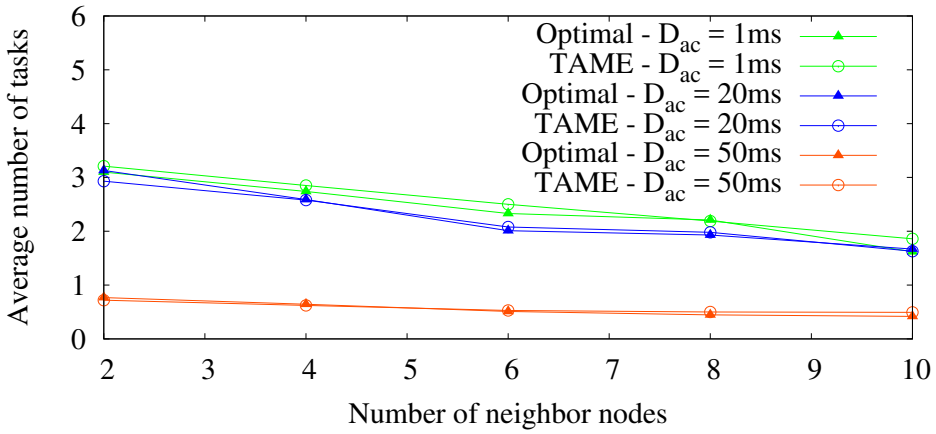

Fig. 5.11: Number of tasks offloaded to the cloud, for $|\mathscr{T}| = 6$

5.5.3 Varying the delay between the PoA and the cloud

We now investigate the impact of the propagation delay D_{ac} between the PoA and the cloud. We consider $|\mathscr{T}| = 6$ tasks, and the response delay constraint $F = 150$ ms, which is the maximum lag for real-time applications. We then assume Wi-Fi Direct connections between mobile nodes and vary $D_{ac} \in \{1, 20, 50\}$ ms. Figs. 5.9-5.11 show how the tasks are offloaded across the player's node, the neighbor nodes and the cloud. As in the previous scenario, TAME behaves almost identically to OPTIMAL. In particular, the number of tasks running on the player's node is always very low as the policy tends to distribute the tasks across all mobile nodes. With regard to the average number of tasks offloaded to the cloud (see Fig. 5.11), it is almost the same

Table 5.9: Average approximation ratio for Wi-Fi Direct (top) and Bluetooth (bottom) connections, $|\mathcal{N}| \in \{4,6,8,10,12\}$ and $|\mathcal{O}| = 10$ with $s_o = 1000$ bytes

Alg. \ $\lvert\mathcal{T}\rvert$	4	6	8	10
TAME	1.04-1.06	1.10-1.16	1.13-1.19	1.16-1.27
BESTFIT	3.09-3.30	4.57-5.17	5.41-5.93	6.19-6.56
GRAPHMERGE	1.35-1.43	1.47-1.52	1.50-1.65	1.63-1.81
Alg. \ $\lvert T\rvert$	4	6	8	10
TAME	1.05-1.07	1.09-1.17	1.10-1.28	1.11-1.32
BESTFIT	3.13-3.59	4.71-5.17	5.21-6.09	6.15-6.77
GRAPHMERGE	1.30-1.36	1.54-1.73	2.30-2.60	4.75-5.55

for D_{ac} equal to 1 ms and 20 ms, while it significantly decreases for $D_{ac} = 50$ ms. Indeed, in the latter case no tasks, or at most one task, can be offloaded to the cloud to meet the strict response delay, independently from the number of neighbor nodes.

5.5.4 Multiple objects

To investigate the effect of object retrieval, we now set the number of objects $|\mathcal{O}| = 10$ and their size to $s_0 = 1000$ bytes. During every run of simulation, each object is located in one mobile node selected at random. Each task requires a given object with probability 0.5, thus it requires on average 5 objects. Then we run exactly the same experiments performed to obtain the results in Table 5.8. Table 5.9 shows that TAME approximates within a factor 1.32 the solution obtained by OPTIMAL and still outperforms the two benchmark algorithms.

5.5.5 Real-world games

We now consider task graphs obtained from real-world games. Table 5.10 shows the approximation ratio for MICshooter and Minecraft clone, considering Wi-Fi Direct and Bluetooth communications. TAME closely matches the performance of OPTIMAL, with a maximum approximation ratio equal to 1.06, and significantly outperforms BESTFIT and GRAPHMERGE. As in the previous scenarios, BESTFIT gives the worst performance, even if with a better approximation ratio than in the case of synthetic task graphs.

Table 5.10: Average approximation ratio for Wi-Fi Direct (top) and Bluetooth (bottom) connections and $|\mathcal{N}| \in \{4, 6, 8, 10, 12\}$

Alg. \ Games	MICshooter	Minecraft clone
TAME	1.00-1.05	1.02-1.06
BESTFIT	1.45-1.45	2.60-2.62
GRAPHMERGE	1.34-1.35	1.15-1.22

Alg. \ Games	MICshooter	Minecraft clone
TAME	1.01-1.03	1.02-1.03
BESTFIT	1.45-1.45	2.51-2.55
GRAPHMERGE	1.45-1.45	2.48-2.52

In conclusion, TAME approximates very well the optimal solution, and always outperforms BESTFIT and MERGEGRAPH in all the scenarios we tested, both synthetic and real-world task graphs.

5.6 Related works

We first review the works concerning the computation offloading problem for mobile computing, then we focus on studies related to mobile gaming.

5.6.1 Offloading for mobile computing

The problem of mobile computation offloading has been widely studied. The works in [45, 95, 53, 68, 93, 42] address the mobile cloud computation offloading problem theoretically. In particular, [45] advocates dynamic application partitioning to adapt to ever-changing network conditions. [95, 53] exploit task offloading to achieve high throughput for data stream applications. [68] proposes a task offloading and scheduling framework to minimize energy consumption. [93] combines mobile cloud offloading and dynamic frequency scaling to achieve energy efficiency. [42] proves that the problem of multi-user computation offloading converges to a Nash Equilibrium, and proposes a distributed approach based on game theory to decide local or cloud execution for the tasks of each user. [79] proposes a mobile task offloading algorithm that merges computationally light but heavily communication

tasks into super-tasks, and offloads them to the most suitable mobile nodes. However, all the above studies only address the mobile cloud offloading problem without exploiting the available resource in the mobile fog. One of the algorithms used for our performance comparison, named MERGEGRAPH and presented in Sec. 5.4.1, is an adaptation of the algorithm proposed in [79] to our hybrid cloud/fog scenario.

Another body of works [47, 57, 82, 60, 46, 104] aim at implementing task offloading platforms in practice. Specifically, MAUI [47] is a system supporting energy-aware, fine-grained code offloading for .NET smartphone applications. It uses .NET common language runtime to ensure platform independence, identify the remoteable tasks with all the related states, and determine the offloading costs. MAUI decides the offloading results by solving a linear programming optimization problem based on online measurements of CPU and network cost. Since .NET serialization is used to profile the mobile applications at runtime, MAUI supports continuous profiling and provides optimal offloading strategies on the fly. Cuckoo [57], instead, is a runtime application offloading framework for the Android platform. It offers a programming model to help developers implement applications remotely, a resource manager to collect and manage remote resources, and a runtime system to perform optimal offloading decisions based on the selected remote nodes. Likewise, Jade [82] is a dynamic computation offloading system, which provides a programming model to develop Android applications with computation offloading capability. In addition, it implements a multi-level data storage to manage data synchronization and avoid unnecessary data transfers. ThinkAir [60] is another runtime computation offloading framework for the Android platform. Aside from providing an easy-to-use interface for developers, ThinkAir enables on-demand paralleled task execution by invoking multiple instances of virtual machines to improve the scalability of the cloud. CloneCloud [46] is a system that automatically transforms Android applications so that they can support computation offloading and benefit the available resource in the cloud. It combines static analysis and dynamic profiling on Java bytecode so as to partition applications in finer granularity, and it aims at achieving both energy-efficiency and execution time optimization. Offloading is performed at thread level, and only those threads that do not require local state are qualified for remote execution. Similarly, DPartner [104] automatically re-factors Android bytecode (thus, without the need for the original source files) to enable computation offloading. *All the above implementation platforms and techniques can be considered as possible key enabling technologies to support our proposed offloading scheme in*

IMG scenarios. In our work, we focus just on the optimal way to offload the tasks, whose feasibility is proved by the aforementioned works.

Relevant to our work are also the studies [89, 91], which investigate the integration of cloud and edge/fog computing to offload tasks for mobile applications. [89] implements a framework to offload a face recognition application, either at some edge servers or to the cloud whenever the edge servers are not available. Notably, face recognition applications use predefined task graphs, which, compared to those of mobile games, are not real-time. Besides, the energy cost is not considered since face recognition algorithms are computation intensive and are always offloaded. [91] proposes a combined cloud and fog/edge architecture, where the cloud servers, the neighbor mobile nodes and the edge servers are all considered as candidates for task offloading. An ILP optimization problem to allocate tasks is formulated, with the objective of minimizing latency. The application is modeled as a set of independent tasks, thus, unlike our work, it does not consider the dependencies and the communication between different tasks. Furthermore, [91] considers neither energy costs nor communication latency.

5.6.2 Mobile gaming

With regard to mobile gaming, [50, 31, 97, 43] study possible methods to offload the artificial intelligence (AI) components of mobile games. [50] proposes an approach to offload part of the AI tasks to external servers and exploit their higher computational power, so as to allow more complex AI components, hereby more interesting and challenging games to play. Similarly, [31] studies the performance of AI offloading with various network latency and evaluates the effectiveness of dead reckoning algorithms to mitigate the performance degradation due to large delays. Furthermore, [43] exploits the general-purposed GPUs to offload the AI components of chess games, so as to perform more complicated computations and improve the AI's winning rates. Besides, [97] studies the performance of offloading the AI components to servers available in the local network by varying the complexity and number of AIs. Obviously, all these works offload tasks at the component level. In contrast, our work offloads tasks at a finer granularity (class level), thus providing higher offloading flexibility.

Instead of studying solely the possibility of offloading AI components, [38, 44, 66] aim at implementing general frameworks for mobile gaming. [38] proposes a framework to decompose a mobile game into tasks and adaptively migrate them between the player's device and the cloud server. The main difference with respect to our work is that, instead of just considering the cloud, our model exploits neighbor mobile devices. Furthermore, their work provides the best offloading scheme through an exhaustive search of the possible offloading choices, which faces scalability issues; the existence of two heuristics is just mentioned without providing details. [44] and [66] exploit the resource of both the cloud and the edge servers to enhance the quality of mobile games. Aside from cloud servers, [44] incorporates content delivery network (CDN) servers as edge servers and deploy them close to mobile users, so as to increase end-user coverage. Similarly, [66] envisions servers available at the user premises as fog servers in charge of video rendering and transmission, so that the user coverage increases while transmission delay and bandwidth consumption decrease. Unlike [44, 66], our work exploits neighbor mobile devices like smartphones and tablets to offload tasks, thus achieving a higher level of pervasiveness thanks to the popularity of such devices.

Note that the incentives for cooperation are crucial for the considered offloading application, but they are outside the scope of the present work. [41] and [103] give detailed discussions about it.

5.7 Summary

We devised a combined Integrated Mobile Gaming (IMG) scheme that efficiently offloads some internal tasks of a game running on the player's device toward neighbor mobile nodes or the cloud. We first formalized the problem of energy-aware task allocation as an ILP problem, which minimizes the maximum energy consumption across all the mobile nodes while accounting for the communication and computation costs involved in running and migrating tasks. Then, in light of the problem complexity, we proposed TAME, an algorithm that, at each iteration, adapts its allocation decisions based on the major factor (either communication or computation) contributing to energy consumption. We evaluated TAME in the case of synthetic and real-world scenarios. Our results show that TAME always approximates very closely the optimal solution and outperforms other state-of-art algorithms. They also

highlight the advantages of task offloading toward neighbor mobile nodes, especially when communication latency with the cloud is significant. Importantly, thanks to the recent availability of offloading platforms, our TAME algorithm can be integrated in real platforms for IMG, enabling highly pervasiveness and efficient real-time mobile gaming.

Chapter 6

Conclusion

In this thesis, we specifically concentrate on two enabling techniques involved in the emerging 5G networks, namely Software Defined Networking and mobile fog computing. In particular, we propose a set of approaches to optimize the SDN control plane in terms of communication latency, scalability as well as reliability, in Chapters 2, 3 and 4, respectively. In Chapter 5, we propose an energy-efficient Integrated Mobile Gaming platform to allocate gaming tasks in the context of fog computing.

In Chapter 2, we begin by investigating the controller placement problem under scenario of in-band control plane. Unlike previous works that considered only the traffic exchange between switches and controllers, we additionally consider the latency incurred by inter-controller communications, since they affect the time required to reach the consensus of the data structures that are shared across multiple controllers. We further define two data ownership models, namely Single Data-Ownership (SDO) and Multiple Data-Ownership (MDO). Both models are implemented in state-of-the-art SDN controllers. In SDO model, controllers such as ONOS and ODL use consensus algorithms (e.g., Raft consensus algorithm) to synchronize a set of data structures requiring strong consistency, making the inter-controller latency non-ignorable. The controller placement problem is studied by building the Pareto frontier for Sw-Ctr and Ctr-Ctr latencies. By observing the Pareto frontier of 114 real ISP network topologies in the Internet Topology Zoo, we claim that a controller placement with small Sw-Ctr delay may impose a much larger Ctr-Ctr delay, and the trade-off between them should be carefully considered

when planning a network deployment. Then we formulate the problem of optimal placement for minimum reaction time as ILP for both data ownership models, and solve it using Gurobi optimizer. Based on the experiments on 89 real ISP topologies, we claim that for SDO model selecting the closest controller as master might not always renders the optimal average end-to-end latency. The last contribution is the implementation of two evolutionary algorithms: BEST-REACTIVITY and EVO-PLACE. The former aims at finding the controller placement with minimum reaction time, while the latter strives for the optimal Pareto frontier. Both of them are based on evolutionary approaches and creates new placements by perturbing the locations of controllers. As demonstrated by experiments, BEST-REACTIVITY approximates the optimal solution with an error rate of less than 30% by just scanning less than 10% of the whole search space. On the other hand, EVO-PLACE always outperforms random walks by providing a better Pareto frontier.

In Chapter 3, we focus on the scalability of the control plane of SDN and exploit OpenState extension to integrate stateful SDN approach with traffic classification. In specific, we design two solutions to accurately redirect packets to the traffic classifiers. Memory footprints of both solutions are analyzed in detail. Experimental results demonstrate that our solutions dramatically decrease the amount of traffic received by both the controller and the traffic classifier, thus scale the SDN control plane.

In Chapter 4, we concentrate to the reliability issues of the SDN control plane and propose BeCheck, which transparently inspects messages of the multiple instances of controllers and detects the misbehaving ones through a voting scheme. BeCheck replies the intercepted messages to the underlying network devices based on one of three forwarding policies, whose trade-off between control plane reactivity and identification accuracy are carefully studied.

In Chapter 5, we propose a brand-new task allocation platform for Integrated Mobile Gaming (IMG) leveraging fog computing, we formulate the problem of energy-efficient task offloading and we devise a heuristic algorithm named TAME to find the optimal task offloading scheme. In IMG, mobile games are partitioned into tasks with finer granularities and offloaded to either the cloud or the neighboring mobile devices for remote execution. By doing so, the computation resources are expected to be fully utilized. We firstly formulate the task offloading problem as an ILP and solve it. To cope with scalability issues, we come up with the TAME

algorithm. In each iteration, TAME evaluates the task graph, selects task(s) based on the evaluation of energy cost due to both computation and communication, and offloads the select task(s) to the most suitable physical node. TAME is evaluated against two state-of-the-art task offloading algorithms, in the context of both synthetic and real-world mobile games. As shown by the results, TAME outperforms both algorithms in almost all the cases.

In summary, we provide a set of tools to facilitate the performance optimization of SDN and fog computing, both of which are key enabling technologies in 5G networks. We believe all the proposed approaches could be integrated into real networks.

References

[1] OpenState SDN. URL http://openstate-sdn.org/.

[2] Sizing individual objects. URL https://pythonhosted.org/Pympler/asizeof.html.

[3] profile and cProfile Module Reference. URL https://docs.python.org/2/library/profile.html#module-cProfile.

[4] Floodlight. URL http://www.projectfloodlight.org/floodlight/.

[5] Gaikai. inc. URL http://www.gaikai.com/.

[6] Samsung Exynos 7 Octa 5433 Mobile Processor, . URL http://www.samsung.com/semiconductor/minisite/Exynos/Solution/MobileProcessor/Exynos_7_Octa_5433.html.

[7] Samsung Exynos 7 Octa 7420 Mobile Processor, . URL http://www.samsung.com/semiconductor/minisite/Exynos/Solution/MobileProcessor/Exynos_7_Octa_7420.html.

[8] Gurobi optimizer reference manual. URL http://www.gurobi.com.

[9] Maestro. URL https://code.google.com/p/maestro-platform/.

[10] MICShooter - old school space shooter. URL http://www.pygame.org/project-MICshooter-2342-.html.

[11] Minecraft - Simple Minecraft-inspired demo written in Python and Pyglet. URL https://github.com/fogleman/Minecraft.

[12] nDPI. URL http://www.ntop.org/products/deep-packet-inspection/ndpi.

[13] Newzoo Free 2016 Global Games Market Report. URL https://newzoo.com/insights/articles/.

[14] Java I/O, NIO, and NIO.2. URL https://docs.oracle.com/javase/8/docs/technotes/guides/io/.

[15] The NOX Controller. URL https://github.com/noxrepo/nox.

[16] The OpenDaylight Platform. URL https://www.opendaylight.org/.

[17] Onlive. inc. URL http://www.onlive.com/.

[18] Distributed primitives. URL https://wiki.onosproject.org/display/ONOS/Distributed+Primitives.

[19] OpenDPI github repository, . URL https://github.com/thomasbhatia/OpenDPI.

[20] OpenFlow Switch Specification v1.5.1, . URL https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf.

[21] OpenFlowJ Loxi, . URL https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi.

[22] SDN system performance. URL http://www.pica8.com/pica8-deep-dive/sdn-system-performance/.

[23] The POX controller. URL https://github.com/noxrepo/pox.

[24] Python call graph. URL http://pycallgraph.slowchop.com/en/master/.

[25] OpenDaylight Raft consensus code review. URL https://github.com/opendaylight/controller/tree/master/opendaylight/md-sal/sal-akka-raft/src/main/java/org/opendaylight/controller/cluster/raft.

[26] Research Report on Global Market. URL https://www.wiseguyreports.com/.

[27] The Internet Topology Zoo. URL http://www.topology-zoo.org/.

[28] Patrick Kwadwo Agyapong, Mikio Iwamura, Dirk Staehle, Wolfgang Kiess, and Anass Benjebbour. Design considerations for a 5g network architecture. *IEEE Communications Magazine*, 52(11):65–75, 2014.

[29] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, 2016.

[30] Sébastien Auroux and Holger Karl. Flow processing-aware controller placement in wireless DenseNets. In *PIMRC*, pages 1294–1299. IEEE, 2014.

[31] Jiaqiang Bai, Daryl Seah, James Yong, and Ben Leong. Offloading AI for peer-to-peer games with dead reckoning. In *IPTPS*, page 3, 2009.

[32] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. *SIGPLAN Not.*, 49(6):282–293, June 2014. ISSN 0362-1340.

[33] Md Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Dynamic controller provisioning in Software Defined Networks. In *CNSM*, pages 18–25, Zurich, Switzerland, 2013.

[34] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *ACM HotSDN*, New York, NY, USA, 2014.

[35] Laurent Bernaille, Renata Teixeira, and Kave Salamatian. Early application identification. In *ACM CoNEXT*, New York, NY, USA, 2006. ISBN 1-59593-456-1.

[36] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2): 44–51, 2014.

[37] Eric Brewer. Pushing the CAP: Strategies for consistency and availability. *Computer*, 45(2):23–29, February 2012. ISSN 0018-9162.

[38] Wei Cai, Henry CB Chan, Xiaofei Wang, and Victor CM Leung. Cognitive resource optimization for the decomposed cloud gaming platform. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12):2038–2051, Dec 2015.

[39] Marco Canini, Damien Fay, David J. Miller, Andrew W. Moore, and Raffaele Bolla. Per flow packet sampling for high-speed network monitoring. In *IEEE COMSNETS*, 2009.

[40] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A NICE way to test OpenFlow applications. In *NSDI*, volume 12, pages 127–140, 2012.

[41] Xu Chen, Brian Proulx, Xiaowen Gong, and Junshan Zhang. Exploiting social ties for cooperative d2d communications: A mobile social networking case. *IEEE/ACM Transactions on Networking*, 23(5):1471–1484, Oct 2015.

[42] 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, 2016.

[43] Kihan Choi, Jaehun Lee, Youngjin Kim, Sooyong Kang, and Hyuck Han. Feasibility of the computation task offloading to GPGPU-enabled devices in mobile cloud. In *IEEE ICCAC*, pages 244–251, 2015.

[44] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. A hybrid edge-cloud architecture for reducing on-demand gaming latency. *Multimedia Systems*, 20(5):503–519, 2014.

[45] Byung-Gon Chun and Petros Maniatis. Dynamically partitioning applications between weak devices and clouds. In *MCS*. ACM, 2010.

[46] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys*, pages 301–314. ACM, 2011.

[47] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *MobiSys*, pages 49–62. ACM, 2010.

[48] Panagiotis Demestichas, Andreas Georgakopoulos, Dimitrios Karvounas, Kostas Tsagkaris, Vera Stavroulaki, Jianmin Lu, Chunshan Xiong, and Jing Yao. 5g on the horizon: key challenges for the radio-access network. *IEEE Vehicular Technology Magazine*, 8(3):47–53, 2013.

[49] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS*, 2015.

[50] John R Douceur, Jacob R Lorch, Frank Uyeda, and Randall C Wood. Enhancing game-server AI with distributed client computation. *ACM NOSSDAV*, 2007.

[51] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency analysis for software-defined networks. *SIGPLAN Not.*, 51(6):402–415, June 2016. ISSN 0362-1340.

[52] Raihana Ferdous, Renato Lo Cigno, and Alessandro Zorat. Classification of SIP messages by a syntax filter and SVMs. In *IEEE GLOBECOM*, 2012.

[53] Javad Ghaderi, Sanjay Shakkottai, and R Srikant. Scheduling storms and streams in the cloud. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(4), 2016.

[54] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *ACM HotSDN*, pages 7–12, 2012.

[55] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. On reliability-optimized controller placement for software-defined networks. *China Communications*, 11(2):38–54, 2014.

[56] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. GamingAnywhere: an open cloud gaming system. In *ACM MMSys*, pages 36–47. ACM, 2013.

[57] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *MobiCASE*, pages 59–79. Springer, 2010.

[58] Jawad M Khalife, Amjad Hajjar, and Jesús Díaz-Verdejo. Performance of OpenDPI in identifying sampled network traffic. *Journal of Networks*, 2013.

[59] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[60] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *IEEE Infocom*, pages 945–953, 2012.

[61] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *HotSDN*, pages 55–60. ACM, 2013.

[62] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015. ISSN 0018-9219.

[63] Stanislav Lange, Steffen Gebert, Thomas Zinner, Phuoc Tran-Gia, David Hock, Michael Jarschel, and Marco Hoffmann. Heuristic approaches to the controller placement problem in large scale SDN networks. *IEEE Transactions on Network and Service Management*, 12(1):4–17, March 2015. ISSN 1932-4537.

[64] Yeng-Ting Lee, Kuan-Ta Chen, Han-I Su, and Chin-Laung Lei. Are all games equally cloud-gaming-friendly?: an electromyographic approach. In *IEEE NetGames*, page 3, 2012.

[65] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *ACM HotSDN*, New York, NY, USA, 2012.

[66] Yuhua Lin and Haiying Shen. CloudFog: Leveraging fog to extend cloud gaming for thin-client MMOG with high quality of service. *IEEE Transactions on Parallel and Distributed Systems*, 28, 2017.

[67] Kshiteej Mahajan, Rishabh Poddar, Mohan Dhawan, and Vijay Mann. Jury: Validating controller actions in software-defined networks. In *IEEE/IFIP DSN*, pages 109–120, June 2016. doi: 10.1109/DSN.2016.19.

[68] S Eman Mahmoodi, RN Uma, and KP Subbalakshmi. Optimal joint scheduling and cloud offloading for mobile applications. *IEEE Transactions on Cloud Computing*, 2016.

[69] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *IEEE FMCAD*, pages 163–170, 2014.

[70] Stephanos Matsumoto, Samuel Hitz, and Adrian Perrig. Fleet: Defending SDNs from malicious administrators. In *HotSDN*, pages 103–108, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620750.

[71] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. ISSN 0146-4833.

[72] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven SDN controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6, 2014.

[73] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31(1):114–127, 1984.

[74] Lucas F Müller, Rodrigo R Oliveira, Marcelo C Luizelli, Luciano P Gaspary, and Marinho P Barcellos. Survivor: an enhanced controller placement strategy for improving SDN survivability. In *GLOBECOM*, pages 1909–1915. IEEE, 2014.

[75] A.S. Muqaddas, Andrea Bianco, Paolo Giaccone, and Guido Maier. Inter-controller traffic in ONOS clusters for SDN networks. In *IEEE ICC*, Kuala Lumpur, Malaysia, May 2016.

[76] ODL clustring. OpenDaylight controller clustering. URL https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Architecture:Clustering.

[77] Tatsuya Okabe, Yaochu Jin, and Bernhard Sendhoff. A critical survey of performance indices for multi-objective optimisation. In *Congress on Evolutionary Computation (CEC)*, volume 2, pages 878–885, Dec. 2003.

[78] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, Philadelphia, PA, 2014.

[79] Shumao Ou, Kun Yang, and Jie Zhang. An effective offloading middleware for pervasive services on mobile devices. *Pervasive and Mobile Computing*, 3 (4):362–385, 2007.

[80] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. CAP for networks. In *HotSDN*, New York, NY, USA, 2013. ISBN 978-1-4503-2178-5.

[81] Joshi Prajakta. ONOS Summit: ONOS Roadmap 2015 , Dec 2014.

[82] Hao Qian and Daniel Andresen. An energy-saving task scheduler for mobile devices. In *IEEE/ACIS ICIS*, pages 423–430. IEEE, 2015.

[83] Hemant Kumar Rath, Vishvesh Revoori, SM Nadaf, and Anantha Simha. Optimal controller placement in Software Defined Networks (SDN) using a non-zero-sum game. In *WoWMoM*, pages 1–6. IEEE, 2014.

[84] Francisco Javier Ros and Pedro Miguel Ruiz. Five nines of southbound reliability in software defined networks. In *ACM HotSDN*, New York, NY, USA, 2014.

[85] Sudhir K Routray and KP Sharmila. Software defined networking for 5g. In *Advanced Computing and Communication Systems (ICACCS), 2017 4th International Conference on*, pages 1–5. IEEE, 2017.

[86] Afrim Sallahi and Marc St-Hilaire. Optimal model for the controller placement problem in software defined networks. *IEEE Communications Letters*, 19(1): 30–33, Jan. 2015. ISSN 1089-7798.

[87] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, et al. Troubleshooting blackbox SDN control software with minimal causal sequences. *ACM SIGCOMM Computer Communication Review*, 44(4):395–406, 2015.

[88] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. Technical report, OpenFlow Switch Consortium, 2009.

[89] Francisco Airton Silva, Paulo Maciel, Rubens Matos, et al. A scheduler for mobile cloud based on weighted metrics and dynamic context evaluation. In *ACM SAC*, pages 569–576, 2015.

[90] Omar Soliman, Abdelmounaam Rezgui, Hamdy Soliman, and Najib Manea. Mobile cloud gaming: Issues and challenges. In *MobiWis*, pages 121–128. Springer, 2013.

[91] VBC Souza, W Ramírez, Xavier Masip-Bruin, Eva Marín-Tordera, G Ren, and Ghazal Tashakor. Handling service allocation in combined fog-cloud scenarios. In *IEEE ICC*, pages 1–5, 2016.

[92] Mininet Team. Mininet: An instant virtual network on your laptop (or other PC), 2012. URL http://mininet.org.

[93] Yonggang Wen, Weiwen Zhang, and Haiyun Luo. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *IEEE Infocom*, pages 2716–2720, 2012.

[94] Peng Xiao, Wenyu Qu, Heng Qi, Zhiyang Li, and Yujie Xu. The SDN controller placement problem for WAN. In *ICCC*, pages 220–224. IEEE, 2014.

[95] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 40 (4):23–32, 2013.

[96] Guang Yao, Jun Bi, Yuliang Li, and Luyi Guo. On the capacitated controller placement problem in Software Defined Networks. *IEEE Communications Letters*, 18(8):1339–1342, Aug 2014. ISSN 1089-7798.

[97] GingFung Matthew Yeung. Cloud gaming and simulation in distributed systems, 2015. URL https://minerva.leeds.ac.uk/bbcswebdav/orgs/SCH_Computing/FYProj/reports/1415/YEUNG.pdf.

[98] Tianzhu Zhang, Carla F. Chiasserini, and Paolo Giaccone. TAME: an Efficient Task Allocation Algorithm for Integrated Mobile Gaming. *IEEE Systems Journal*.

[99] Tianzhu Zhang, Andrea Bianco, and Paolo Giaccone. The role of inter-controller traffic in sdn controllers placement. In *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*, pages 87–92. IEEE, 2016.

[100] Tianzhu Zhang, Andrea Bianco, Paolo Giaccone, Seyedaidin Kelki, Nicolas Mejia Campos, and Stefano Traverso. On-the-fly traffic classification and control with a stateful SDN approach. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2017. doi: 10.1109/ICC.2017.7997297.

[101] Tianzhu Zhang, Andrea Bianco, Paolo Giaccone, and Aliakbar Payandehdari Nezhad. Dealing with misbehaving controllers in SDN networks. pages 1–6, 2017.

[102] Tianzhu Zhang, Paolo Giaccone, Andrea Bianco, and Samuele De Domenico. The role of the inter-controller consensus in the placement of distributed SDN controllers. *Computer Communications*, 113(Supplement C):1 – 13, 2017. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2017.09.007. URL http://www.sciencedirect.com/science/article/pii/S0140366417306217.

[103] Yanru Zhang, Lingyang Song, Walid Saad, Zaher Dawy, and Zhu Han. Contract-based incentive mechanisms for device-to-device communications in cellular networks. *IEEE Journal on Selected Areas in Communications*, 33 (10):2144–2155, Oct 2015. ISSN 0733-8716.

[104] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In *SIGPLAN Notices*, volume 47, pages 233–248. ACM, 2012.