UNIVERSITÀ
DEGLI STUDI
FIRENZE

PhD Program in Smart Computing
Dipartimento di Ingegneria dell'Informazione (DINFO)

Measurements in the Edge-Cloud Continuum: Network Metrics and Energy Consumption

Chiara Caiazza

Dissertation presented in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Smart Computing

PhD Program in Smart Computing
University of Florence, University of Pisa, University of Siena

Measurements in the Edge-Cloud Continuum: Network Metrics and Energy Consumption

Chiara Caiazza

Advisor:

Prof. Alessio Vecchio

Head of the PhD Program:

Prof. Stefano Berretti

Evaluation Committee:
Prof. Dario Bruneo, *Università degli Studi di Messina*
Prof. Vincenzo Mancuso, *IMDEA Networks Institute*

XXXIV ciclo — January 2022

To my family

Acknowledgments

First and foremost I would like to thank my Advisor Prof. Alessio Vecchio for the invaluable and patient guidance received during my Ph.D. I would also like to thank Researcher Valerio Luconi for the valuable advice received during these 3 years. I would like to thank the members of my Supervisory Committee Prof. Enzo Mingozzi and Researcher Raffaele Bruno for having provided useful recommendations that helped me to improve my work. Then, I would also like to express my gratitude to Prof. Silvia Giordano for providing me with a stimulating collaboration during my period abroad despite the countless problems caused by the pandemic. Finally, I would thank Regione Toscana for the Pegaso grant and Simona Altamura for her administrative support.

Abstract

Edge computing is an emerging network paradigm based on the idea of moving computational and storage resources closer to the end users. This type of architecture can bring a variety of benefits compared to the traditional cloud-based one, such as lower latency, higher throughput, increased privacy, and reduced congestion of the Internet core. However, making an effective use of edge computing requires to monitor the performance of the network, e.g. to take appropriate decisions about if and where computation should be offloaded, or which server in the edge-cloud continuum is more suitable for a given operation.

Most of the existing network measurement methods and tools are not specifically designed to operate in an edge computing scenario. For this reason, mechanisms aimed at collecting network metrics in an edge environment have been first designed and then used to collect experimental data in a realistic testbed. Network measurements can also be useful at design time, e.g. to evaluate different edge/cloud solutions by means of trace-driven simulations, as discussed in this thesis. The energy needed by client devices to communicate with edge or cloud resources is another important aspect, since such devices, which include smartphones and IoT nodes, are generally battery-operated. To better understand how the edge computing paradigm impacts the energy needed to communicate, an analytical model of a request-response communication scheme has been defined. The model highlights that the improved latency of an edge server, compared to a cloud one, can reduce the energy needed by clients. Energy savings are particularly significant when communication takes place according to a connection-oriented protocol.

This thesis also looks at the path between client nodes and cloud resources from a purely topological perspective. Traceroute is the most commonly used tool, not only for network diagnostics, but also for discovering all the nodes towards a server. We evaluated the discovery capability of three variations of TCP-based traceroute. The first version is the classical one and uses SYN segments as probes. The other two versions operate on a connection already established with a server and use DATA and ACK segments as probes. This is done to possibly bypass traceroute suppression mechanisms or firewalls. Experimental results show that using different types of probes is useful to obtain a richer view of the path towards network resources.

Contents

Contents **1**

List of Figures **3**

List of Tables **8**

1 Introduction **11**
1.1 Multi-access Edge Computing 11
1.2 Understanding the paths towards remote cloud servers 13
1.3 Contributions 14
1.4 Outline 15

2 Background **17**
2.1 Compare Edge- and cloud-based performance 17
2.2 Evaluating the energy consumption of a Terminal Node in an edge-computing system 23
2.3 Evaluating the paths towards the cloud 25

3 Collecting performance metrics in a MEC environment: challenges and requirements **29**
3.1 The MEC architecture 29

4 A tool for collecting network metrics in a MEC environment **33**
4.1 MECPerf architecture 33
4.2 Validation 43

5 Data collection and experimental results **49**
5.1 Network setup 49
5.2 MECPerf-active measurement results 55
5.3 DASH measurement results 61
5.4 Lessons learned 71

6 Use MECPerf experimental results to build a simple trace-based network simulator **73**
6.1 Using the experimental results to generate input traces 74
6.2 Use the NetworkTraceManager to implement a simple simulator . . . 78

7 Estimating the energy consumption of terminal nodes in edge/cloud scenarios **81**
7.1 Modeling the LTE interface as a finite state machine 82
7.2 Estimating the energy consumption of a connectionless application . 84
7.3 Estimating the client-side energy consumption of a connection-oriented application 87
7.4 The Energy evaluator module 91
7.5 The energy consumption of an ideal connectionless application 91
7.6 The energy consumption of a trace-based connection-oriented application 99

8 Evaluating the path to remote clouds **107**
8.1 The camouflage traceroute software modules 108
8.2 The camouflage traceroute discovery capabilities 113

9 Conclusions **125**

A Evaluating the computational load of MECPerf **129**

B List of Acronyms **133**

C Publications **137**

Bibliography **141**

List of Figures

3.1 The high-level architecture of a MEC network. 30
3.2 A path composed of three links with different utilization levels, available bandwidths, and capacities. 30

4.1 The four components composing the MECPerf architecture and the interaction between them during MECPerf-active, MECPerf-passive, self-active, and self-passive measures. 34
4.2 The interaction between the MECPerf Client (MC), the MECPerf Observer (MO), and the MECPerf Remote Server (MRS) during a generic MECPerf-active measure. 37
4.3 The sequence diagram of a TCP MECPerf-active bandwidth measure. . . 38
4.4 The basic functioning mechanism behind the packet pair technique. . . . 39
4.5 The sequence diagram of a UDP MECPerf-active bottleneck capacity measure. 40
4.6 The sequence diagram of a MECPerf-active latency measure. 41
4.7 The setup used during the validation of the MECPerf collection system. . 43
4.8 The results of the validation for MECPerf-active latency measurement methods. RTT values have been averaged among 10 repetitions. 44
4.9 The results of the validation for MECPerf-active TCP bandwidth and UDP bottleneck capacity measurement methods. TCP bandwidth and UDP bottleneck capacity values have been averaged among 10 repetitions. 45

5.1 The setup used during the collection of MECPerf-active metrics. 51
5.2 The setup used during the collection of MECPerf- and self-passive metrics for a DASH application. 53
5.3 The boxplots of MECPerf-active TCP bandwidth metrics computed during Wi-Fi and LTE experiments. 56
5.4 The boxplots of MECPerf-active UDP and TCP latency metrics computed during Wi-Fi experiments. 58
5.5 The boxplots of MECPerf-active UDP and TCP latency metrics computed during LTE experiments. 59

5.6 The results collected during UDP bottleneck capacity MECPerf-active downlink Wi-Fi experiments considering a scenario where no cross-traffic is injected into the access network. 60
5.7 The results collected during MECPerf-passive downlink bandwidth Wi-Fi experiments considering different levels of cross-traffic injected into the access network. The bandwidth values have been computed considering buckets of 0.5 seconds. 63
5.8 The results collected during MECPerf-passive downlink bandwidth LTE experiments considering different levels of cross-traffic injected into the access network. The bandwidth values have been computed considering buckets of 0.5 seconds. 64
5.9 The boxplots of self-passive bandwidth metrics. The results are based on Wi-Fi downlink experiments and consider a scenario where a cross-traffic generator injects different levels of cross-traffic into the access network. . 65
5.10 The boxplots of self-passive bandwidth metrics. The results are based on LTE downlink experiments and consider a scenario where a cross-traffic generator injects different levels of cross-traffic into the access network. . 66
5.11 The uplink and downlink traffic during DASH Wi-Fi experiments. 67
5.12 The boxplots of MECPerf-passive downlink latency metrics, representing the performance of the wireless segment that connects the clients and the AP. The results are based on Wi-Fi experiments and consider different scenarios where a cross-traffic generator injects different levels of cross-traffic into the access network. 68
5.13 The boxplots of MECPerf-passive uplink latency metrics, representing the performance of the wired network segments that connect the AP with both MEC and cloud servers. The results are based on Wi-Fi experiments and consider different scenarios where a cross-traffic generator injects different levels of cross-traffic into the access network. 69
5.14 The uplink and downlink traffic during DASH LTE experiments. 70
5.15 The boxplots of MECPerf-passive downlink latency metrics, representing the performance of the wireless segment that connects the clients and the AP. The results are based on LTE experiments and consider a scenario where a cross-traffic generator injects different levels of cross-traffic into the access network. 70

6.1 The interaction between the MECperf collection tools, the MECPerf Library, and third-party software that requires metrics to compute simulation or analysis activities. 74
6.2 The interaction between the MECPerf Library and a generic third-party software components. 76
6.3 The simulated architecture. 79

6.4 The 0,5, 0,75, 0,95 RTT quantiles for γ values that goes form 0 to 0,5. Values are plotted considering a 95% confidence interval. 79

7.1 The reference architecture considered in this Chapter. The client application is running on TNs hosted in the access network using LTE connections to communicate with servers located on both edge and cloud networks. 82
7.2 The finite state machine of an LTE module. 82
7.3 The interaction between the client and the server considering a connectionless application operating on an application period T_I. 85
7.4 The interaction between the client and the server, considering a connection-oriented application operating on an application period T_I. 88
7.5 The TCP-based application used to collect T_{TX}, T_{RX}, and T_W values in a real environment. 89
7.6 The setup used to collect T_{TX}, T_{RX}, and T_W values in a real environment for a connection-oriented application. 90
7.7 The values of ρ obtained setting the amount of transmitted data and the elaboration time, while variable RTT^C and T_I are adopted. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge. 93
7.8 The values of ρ obtained setting the application period and the elaboration time, while a variable RTT^C and a variable amount of transmitted data are considered. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge. 96
7.9 The values of ρ obtained setting the application period and the amount of transmitted data, while variable RTT^C and elaboration times are adopted. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge. 97
7.10 The values of ρ obtained setting the amount of transmitted data, while variable elaboration times and application periods are adopted. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge. . 98
7.11 The value of ρ collected for a connection-oriented application when an increasing amount of data transferred is considered. The results are based on T_{TX}, T_W, and T_{RX} values collected at night. 100

7.12 The mean energy consumption (E_I) required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds and an application period (T_I) of 40 seconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed at night. The plot shows the 99% of confidence interval of the mean values. 101

7.13 The mean time required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed at night. The plot shows the 99% of confidence interval of the mean values. 102

7.14 The mean time required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed during the day. The plot shows the 99% of confidence interval of the mean values. 104

7.15 The value of ρ collected for a connection-oriented application when an increasing amount of data transferred is considered. The results are based on T_{TX}, T_W, and T_{RX} values collected during the day. 105

7.16 The mean energy consumption (E_I) required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds and an application period (T_I) of 40 seconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed during the day. The plot shows the 99% of confidence interval of the mean values. 106

8.1 The basic operating principles of camotrace. The source node combines probes and legitimate HTTP GET requests, eliciting some ICMP Time Exceeded messages from the intermediate routers. 108

8.2 The software components of the ACK-based version of camotrace and their interactions. 112

8.3 The setup used to validate the data segment-based version of camotrace. 113

8.4 The 20 states where the majority of the targets are hosted and the distance in kilometers between the source and the target in the corresponding country. 116

8.5 The eCDF of the AS path length using the data segment-based camotrace, the ACK-based camotrace, and the TCP traceroute probing methods. . . 117

8.6 The eCDF of the number of interfaces per AS using the data segment-based camotrace, the ACK-based camotrace, and the TCP traceroute probing methods 118

8.7 The upset plots of the number of unique IP interfaces, unique AS numbers, unique AS links found by the SYN-, the DATA-, and the ACK-based methods. 119

8.8 The percentage of paths for which differences in terms of number of hops can be appreciated when (1) the DATA-based methods found at least one hop that cannot be discovered using the SYN-based method, (2) the SYN-based methods found at least one hop that cannot be discovered using the DATA-based method, (3) both methods found at least one hop that cannot be discovered using the other method, or (4) the two methods found the same hops. 120
8.9 The number of additional hops discovered. 121
8.10 The normalized position of groups of additional hops 122
8.11 An example of d_M^{k*} computation. 123

A.1 The setup used to assess the computational load of MECPerf. 129
A.2 The CPU usage rate measured on the MO during a set of experiments aimed at evaluating the computational load of MECPerf. Precisely, each value represents the average rate monitored for 1000 seconds during a set of TCP uplink measures. 130
A.3 The CPU usage rate with and without the optimizations applied to the measurement library. 131

List of Tables

2.1 A comparison between emulators and simulators in the field of edge computing systems. The comparison was based on network-related characteristics. 28

4.1 The values of the operational parameters used during the validation of both MECPerf-active TCP bandwidth and MECPerf-active UDP bottleneck capacity methods. 45
4.2 The results obtained during the validation for MECPerf-passive measurement methods. 47

5.1 The technical characteristics of the devices used during MECPerf-active, MECPerf-passive, and self-passive experimental data collection. 50
5.2 The operational parameters used to collect MECPerf-active metrics. . . . 52
5.3 The operational parameters used to collect MECPerf- and self-passive metrics. 55

7.1 The mean power consumption of the LTE interface adopted. 83
7.2 The maximum time spent in each state of the adopted FSM. 84
7.3 The setup used to compute the ρ values of Figure 7.7. 93
7.4 E_W, E_Q, and E_I values for the edge- and the cloud-based configurations considering a connectionless application with T_I equal to 750 and 1 000 milliseconds, RTT^C values ranging from 50 to 300 milliseconds, T_{ELAB} equals to 150 milliseconds, RTT^E equals to 40 milliseconds, and 16 000 bytes for both B_{TX} and B_{RX}. 95
7.5 The setup used to compute the ρ values of Figure 7.8. 96
7.6 The setup used to compute the ρ values of Figure 7.9. 97
7.7 The setup used to compute the ρ values of Figure 7.10. 98
7.8 The setup used to compute the ρ values of Figure 7.11. 100
7.9 The setup used to compute the ρ values of Figure 7.15. 105

8.1 δ^{k*} values obtained with the three different probing methods 124

A.1 The technical characteristics of the devices used to assess the computational load of MECPerf. 130

C.1 CRediT – Contributor Roles Taxonomy 139

Chapter 1

Introduction

Over the past decade, the amount of data generated by mobile devices has seen a significant growth, and this trend is expected to endure for the next years (Ericsson Mobility Report, 2021). This increasing amount of data transmitted, combined with the growing number of web services provided, has pushed the design of the Multi-access Edge Computing (MEC) architecture, which basically extends the centralized cloud architecture, distributing computing and storage capabilities at the edges of the network.

1.1 Multi-access Edge Computing

MEC (ETSI, 2018; Kekki et al., 2018) is a network paradigm that enables the shifting of storage and computing capabilities from centralized remote clouds to points of the network closer to end-users (Campbell, 2019; Pan and McElhannon, 2018). Edge servers may be placed on Base Station (BS) (Guo et al., 2018) or a few hops from them (i.e., on Internet of Things (IoT) gateways (Bellavista et al., 2019) or on other devices belonging to the network operator (Liu et al., 2020a)).

This kind of network architecture introduces several benefits when compared with the classical cloud architecture (Campbell, 2019; Taleb et al., 2017; Filippou et al., 2020). Firstly, edge servers provide lower latencies than cloud servers. This is particularly important for applications that have particularly stringent latency requirements such as online gaming (Zhang et al., 2019), Augmented Reality (AR) (Braud et al., 2017a), Virtual Reality (VR) (Braud et al., 2017b), and connected vehicles communications (Giust et al., 2018; 5GAA, 2017). Secondly, the use of edge servers allows obtaining higher throughput, providing several advantages to applications that need to send or receive considerable amounts of data, such as real-time video analytics (Wang et al., 2018) and other multimedia applications (Qadri et al., 2020). Thirdly, when a client is communicating with an edge server, data packets do not need to cross the public Internet. This has implications for the privacy of

user's data(Caprolu et al., 2019), which is a critical aspect for industrial 4.0 applications (Zheng and Cai, 2020; Krupitzer et al., 2020; Industrial Internet Consortium, 2019; 5G-ACIA, 2019; Reznik et al., 2018) or for other applications based on sensitive data (i.e., healthcare applications (Nauman et al., 2020), facial/behavior recognition (Wang et al., 2017), and so on). Furthermore, since the adoption of edge servers ensures that data no longer has to leave the operator's network, the use of edge servers can diminish congestion within the core Internet network.

It should be also remarked that in this type of architecture the destination servers can be chosen among multiple servers placed in both edge and cloud networks. Finding server placement and orchestration strategies that are local and globally optimal is a convoluted optimization problem. Such optimization problems should consider multiple factors such as application requirements, network load, energy consumption, and privacy constraints. However, current network performance measurement strategies are designed for end-to-end applications running on cloud servers. In fact, in a MEC architecture, the client can interact with a myriad of servers located in different parts of the infrastructure. For this reason, assessing the performance of relevant network segments separately may be more beneficial than collecting the performance of the entire path. For example, by considering the metrics between the client and the BS and between the BS and a target server, it is possible to isolate the performance of the wireless link from that of the wired part of the path. Using this type of knowledge, an orchestrator may take application placement decisions without considering the wireless link, which is common to all the possible paths between the client and both edge and cloud servers. Additionally, by combining network metrics collected on different network segments, it is possible to estimate the performance between the clients and both edge and cloud servers without repeating measures on some segments (e.g., the wireless link can be measured a single time). Consequently, they do not provide measurement methodologies that are appropriate for the systematic collection of metrics within the MEC environment. Consequently, they do not provide measurement methodologies that are appropriate for the systematic collection of metrics within the MEC environment. For example, runtime network metrics calculated between the client and the application server can be used to assess whether the current deployment continues to meet the application requirements or if some changes in the network conditions require to relocate the application server. Alternatively, runtime metrics collected between the client and multiple application servers can be compared to assess whether a different placement is able to introduce additional benefits (e.g, diminishing the workload on the edge node or decreasing the energy consumption). Similarly, metrics calculated between edge and cloud servers can be used to assess the cost of propagating data. Then, with the objective of gathering network performance metrics for the network segments that connect Terminal Nodes (TNs) and application servers, running on

both MEC and cloud networks, and the network segments that connect edge and cloud servers a measurement tool called MECPerf was developed.

In the literature, several works attempt to study how the use of a MEC infrastructure can have an impact on the performance of applications running on TN. In fact, offloading part of the operations to edge (or cloud) nodes can bring different benefits to constrained devices such as smartphones, IoT nodes, and other similar devices (Srinivasa et al., 2019). For example, the transfer of complex tasks allows to run computationally demanding applications on devices that normally would not have enough computational capabilities to run them (Hao and Wang, 2019; Zahed et al., 2020; Li et al., 2021a). Additionally, a node with sufficient computing capacity may still choose to transfer a task in order to save power and extend its battery life. In fact, it is important to note that constrained devices are generally also battery-powered. Therefore, the reduction of energy consumption is often a non-negligible requirement, while communication is an energy-demanding process. Consequently, a trade-off between task offloading and the optimization of communication operations is required. Energy consumption optimization strategies in the MEC environment are still an open research problem (Jiang et al., 2020) that could affect not only the lifetime of the device but also the environmental impact of the entire system. The second part of this thesis was devoted to evaluating how communication latency can affect the energy consumption of a constrained Long Term Evolution (LTE) end-device.

1.2 Understanding the paths towards remote cloud servers

Finally, the last part of this thesis is aimed at studying network paths between TNs and remote application services. A widely adopted mechanism for identifying hosts belonging to the path that connects two hosts is the one based on traceroute. The original implementation, developed by Van Jacobson, relies on UDP probes sent to a target host, while other variants have been added in the following years. Basically, the traceroute procedure is based on the use of IP packets, manipulating their Time To Live (TTL) header fields. The procedure is based on the assumption that a packet should be dropped whenever it is received by a router with TTL equal to 1. In addition, an ICMP Time Exceeded(Postel, 1981) message should be sent to the sender. At this point, the sender can correlate the IP address of the source router, obtained from the ICMP message, with the TTL value that elicited the ICMP message. Instead, different mechanisms are adopted to detect the target machine, depending on the version of the traceroute used. For instance, the target hosts reply to UDP-based probes using ICMP Port Unreachable messages, to TCP-based probes using

a TCP SYN + ACK or a TCP RST packets, and to ICMP-based probes using ICMP Echo Reply messages.

Investigating the topology of the Internet is a challenging task(Luckie et al., 2008a) and numerous papers in the literature have attempted to use traceroute-based mechanisms to carry out this type of investigation (claffy et al., 2009; Gregori et al., 2018; Madhyastha et al., 2006; Shavitt and Shir, 2005; Luckie et al., 2008b). For example, some routers may not send any ICMP packets, or they may only send them only under specific conditions (e.g., when their workload is low or when a given request rate was not exceeded). Additionally, firewalls, traffic shapers, and other similar machines are capable of identifying traceroute traffic using deep packet inspection techniques. Therefore, in some domains, probes and ICMP Time Exceeded messages can be eliminated according to completely arbitrary policies, even if these practices are prohibited within the European Union as they conflict with net neutrality principles(Guidelines on the Implementation by National Regulators of European Net Neutrality Rules , 2016). In any case, if packets are dropped at some point, the last part of the Internet path will be undetectable by traceroute-based mechanisms. Therefore it is important to implement mechanisms capable of circumventing these classification and filtering mechanisms. With this objective, a traceroute variant has been conceived. Basically, the main idea behind this mechanism is based on alternating probe messages and legitimate application data. In other words, the probe messages are sent together with the application data using an already established TCP connection. The goal of this mechanism is to conceal the probes, making them appear as genuine application data and bypassing possible classification and filtering machines.

1.3 Contributions

In the following the contribution of this thesis will be outlined:

- A tool for the collection of network-related performance metrics in a MEC environment was designed. The tool, called MECPerf, was used during an extensive measurement campaign that involved various measurement methods. Experiments were carried out in a testbed participating in a Fed4Fire+ project funded by the European Commission within the H2020 program. Finally, the dataset containing the collected metrics has been made publicly available.

- Two analytical models aimed at estimating the energy consumption of TNs operating within a MEC environment using LTE connections were designed. The former is an analytical model specifically designed for connectionless applications. Instead, the latter is a hybrid model designed for evaluating the consumption of connection-oriented applications. Indeed, the hybrid model

integrates the ideas behind the analytical model with a set of experimental measurements, in order to adapt to the increased complexity of TCP connections. These two models have been used to compare the energy consumption of a TN communicating with an edge server and the energy consumption of a TN communicating with a cloud server. Results show that in many cases an edge-based placement of the server is more energy efficient.

- A traceroute variant operating at the application level has been proposed. The tool called camouflage traceroute (camotrace) aimed to study the network path that connects a source node and a destination Web server using as probes HTTP requests and acknowledgment packets sent over previously established TCP connections. Fundamentally, the purpose of this tracerouting mechanism was to persuade firewalls to classify probes as normal application traffic, avoiding filtering events.

1.4 Outline

This thesis is organized as follows.

First, Chapter 2 summarizes the relevant literature concerning the comparison of performance in edge- and cloud-based environments, analyzing offloading strategies, publicly available datasets containing edge computing data, benchmarking tools for edge computing platforms, and emulators and simulators of edge network systems. Then, some works that attempt to assess the energy consumption of LTE TNs will be discussed. Finally, this chapter will conclude with an overview of works based on the study of the topology of the public Internet.

Chapter 3 will briefly illustrate the MEC architecture. Then the types of metrics that can be collected and the main idea behind their collection will be explained.

Chapter 4 will present MECPerf. First of all, the software modules that compose MECPerf and their integration within a MEC architecture will be explained. Then, MECPerf was used during an experimental campaign aimed at collecting network metrics according to different methods in a MEC environment. The design of the experiments and the metrics collected will be debated in Chapter 5. Finally, some considerations regarding the computational load of MECPerf will be given in Appendix A.

Chapter 6 will illustrate the MECPerf Library, an Application Programming Interface (API) developed to allow programmers to easily access the MECPerf dataset, which contains all the metrics collected during the European project. This Chapter will conclude with a simple example of the usage of the library.

Chapter 7 will investigate how an edge-based architecture can affect the energy consumption of an LTE TN operating in a MEC environment according to a request-

response schema. Precisely, first I will provide a Finite State Machine (FSM) model of the LTE interface. Then, analytical and hybrid energy-consumption models will be provided. The two models will be used to assess the energy consumption of both connectionless and connection-oriented applications. Finally, the Chapter will conclude with a comparison of the consumption of a TN interacting with both edge and cloud servers, considering multiple amounts of data transmitted, multiple application periods, multiple server-side computation delays, and multiple RTTs.

Chapter 8 will illustrate the implementation details of camotrace, debating both the HTTP- and the ACK-based probes methods. Then, after a validation measurement campaign, the discovery capabilities of camotrace will be compared with those of the TCP traceroute.

Finally, Chapter 9 will conclude this thesis.

Chapter 2

Background

In the last few years, the edge computing network paradigm(Bellavista et al., 2019) has received considerable attention in different research fields, such as computation offloading modeling (Lin et al., 2020a), resource management (Hong and Varghese, 2019), communication (Porambage et al., 2018), and service orchestration (Taleb et al., 2017).

2.1 Compare Edge- and cloud-based performance

Some works tried to assess the advantages introduced by edge computing elements quantitatively.

For example, Vilela et al. (2019) presented a 3-layer architecture for a Fog Computing system in a healthcare environment. The proposed architecture was formed by a sensor network, a middle (fog) layer, and a remote cloud infrastructure layer. The results were compared with a traditional cloud computing solution. As a result, the authors found that the energy consumption and the communication latencies can be reduced by executing the tasks into the fog layer. Another comparison between edge computing and traditional cloud-based architectures was presented by Hu et al. (2016a). The benefits introduced by the edge-computing infrastructure have been evaluated regarding response time and energy consumption metrics. The experiments were conducted considering the offloading of both processing-intensive and latency-sensitive applications and using different network configurations and access technologies (i.e., Wi-Fi and cellular technologies). The results showed that all the considered metrics enhanced when computations were offloaded to the edge nodes.

SOUL (Jang et al., 2016) is an application framework intended for Android devices. It aimed to aggregate multiple sensors on different devices transparently to the programmer, moving on edge nodes the computational load of sensor-based applications. The SOUL architecture is based on two main blocks called SOUL en-

gine and SOUL Core. The SOUL Engine is executed on mobile devices and was in charge of aggregating virtual sensors and virtual actuators. Instead, the SOUL Core is implemented in edge nodes, and it is in charge of applying access control policies, storing into a database the data collected from sensors, and distributing the load of constrained devices. Results based on micro-benchmarks have shown that overhead, scalability, and power consumption metrics achieved by SOUL are better than those achievable by standard approaches.

The adoption of an edge-based solution in vehicle-to-pedestrian systems has been studied by Nguyen et al. (2020). In this type of environment, collision detection algorithms can use contextual information generated by cars and users' smartphones to prevent dangerous situations. The study evaluated the cost and the benefits of offloading tasks from the smartphone to the edge, considering battery consumption and processing time. Then, other works concerning the benefits of migrating applications from remote clouds to edge nodes in a Vehicle to Everything (V2E) scenario can be found in (Napolitano et al., 2019), (Emara et al., 2018), and (Quadri et al., 2022). Precisely, Napolitano et al. (2019) presented a warning system capable of gathering context information from all the road users, notifying vulnerable users (e.g., pedestrians, bicyclists, and other non-motorized vehicles) whenever a potentially dangerous situation is detected. The results showed that the introduction of the edge can reduce the time needed to propagate information. Similarly, Emara et al. (2018) studied the impact of a MEC infrastructure on end-to-end latency metrics. The simulation-based results demonstrated that the introduction of edge nodes can reduce end-to-end latency by up to 80% when compared to the conventional cellular network architecture. Finally, Quadri et al. (2022) studied the feasibility of migrating platooning control applications from vehicles to edge hosts considering the impact of delay metrics, packet losses, coverage holes, and the scalability of the system.

To conclude, De Vita et al. (2021) show DeepLeaf, an Artificial Intelligence (AI) application based on Convolutional Neural Networks (CNN) and deployed on edge nodes. To fulfill the hardware constraint of edge nodes, the application uses a dynamic K-Means-based compression algorithm to reduce the memory footprint of its Deep Neural Networks (DNN) model.

Offloading strategies

In recent years, a large number of articles concerning task offloading strategies have been presented in the literature. For example, Xu et al. (2019a) shows an offloading strategy for deep-learning applications. The offloading decisions are based on a heuristic algorithm aimed at minimizing the offloading transmission delay and maximizing the number of offloaded tasks. Instead, iTaskOffloading (Hao et al., 2019) was another offloading decision scheme for AI applications. Differently from

the previous work, iTaskOffloading supports the offloading of tasks from TNs to both edge and cloud nodes. Offloading decisions are computed using a cognitive engine that considers a multiplicity of factors (i.e., latency, computing, and storage requirements) to minimize the time needed to execute the task and the energy consumption.

Cheng et al. (2020) presented an offloading strategy for Automatic Speech Recognition (ASR). The tool provides two different offloading methods. In the first case, the TNs can offload the task to the cloud, which executes the task entirely. Otherwise, they can offload the task to an edge node, which only extracts the features of the audio during a pre-processing phase. Then, the extracted features were sent to the cloud for further processing. Instead, Yang et al. (2020) presented EdgeRNN, a speech recognition tool developed to run on edge devices. EdgeRNN performed speech recognition tasks by combining 1-Dimensional Convolutional Neural Networks (1-D CNN) with Recurrent Neural Networks (RNN) based on spatial and temporal features.

Offloading strategies based on deep reinforcement learning solutions have been studied by Rahman et al. (2020) and Nath and Wu (2020). Precisely, Rahman et al. (2020) presented an offloading mechanism that minimized the overall latency, considering both power consumption and computing capabilities constraints. The results, obtained through simulation, showed that the proposed method can achieve lower delays than other well-known benchmarking schemes. Instead, Nath and Wu (2020) presented an offloading strategy for cache-assisted MEC systems. Fundamentally, in this type of system, the edge nodes can store popular task data in their caches. This reduces both the latency and the energy required to compute offloading operations as popular tasks have not to be uploaded to edge nodes.

Zhao et al. (2020), Liu et al. (2020b), and Xu et al. (2019b) analyzed privacy-related problems. The analysis conducted by Zhao et al. (2020) studied an offloading strategy based on artificial neural networks and genetic algorithms. The model was computed under the assumption of having an attacker capable of monitoring the offloaded tasks by hacking the edge nodes. This vulnerability could lead to several privacy-related problems. For example, by monitoring the frequency at which the user offloads his tasks, an attacker may be able to discover the user's identity. Instead, Liu et al. (2020b) and Xu et al. (2019b) investigated about the secrecy of the offloaded data. The former paper(Liu et al., 2020b) showed a framework, called DataMix, strived to guarantee the privacy of the user data. DataMix was based on the assumption that cloud servers should be considered malicious nodes unable to meet privacy constraints, while edge nodes can be trusted. Consequently, the malicious clouds should not receive user data in clear. However, they cannot be fully decommissioned as edge nodes with poorer capabilities may not be sufficient to execute users' tasks. Then, to solve the problem, an inference process composed of

three phases was proposed. During the first phase, raw inputs were sent in clear to a trusted edge node. At this stage, the trusted edge is able to mix together multiple different inputs in order to anonymize them. During the second phase, the mixed data was then sent to a classifier running in the untrusted cloud. Finally, during the third and last phase, the aggregated output is post-processed on an edge node in order to generate a result for every single original input. Instead, an offloading decision strategy based on a privacy entropy model was studied by Xu et al. (2019b).

Open Data on Edge Computing

In the following, some open datasets of edge applications will be discussed.

Toczé et al. (2020) used a prototype testbed to conduct a set of experiments aimed to study Mixed Reality (MR) applications in an edge computing environment. The data collected during the experiments was made publicly available for further studies. Similarly, Toczé et al. (2020) publicized workload traces regarding speech-based and MR edge applications, while the dataset in (Rashed, 2019) contains data related to both the placement and the execution of functions in a mixed cloud-edge cluster. Similarly, the dataset in (Rashed and Rausch, 2020) published Machine Learning (ML) applications traces collected using a real testbed and several functions. Then, the execution time and the time required to transmit the data were published for each considered function. Furthermore, the collected traces were used to evaluate different container scheduling strategies for serverless edge computing (Rausch et al., 2021). A different dataset containing statistics about the edge-based architecture can be found in (Apostolis, 2020). Instead, state machine replication performance was analyzed in (Yan et al., 2020). Also for this case, the latency traces obtained were made publicly available. However, differently from the previous works, these traces belong to six different remote clouds and did not include any edge metrics. Finally, Lin et al. (2020b) presented a different approach based on generative adversarial networks. Basically, the tool called DoppelGANger aimed to generate synthetic datasets and required a minimal level of knowledge.

Benchmarking Edge Computing Platforms

EdgeBench (Das et al., 2019) is a benchmarking framework for serverless edge computing systems. The paper shows a comparison between two commercial edge infrastructures (i.e., AWS Greengrass and Azure IoT Edge). Instead, a different benchmarking framework called DeFog was presented by McChesney et al. (2019). DeFog aimed to compare the performance of fog- and cloud-based platforms. Benchmarking operations were performed using various containerized applications, including speech-to-text, real-time face recognition, and video streaming applications. Applications can be evaluated considering three different execution modes (i.e., a cloud-

only, a fog-only, and a mixed cloud-fog mode). In addition, the framework collected heterogeneous metrics such as the latency observed, the number of CPUs/-cores, the execution time, and the number of bytes transferred. Instead, a different work (Yeganeh et al., 2020) studied the performance of cloud providers by considering three distinct connectivity options. Precisely, connections based on (i) the public Internet, (ii) private Cloud connectivity, and (iii) third-party private providers had been taken into account. However, the active metrics collected by this framework did not focus specifically on the edge concept.

Finally, the lack of publicly available benchmarks was highlighted by McChesney et al. (2019). The authors stated that this could make comparing fog/edge- and cloud-based solutions more difficult.

Edge Computing Emulators and Simulators

Applications and network protocols can be tested through the emulation of an edge network, while simulators are more suitable for evaluating, and possibly tuning, the impact of both design choices and operational parameters. In the following, some edge emulators and simulators will be analyzed.

openLEON (Fiandrino et al., 2019) is an emulator for edge data centers. More in detail, openLEON uses srsLTE (Gomez-Miguelez et al., 2016) to emulate the wireless segment that connects TNs and the edge data centers, while Containernet (Peuster et al., 2016) is used to emulate the data center. However, it is relevant to note that openLEON is not a fully-software solution as it includes some hardware devices.

Fernández-Cerero et al. (2020) presented a simulator designed for experimenting with orchestration strategies for clusters of edge nodes. The simulator supports centralized, distributed, and hybrid orchestration models. Additionally, clusters could be powered on and off independently from each other, using different strategies based on efficiency and performance metrics at both network-/cluster-level. FogNetSim++ (Qayyum et al., 2018) is an OMNeT++-based simulator. It allowed researchers to evaluate task scheduling algorithms, and it considers a multiplicity of factors such as fog nodes utilization, Service Level Agreements (SLAs), and handover events. Instead, iFogSim (Gupta et al., 2017) aimed at evaluating the performance of IoT fog applications, considering multiple cost metrics (e.g., power consumption, latency, network congestion, etc.). In addition, different placement strategies have been provided. EdgeCloudSim (Sonmez et al., 2018) is a CloudSim-based simulator. It was intended for the evaluation of Virtual Machines (VMs) orchestration strategies, resource management, and task offloading. Finally, the YAFS (Lera et al., 2019) simulator was aimed to study IoT fog scenarios. YAFS allows researchers to define network topologies, even complex ones, by importing from CAIDA and

BRITE. However, the defined network topology is characterized by links with fixed bandwidth and latency metrics.

The key characteristics of both emulation and simulation platforms listed in this Section have been summarized in Table 2.1. Observing the table it is evident that most of the described tools have relatively simple network models. For example, the transport, network, data-link, and physical layers are not included in the model for most of the tools analyzed, while the communication is characterized by simple (and often deterministic) bandwidth and delay models. However, links characterized by realistic bandwidth and latency metrics are decisive prerequisites for properly simulating an edge network. For example, wireless links connecting TNs and edge nodes could introduce intermittent distortions that could provoke significant changes in the observed metrics, while this variability has a smaller effect in a cloud scenario characterized by longer-haul links with higher latency. In the literature, there are some simulators characterized by more complex network models. For example, ECSim++ (Nguyen and Huh, 2018), a simulator based on OmNET++, simulates the network stack in its entirety. However, some parts of the MEC infrastructure may still not be easy to model (e.g., the backhaul connections and the presence of cross-traffic).

Open problems and filling the gap

As debated in this Section, a number of articles compared the performance of edge- and cloud-based solutions. However, such works sporadically took into account the effect of different network conditions. Moreover, as explained by Kolosov et al. (2020) most of the edge-based datasets are based on computing aspects and did not represent the MEC environment realistically. In the literature, other works take into account network parameters. For instance, Harutyunyan et al. (2019) investigated the problem of finding a latency-aware placement of service function chains. The model was computed using Integer Linear Programming (ILP) techniques under the assumption of knowing the capacity of the links between edge nodes and the centralized cloud. Instead, for hierarchical edge networks, bandwidth information of relevant network segments was considered by Tong et al. (2016). These papers highlight the importance of collecting network metrics at runtime in order to ensure to the end-users an adequate Quality of Experience (QoE), especially when applications with strict latency or high bandwidth requirements are considered. Furthermore, these works show how important it is to have public datasets built upon real experiments. In fact, for example, they make possible to compare different edge applications under the same scenario, or they can be used to evaluate the performance of an application when there is no real edge infrastructure at all.

To fill this gap, an extensive measurement campaign strived to collect network performance metrics in a MEC environment has been performed. The tool and the

experimental setups adopted will be described in Chapters 4 and 5, respectively. Moreover, the dataset containing all the collected metrics was distributed on Zenodo(MECPerf experimentation results, 2020).

2.2 Evaluating the energy consumption of a Terminal Node in an edge-computing system

Energy evaluation of task offloading strategies

A performance evaluation of a three-tier fog network for IoT applications has been presented by Sarkar et al. (2018). The proposed schema considered service latency, power consumption, and CO_2 emission metrics. In particular, the overall power consumption has been split into the power consumed to forward packets, perform computation, store data, and migrate applications to the cloud. In contrast, latency has been divided into the time needed to transmit and process the data.

Pei et al. (2020), and Zhang et al. (2018) investigated upon task offloading mechanism for hierarchical edge computing networks. Basically, in this sort of network, the edge servers can be deployed on both Small Base Stations (SBSs) and Macro Base Stations (MBSs). Generally, SBSs are placed nearby of TNs and are distinguished by limited computing capabilities and low latencies. So they are more suitable for computing tasks with strict latency requirements. Conversely, MBSs are generally placed farther than SBSs, but they are also more powerful. This means that they could be more appropriate for executing tasks with heavy computational and loose latency requirements. In the former paper (Pei et al., 2020), tasks can be partially offloaded to both SBSs and MBSs. The offloading strategy aimed to compute the optimal workload placement characterized by the minimum energy consumption and constrained latency. Instead, the algorithm proposed in the latter paper (Zhang et al., 2018) aimed to compute the optimal computation offloading that minimizes the weighted sum of energy consumption and the task latency. Furthermore, the model also considers the residual amount of energy stored in the battery of the TN.

The work presented by Hu et al. (2016b) tested multiple applications with different computational and communication requirements, considering a smartphone with LTE and Wi-Fi connectivity capabilities. The results showed that an edge scenario can diminish both the energy consumption and the response delay when compared with the local execution of tasks. Conversely, when far cloud servers characterized by high RTT are considered, the local execution is demonstrated to be the best approach in terms of both response delay and energy consumption.

Finally, other works concerning optimal energy-aware offloading schemes in a MEC environment can be found in (Mazouzi et al., 2019) and (Li et al., 2021b).

Energy Models for an LTE Interface

An overview of the work concerning models for a generic LTE network interface will be discussed in the following. These models make feasible to analytically estimate the energy consumption of network transmission, a problem that would otherwise require expensive and complex hardware devices such as the monsoon power monitor (Monsoon Power Monitor, 2022).

In the literature, several works described the LTE interface as a 4-state FSM (Huang et al., 2012; Chen et al., 2015). For example, the former work (Huang et al., 2012) has presented a methodology to infer the operational parameters of the FSM. A subsequent validation phase showed that the model and the inferred parameters exhibited an error rate below 6%. Instead, the latter work (Chen et al., 2015) compared the energy consumption of Wi-Fi, LTE, and 3G interfaces. Precisely, the CPU, GPU, and screen consumption have been modeled using *utilization-based* models, which assume that a component's energy consumption is strictly related to its utilization level. Instead, the three network interfaces were modeled using three different FSMs. Finally, the overall energy consumption has been computed using real traces belonging to 1520 smartphones from 56 countries. This approach was used to assess the relationship between users' behaviors and energy consumption. As a result, the authors found that Wi-Fi and cellular interfaces required approximately 7.0% and 24.4% of the overall energy, respectively.

Fundamentally, some states of the LTE FSM are based on the *Discontinuous Reception* (*DRX*) *mechanism*, which basically consists in the interleaving of sleeping and wake-up periods. The DRX mechanism is fundamental for lowering the energy consumption of the interface, however, it must be not too aggressive to meet transmission latency constraints. Therefore the balancing of DRX parameters is a crucial problem widely investigated in the literature. For example, Tseng et al. (2016) used an approach based on Markov chains to evaluate the impact of DRX parameters on both power-saving and wake-up delay constraints. Instead, Zhou et al. (2008) and Mehmood et al. (2019) used semi-Markov processes. In particular, the latter work studied the DRX mechanism problem for the Machine Type Communication (MTC) environment. Instead, the work in (Brand et al., 2020) used two ML algorithms to predict the optimal sleeping intervals. The first algorithm was based on supervised learning, while the second one was based on reinforcement learning. Essentially, the two algorithms aimed to turn off the interface as soon as possible while minimizing the number of transmissions lost from the BS.

Open problems and filling the gap

The reduction of energy consumption represents an important aspect that is gradually becoming more and more relevant. Most of the paper in the literature shows

how task offloading strategies can be used to reduce the energy consumption of edge and cloud networks. This kind of optimization can lessen both the economic cost and the environmental impact of the server infrastructure. However, the energy consumption of TNs is generally not considered in these works. It should be remarked that other numerous works aimed to assess the energy consumption of terminal devices connected through LTE connections. However, these works do not involve the MEC environment, and generally, they do not exhaustively explore the relationships between the network metrics and the energy consumption of TNs.

To fill this gap, Chapter 7 will present two models used to compute the energy consumption of an application running on a TN operating in a MEC environment. Therefore, the relationship between network metrics and energy consumption will be examined considering different application and network parameters.

2.3 Evaluating the paths towards the cloud

In the past 15-20 years, multiple works tried to study the Internet topology (Donnet, 2013). Most of these works are based on traceroute measurement campaigns, considering multiple abstraction levels (Cheswick et al., 1999). For example, Keys et al. (2013) and Keys (2010) used the alias resolution techniques to evaluate the path at the router level. Instead, Chang et al. (2001) used IP-to-AS mapping strategies to assess paths in terms of AS. Moreover, traceroute measurements have been adopted by multiple Internet mapping projects, such as CAIDA Ark (claffy et al., 2009; ARK, 2019), iPlane (Madhyastha et al., 2006), DIMES (Shavitt and Shir, 2005), RIPE Atlas (RIPE Atlas, 2019), M-Lab (Mlab, 2019), and Portolan (Faggiani et al., 2014, 2012; Gregori et al., 2013).

However, in the last years, traceroute implementation has led to several problems. In fact, the presence of load balancers, firewalls, or other middleboxes (Augustin et al., 2006; Detal et al., 2013) can bias the outcome of Internet mapping measurements. For example, load balancers can operate following per-destination, per-flow, or per-packet strategies. Basically, a per-destination load balancer forwards all packets with the same destination on the same route. A per-flow load balancer sends all packets with the same flow on the same route. Finally, a per-packet load balancer disseminates the packets on multiple paths, regardless of their destination or their flow. These three types of load balancers can lead classic tracerouting algorithms to not detect some nodes (i.e., when packets with the same destination are always forwarded on the same route) or to detect false paths (i.e., when probes with consecutive TTLs are forwarded to different paths). Paris traceroute (Augustin et al., 2006, 2007; Vermeulen et al., 2018) and its multipath detection algorithm (MDA) have been implemented to overcome this limitation. To be more precise, Paris traceroute manipulates the probe's header at the transport level, while the IP header is kept

fixed. This means that all the probes belong to the same flow and follow the same path in presence of a per-flow load balancer. However, the incoming ICMP packets still contain enough information to link each probe with the correspondent elicited ICMP packet. Instead, Tracebox (Detal et al., 2013) is a tool capable of discovering the presence of middleboxes along the path between two hosts. In other words, it can be identifying non-destination machines operating above the network level. Marchetta and Pescapé (2013) studied the problem of identifying hidden routers along a path. Basically, a hidden router is a device that forwards the incoming packets without decreasing their TTL values, making impossible for traditional tracerouting mechanisms to detect them. The developed tool, called DRAGO, solved this problem using IP probes with the Timestamp (TS) option set. From the analysis of the ICMP packets, DRAGO can detect the number of hosts that managed the TS option, estimating the number and location of hidden routers.

Moors (2004) and Huang et al. (2020) presented two strategies strived at reducing the time needed to collect tracerouting information. The former work (Moors, 2004) takes advantage of a scout packet sent to the target. The source node uses the TTL of the response to estimate the length of the path and speed up the probing phase. Two different probing methods, called raceroute and aceroute, have been proposed. Basically, raceroute tries to speed up the tracerouting process by sending the probes in quick succession. Instead, aceroute tries to reduce the number of probes by starting to explore from the target and progressively decreasing the TTL value. The process is interrupted when a hop on a known path is reached. Instead, the latter work (Huang et al., 2020) is intended for massive tracerouting campaigns. The developed tool, called FlashRoute, explores the paths toward multiple destinations with a high level of parallelism. FlashRoute supports both forward probing (i.e., probing from the source to the destination using increasing TTL values) and backward probing (i.e., probing from the destination to the source using decreasing TTL values). Finally, the discovery capabilities of Flashroute were compared with those of Yarrp(Beverly, 2016) and scamper (Luckie, 2010) during an extensive experimental campaign. The results demonstrated that FlashRoute succeeded in decreasing the time needed to complete an entire scan.

Finally, Morandi et al. (2019) presented Service traceroute. The tool listens for application traffic in a passive way. Then, it automatically selects an application flow and injects its probes within the target application traffic. Service traceroute supports applications based on multiple concurrent flows and applications that adopt UDP flows. The experimental phase shows that the probes sent by Service traceroute have no effect on the performance of the target application flow.

Open problems and filling the gap

These tools discussed in this section perform complex analyses to discover the path between two hosts with high accuracy. However, they cannot bypass firewalls specifically configured to block traceroute-based applications.

To fill this gap, differently from similar tools (Sherwood and Spring, 2006), camotrace uses legitimately establishes TCP connections and HTTP-based probes to conceal its traffic, making more complex for its probes to be identified by stateful firewalls.

Table 2.1: A comparison between emulators and simulators in the field of edge computing systems. The comparison was based on network-related characteristics.

Tool	Based on	Type	Simulated/emulated devices	Network model
openLEON Fiandrino et al. (2019)	srsLTE, mininet	Emulator	Edge data center, and cloud data center	**End-devices access technology:** LTE. **Network topology:** a real smartphone connects to the eNB using dedicated hardware. The rest of the network is emulated using Containernet. **Link characteristics:** real, between user equipment and eNB, emulated according to mininet the rest. **Network stack:** real.
SPHERE Fernández-Cerero et al. (2020)	SCORE simulator	Simulator	IoT, mobile devices, independent clusters of cloudlets and clouds	**End-devices access technology:** not simulated. **Network topology:** defined by the programmer. **Link characteristics:** deterministic latency and bandwidth. **Network stack:** not simulated.
FogNetSim++ Qayyum et al. (2018)	OMNeT++, INET	Simulator	IoT, mobile devices, fog nodes, and cloud data center	**End-devices access technology:** both wired and wireless. **Network topology:** defined through a GUI or a configuration file. **Link characteristics:** according to INET models. **Network stack:** simulated physical, link-layer, network, transport, and application layers.
iFogSim Gupta et al. (2017)	CloudSim	Simulator	IoT, fog nodes, cloud data center	**End-devices access technology:** wireless **Network topology:** defined by through a GUI or a configuration file. **Link characteristics:** fixed latency and bandwidth. **Network stack:** not simulated.
EdgeCloudSim Sonmez et al. (2018)	CloudSim	Simulator	Mobile devices, edge servers, and cloud data center	**End-devices access technology:** wireless. **Network topology:** defined through an XML file. **Link characteristics:** simple queue model, empirically derived properties. **Network stack:** not simulated.
YAFS Lera et al. (2019)	Python, Simpy, and NetworkX	Simulator	IoT, fog nodes, and cloud data center	**End-devices access technology:** not simulated. **Network topology:** defined through a configuration file or imported (CAIDA, BRITE topologies). **Link characteristics:** fixed latency and bandwidth. **Network stack:** not simulated.
ECSim++ Nguyen and Huh (2018)	OMNeT++, INET	Simulator	Mobile devices, edge nodes, and clouds	**End-devices access technology:** both wired and wireless. **Network topology:** defined through a GUI or a configuration file. **Link characteristics:** according to INET models. **Network stack:** simulated physical, link-layer, network, transport, and application layers.

Chapter 3

Collecting performance metrics in a MEC environment: challenges and requirements

As previously stated, in a MEC architecture TNs can communicate with application servers placed at the edge of the network. As a result, edge servers bring lower communication latency and higher bandwidth. Moreover, edge nodes can be used to reduce the amount of traffic directed towards cloud application servers. Conversely, cloud servers generally provide higher computational capabilities. Finally, in many cases, an edge server can interact with a centralized cloud server to retrieve or propagate data. In the following, some suggestions regarding the approaches that can be used during the collection of network metrics within a MEC infrastructure and their possible usage will be provided.

3.1 The MEC architecture

Figure 3.1 illustrates the general architecture of a MEC network. In this type of network three relevant network segments can be identified. The first one is the *access-MEC* segment that connects TNs with edge servers hosted into the MEC networks. The second one is the *MEC-cloud* segment that connects edge and cloud servers. Finally, the third one is the *access-cloud* segment that connects TNs and cloud servers.

Collected metrics and possible uses

For what concerns the network performance of a MEC network, three relevant metrics can be collected: the network latency, the bottleneck capacity, and the available bandwidth. Basically, the network latency can be defined as the amount of time

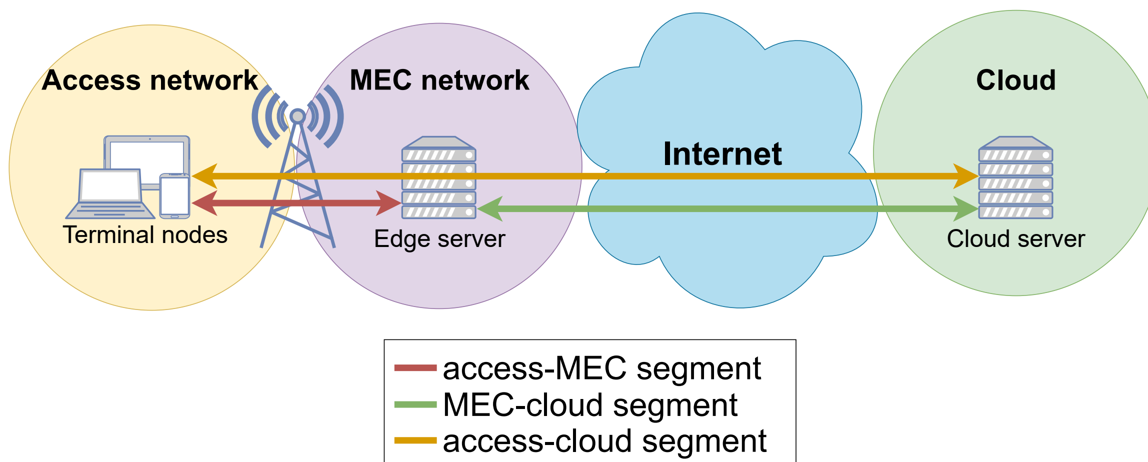


Figure 3.1: The high-level architecture of a MEC network.

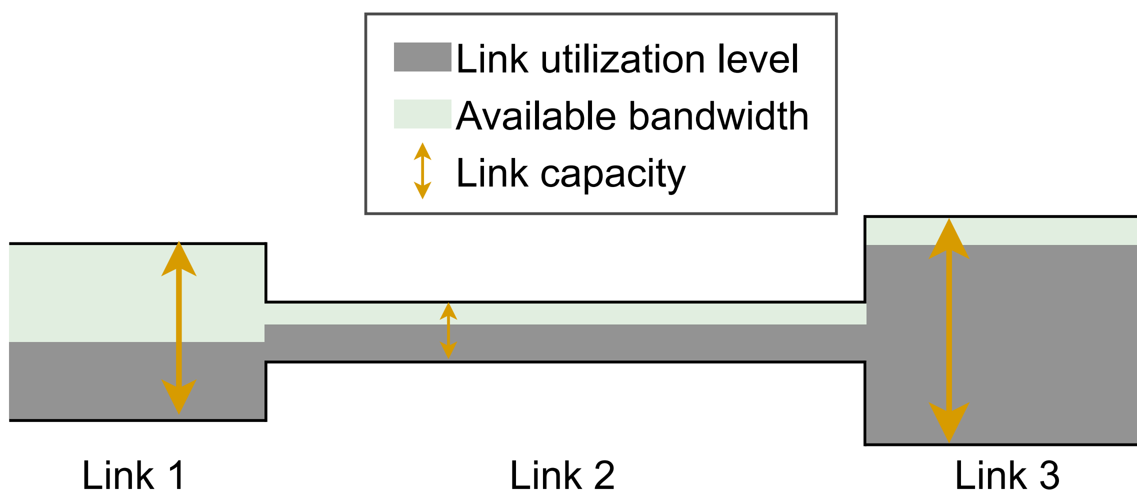


Figure 3.2: A path composed of three links with different utilization levels, available bandwidths, and capacities.

required to transfer a small amount of data across the network. Note that computing the network latency by collecting timestamps on different hosts is not a trivial task. In fact, there is no guarantee that the clocks of the sender and the receiver are synchronized. Hence, to avoid any synchronization problem, the network latency is often computed considering the RTT between the two hosts.

$$network_latency = \frac{RTT}{2} \tag{3.1}$$

Instead, both bottleneck capacity and bandwidth metrics provide a measure of the amount of data that can be transmitted over a path in a time unit. These two metrics may appear very similar to each other as they collect slightly different information. In fact, the bottleneck capacity can be defined as the maximum data rate *that can be*

achieved upon a given path. In other words, it is the smallest capacity found along the links that form the path. It describes a physical property of a sequence of links and it is ideally independent from the presence of cross-traffic. Instead, the available bandwidth can be defined as the maximum data rate *that is currently available* upon a given path. This means that bandwidth metrics are strongly affected by the utilization level of all the links along the path. Figure 3.2 shows a simple example composed of a path made up of 3 links. For each link i, the utilization level (U_i), the available bandwidth (B_i), and the nominal capacity (C_i) are displayed. Then we can compute the bottleneck capacity and the available bandwidth as follows.

$$bottleneck_capacity = min(C_i) \qquad i \in 1,2,3 \tag{3.2}$$

$$available_bandwidth = min(B_i) \qquad i \in 1,2,3 \tag{3.3}$$

Once collected these metrics can be used by orchestrators and other similar devices to take a wide range of decisions, depending on the segment on which they are collected. Network metrics calculated on the network segments that connect a TN with a candidate application server can be used to determine whether a given application can satisfy the application requirements. For instance, latency metrics computed on access-MEC/access-cloud network segments can be used to assess whether a candidate application deployment can deliver an upper bound on the communication latency, meeting the needs of a generic real-time application. Similarly, bandwidth and bottleneck capacity metrics computed on access-MEC/access-cloud network segments can be employed to determine whether the candidate deployment is able to fulfill the minimum requirements of a generic bandwidth-intensive application such as a video streaming application. Instead, the metrics collected on the MEC-cloud segment can be used to assess the cost of retrieving or propagating applications and context data from centralized clouds to the edges and vice-versa.

Measurements methods and possible uses

Network metrics can be gathered following both *active* and *passive* approaches. Basically, an active measurement method is based on the injection of traffic onto the segment that needs to be measured. Then, the injected traffic is used to collect the desired metric. As can be easily guessed, active measurement methods have the cons of generating additional traffic within the network however, they have the pros of controlling how this traffic is sent. Conversely, passive measurement methods are aimed at collecting network metrics without injecting new packets into the network or manipulating the existing ones.

MECPerf can gather network performance metrics following both active and passive approaches. In the remaining of this thesis, we will use the terms *MECPerf-active*

and *MECPerf-passive* to refer to these types of metrics. For example, the MECPerf-active measurement methods can be used to check the actual network conditions of a given network segment. This allows assessing whether a given deployment is able to meet the requirements of an application before its placement. Instead, MECPerf-passive metrics can be computed by examining traffic traces received from other devices. Therefore, to compute MECPerf-passive metrics which are representative of the current status of the network, packet sniffers or other similar devices must be positioned at strategic points on the network so that all the traffic of interest can be monitored. It is important to note that MECPerf-passive methods show two advantages with respect to MECPerf-active measurement methods. In fact, MECPerf-passive measurement methods do not increase the volume of traffic transmitted within the network. Moreover, the traffic traces can be used to gather both general network performance metrics, considering all the packets traveled within the network, and application-specific network performance metrics that consider only packets that match specific conditions (e.g, they can consider only the traffic between two target hosts). In other words, these measurement methods can be used by orchestrators to monitor the performance of a running application in real-time and eventually implement relocation strategies. Finally, it should be noted that MECPerf-passive measurement methods have no knowledge about the internal functioning mechanism of the application under analysis. Consequently, they cannot take into account periods of legitimate inactivity. So it might be worthwhile to permit an application of collecting its own metrics and sharing them with the collection system. Note that also the application can compute its own metrics following both active and passive approaches. Hereafter, we will use the terms *self-active* and *self-passive* to refer to these types of metrics. Self-passive metrics can be used jointly with the MECPerf-passive ones to conduct real-time monitoring and relocation activities as they collect the performance of a given application from different viewpoints. In fact, MECPerf-passive metrics evaluate the performance of an application from the outside and they are transparent to the application. In contrast, the self-passive metrics enable the application to provide feedback on its own performance. Furthermore, they represent the only reliable measurement methodology for the monitoring of applications based on a sporadic communication schema and characterized by the transmission of very small quantities of data.

Chapter 4

A tool for collecting network metrics in a MEC environment

In the following will be illustrated MECPerf, a tool aimed at collecting and storing network metrics in a MEC environment. MECPerf aims to collect and share network metrics in a MEC environment using multiple software components. As rapidly stated in Chapter 1, MECPerf was developed during a Fed4Fire+ project (FED4FIRE+ - MECPerf, 2022). Basically, Fed4Fire+ is a European project started in January 2017 and financed by the European Union under the Horizon 2020 program. Fed4Fire+ offers a large set of open, accessible, and reliable federated Next Generation Internet (NGI) testbed facilities (Fed4Fire+, 2017; About Fed4Fire+, 2022). Additionally, during the MECPerf project, was also conducted an extensive measurement campaign that involved MECPerf-active, MECPerf-passive, and self-passive measurement methods. The experiments were conducted using NITOS (The NITOS facility, 2022), one of the federated Fed4Fire+ testbeds. Precisely, the NITOS testbed is hosted at the University of Thessaly (Greece) and it provides three different deployments that allow experimenters to test protocols and applications considering outdoor, indoor, and office wireless scenarios. The testbed provides Software Defined Networking (SDN) capabilities and the NITOS nodes support Wi-Fi, WiMAX, and LTE communication technologies. In the following, the architecture of MECPerf will be illustrated considering its software components, their interaction, and the collected metrics.

4.1 MECPerf architecture

The architecture of MECPerf is illustrated in Figure 4.1. MECPerf is composed of a MECPerf Client (MC), a MECPerf Observer (MO), a MECPerf Remote Server (MRS), and a MECPerf Aggregator (MA). The first three components cooperate in collecting network metrics, while the last one has the purpose of storing and retriev-

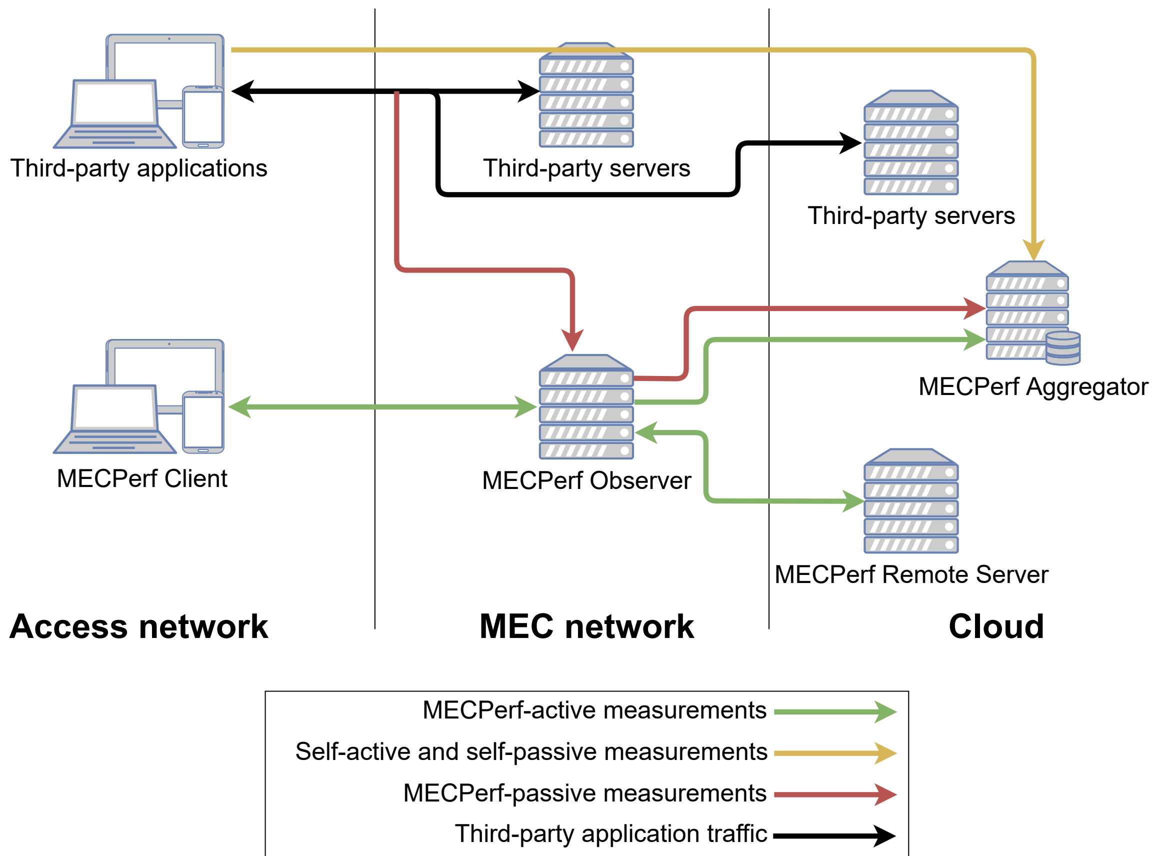


Figure 4.1: The four components composing the MECPerf architecture and the interaction between them during MECPerf-active, MECPerf-passive, self-active, and self-passive measures.

ing them. The number of components involved in each measure depends on the type of metric collected. Basically, MECPerf supports MECPerf-active, MECPerf-passive, self-active, and self-passive measurement methods. MECPerf-active measurement methods are intended to measure the network performance independently from the application executed on TNs, and they are based on additional traffic injected into the network. MECPerf-passive measurement methods aim to compute network metrics using traffic traces produced by third-party components. This method is intended for measuring the network metrics of a specific application in a transparent way. Finally, self-active and passive measurement methods allow third-party applications to store into MECPerf their self-computed metrics. More details about the implementation of the measurement methods will be addressed in the following.

The MECPerf software components

The MECPerf Client (MC)

The MC is hosted in the access network, and it interacts only with the MO in order

to collect MECPerf-active metrics for the access-MEC segment. Note that, due to its location, the MC can be executed on both dedicated devices and user-owned TNs.

The MECPerf Observer (MO)
The MO resides on the MEC infrastructure of a network operator, and it interacts with all the other components to collect and store both MECPerf-active and MECPerf-passive metrics. Precisely, during MECPerf-active measurement methods, the MO interacts with the MC and the MRS to assess the performance of the access-MEC and the MEC-cloud segments, respectively. Instead, during MECPerf-passive measurement methods, the MO uses traffic traces to compute the performance of a specific target application. Finally, the MO sends both MECPerf-active and MECPerf-passive metrics to the MA for storage.

The MECPerf Remote Server (MRS)
The MRS resides in the cloud network. The MRS is involved only during MECPerf-active measurement methods, interacting with the MO to compute MEC-cloud metrics like the MC case.

The MECPerf Aggregator (MA)
The MA is the component entitled to collect, store, and distribute the network metrics collected from different components with different points of view. For both MECPerf-active and MECPerf-passive measurement methods, the metrics are received from the MO, while self-active and self-passive measurement metrics are directly obtained from the third-party application that computes them. Finally, it should be noted that the MA is never involved in any measure. This implies that it can conceptually reside in any part of the network.

In this Section, we have seen how the MECPerf component can be used to collect network performance metrics in a classical MEC architecture. However, it should be noted that the architecture of MECPerf is generic enough that it can be employed to measure the performance of different types of networks. For example, the MO can be executed on hosts belonging to commercial edge networks such as those offered by Cloudflare (Cloudflare - Security and innovation at the network edge, 2022), Amazon (AWS for the Edge, 2022), and other similar content providers. Furthermore, the MO can be used also in a non-MEC scenario. For example, in this latter case, the MO can be run in the cloud, near the MRS, to measure the performance between different VMs hosted in the cloud(Liu et al., 2019).

Collected measurement types

As stated earlier, MECPerf uses four different measurement methods to collect network metrics. The first type of metric is based on *MECPerf-active* measurement methods, used to collect access-MEC, access-cloud, and MEC-cloud network segment's performance metrics without considering the specific application that needs to be run. The second measurement mode is based on *MECPerf-passive* measurement methods. These methods are designed to measure network metrics considering the traffic generated from the application. In other words, they can be used to monitor the performance of a running application in real-time. Finally, the last two types of measurement methods are based on network metrics computed by the application itself following both *self-active* and *self-passive* approaches and shared with the collection system. Note that both MECPerf-passive and self-passive metrics can be used to conduct real-time monitoring activities and eventually relocate the application dynamically. However, these measurement methods collect the performance of a given application from different viewpoints. In fact, MECPerf-passive metrics evaluate the performance of an application from the outside and they are transparent to the application. In contrast, the self-passive measured metrics enable the application to provide feedback on its own performance.

MECPerf-active measurement methods

MECPerf supports four different types of MECPerf-active measurement methods: TCP bandwidth, UDP bottleneck capacity, and both TCP and UDP latency. Additionally, metrics are collected for the two communication directions. For MECPerf-active *uplink* measures, additional traffic is injected from the MC to the MO and from the MO to the MRS. Instead, the injected packets go from the MO to the MC and from the MRS to the MO during MECPerf-active *downlink* measures.

The relevant interactions between the three components during a MECPerf-active measure are shown in the sequence diagram of Figure 4.2. The MC always initiates MECPerf-active measures. At the beginning of each measure, a *request measurement* message is sent from the MC to the MO. The request specifies the metrics that have to be collected and other measurement-specific parameters. Then, the MC and the MO interact to measure the access-MEC segment. At this point, the MO uses the MRS to repeat the measurements procedure on the second segment. Finally, the MO collects the raw measurement data, sending them to the MA for storage. For any measurement method, MECPerf performs up to three attempts to deal with failures. After that, the measure is aborted.

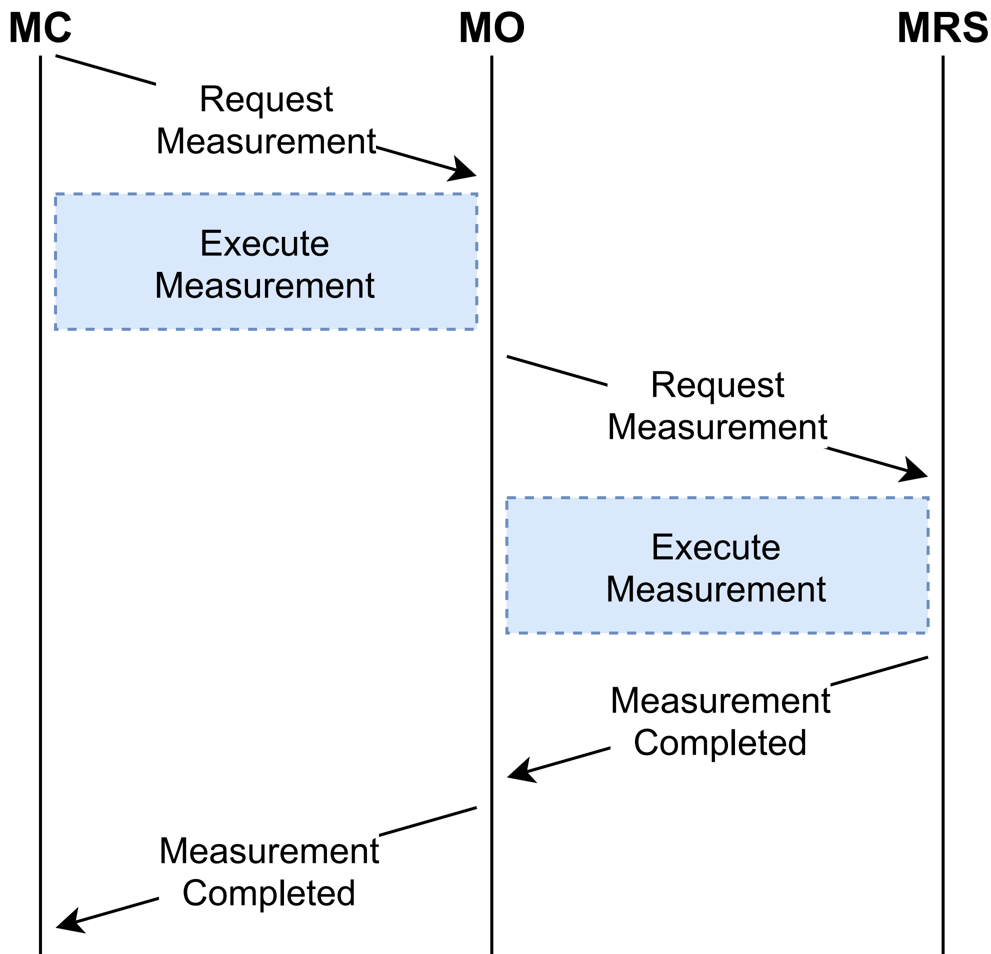


Figure 4.2: The interaction between the MECPerf Client (MC), the MECPerf Observer (MO), and the MECPerf Remote Server (MRS) during a generic MECPerf-active measure.

MECPerf-active measurement methods: TCP bandwidth and UDP bottleneck capacity

MECPerf computes both the TCP bandwidth of a data stream and the UDP bottleneck capacity. A TCP bandwidth measure computes the application-level throughput of a TCP connection. The behavior of a TCP bandwidth measure is depicted in Figure 4.3. At the beginning, the sender sends a request message containing the type of measures requested (e.g., a TCP bandwidth measure), the desired direction (i.e., uplink/downlink), and the number of bytes that need to be sent. Then, the sender starts to send its traffic, while the receiver starts to read the bytes from the TCP stream. The number *n_bytes* of bytes read, and the time t at which the operation completes are collected for each reading operation. Then, for the n-th reading operation, the TCP bandwidth can be computed at the receiving node as the number

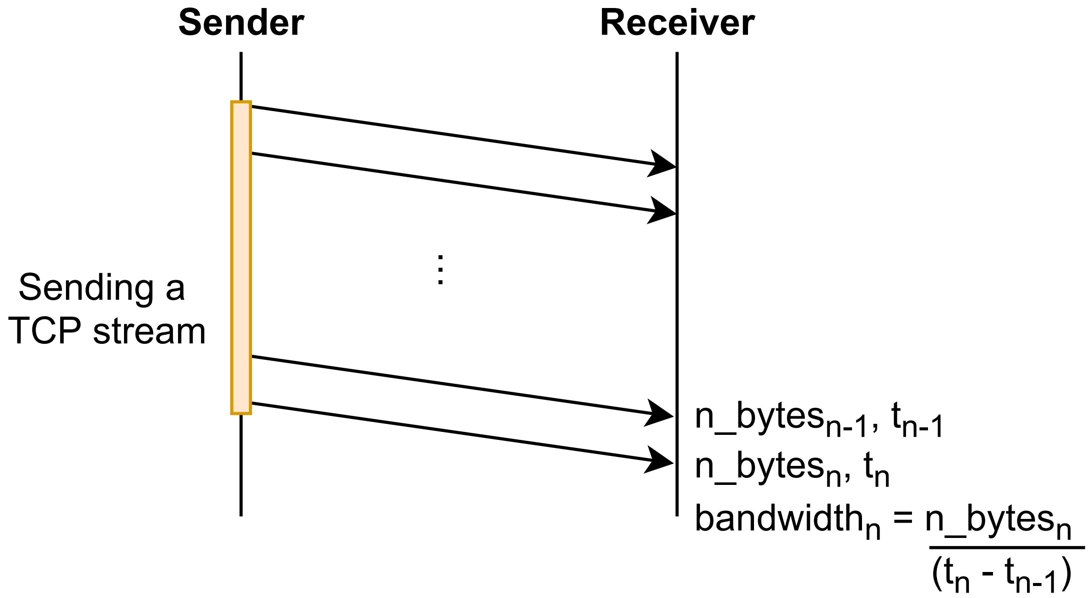
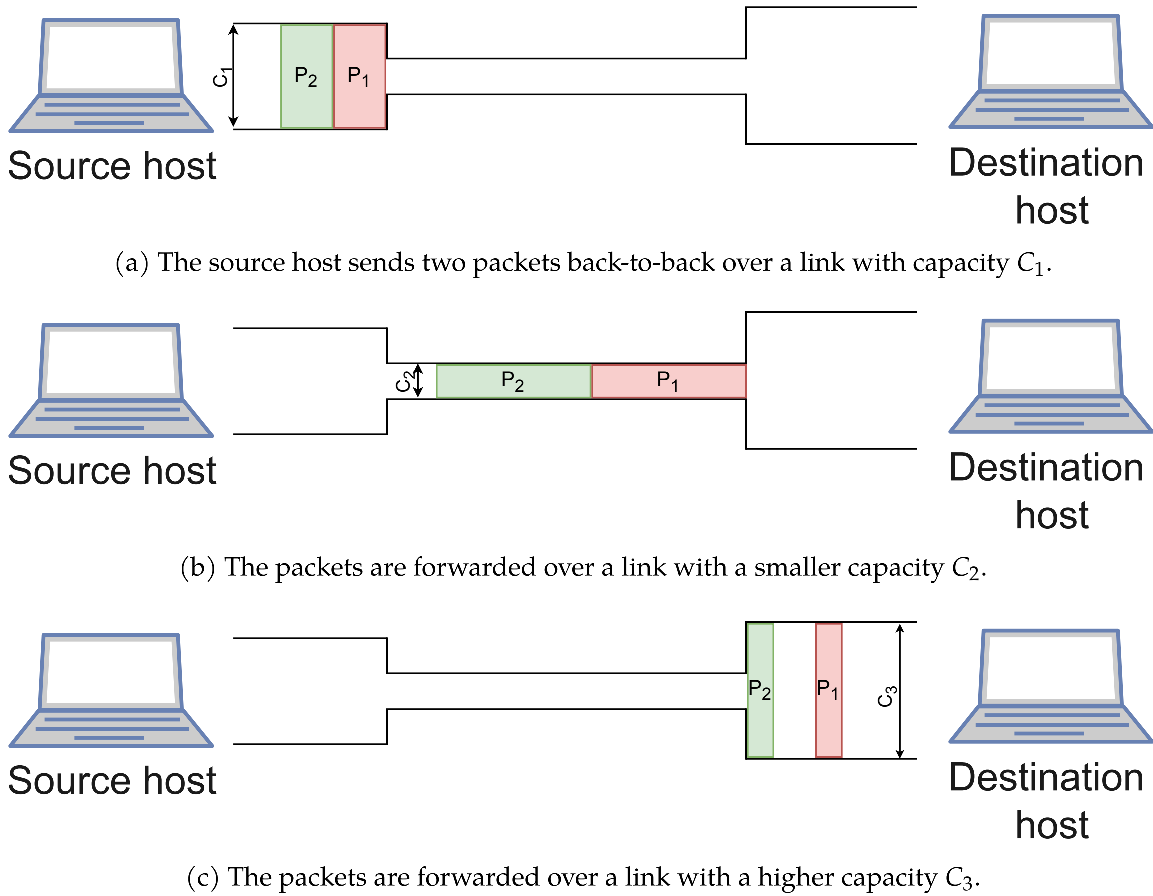


Figure 4.3: The sequence diagram of a TCP MECPerf-active bandwidth measure.

of bytes reads divided by the time elapsed from the previous read operation.

$$bandwidth_n = \frac{n_bytes_n}{t_n - t_{n-1}} \tag{4.1}$$

Instead, the bottleneck capacity is defined as the capacity of the narrowest link over a specific network path (Prasad et al., 2003). Thus, it can be used to estimate the maximum obtainable throughput over a path. MECPerf computes bottleneck capacity metrics using a well-known technique based on packet pairs which behave as shown in Figure 4.4. At the beginning, the source host sends two packets of identical size back-to-back over a link characterized by a capacity C_1. Each packet needs a time equal to $size(P_i)/C_1$ to be sent. At a certain point, a router receives P_1 and forwards the packet over a link characterized by a smaller capacity C_2. Since the capacity of the outbound link is smaller than the capacity of the inbound link, the time needed to receive P_2 is smaller than the time needed to transmit P_1. Hence, it is reasonable to assume that P_1 and P_2 are retransmitted back-to-back. Finally, the two packets encounter a third link with higher capacity C_3. In this circumstance, the time required to receive P_2 is greater than the time needed to transmit P_1. Therefore, the two packets cannot be forwarded back-to-back and the time between the two packets can be used to infer the capacity of the narrowest link. Note that the MC, the MO, and the MRS can behave as both sending and receiving nodes, depending on the considered segment and the desired direction. For example, during an uplink bottleneck capacity measure collected on the access-MEC network segment, the source and the destination hosts are the MC and the MO, respectively. Conversely,



(a) The source host sends two packets back-to-back over a link with capacity C_1.

(b) The packets are forwarded over a link with a smaller capacity C_2.

(c) The packets are forwarded over a link with a higher capacity C_3.

Figure 4.4: The basic functioning mechanism behind the packet pair technique.

for a downlink bottleneck capacity measure collected on the same network segment, the source host is the MO, while the MC is the destination host. The reader can find a more detailed explanation of this technique on (Prasad et al., 2003) and (Gregori et al., 2016).

Precisely, after the request message, two packets are sent back-to-back to the receiver. The receiver receives the two consecutive packets and stores the time t_0 and t_1 of their reception. Then, the UDP bottleneck capacity is computed as the size of a packet divided by the time elapsed between t_0 and t_1.

$$bottleneckcapacity = \frac{packet_size}{t_1 - t_0} \tag{4.2}$$

The procedure is repeated a certain number of times to ensure that the measurement is affected by the minimum amount of cross-traffic. The number of repetitions k and the size *packet_size* of each UDP packet are contained in the request message. The execution of a UDP bottleneck measure execution is illustrated in Figure 4.5.

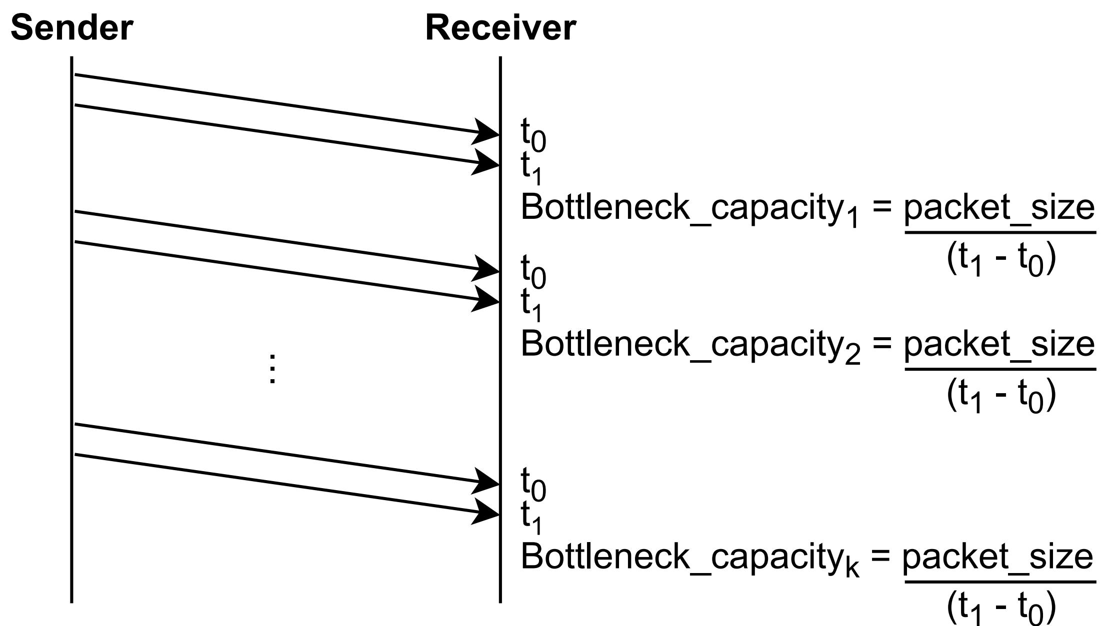


Figure 4.5: The sequence diagram of a UDP MECPerf-active bottleneck capacity measure.

MECPerf-active measurement methods: TCP and UDP latency

Latency measurement methods can be based on both TCP and UDP packets. A TCP-based latency measure can be considered as an application layer measurement since TCP is equipped with connection-oriented and reliability mechanisms. Conversely, a UDP-based latency measure can be regarded as the network/transport layer since UDP is best effort. However, differently from bandwidth measurement methods, their functioning mechanism is similar. After the request message, the sender sends a certain amount of bytes to the receiver, which sends back the data as soon as possible. For TCP latency metrics, this is performed by sending data over a previously established TCP stream. Instead, for UDP latency metrics, the data is sent as a UDP packet payload. Finally, after receiving the packet, the sender can compute the Round Trip Time (RTT). Then the procedure is iterated several times. The number of repetitions k and the size of each packet are contained in the request message. The execution of a generic MECPerf-active latency measure is summarized in Figure 4.6.

MECPerf-passive measurement methods

MECPerf-passive measurement methods involve only the MO and the MA, and they are used to compute transparently the TCP bandwidth, the UDP bandwidth, and the TCP latency metrics of third-party applications. Basically, the MO uses pcap files containing traffic traces to compute the metrics. The pcap file may be generated in

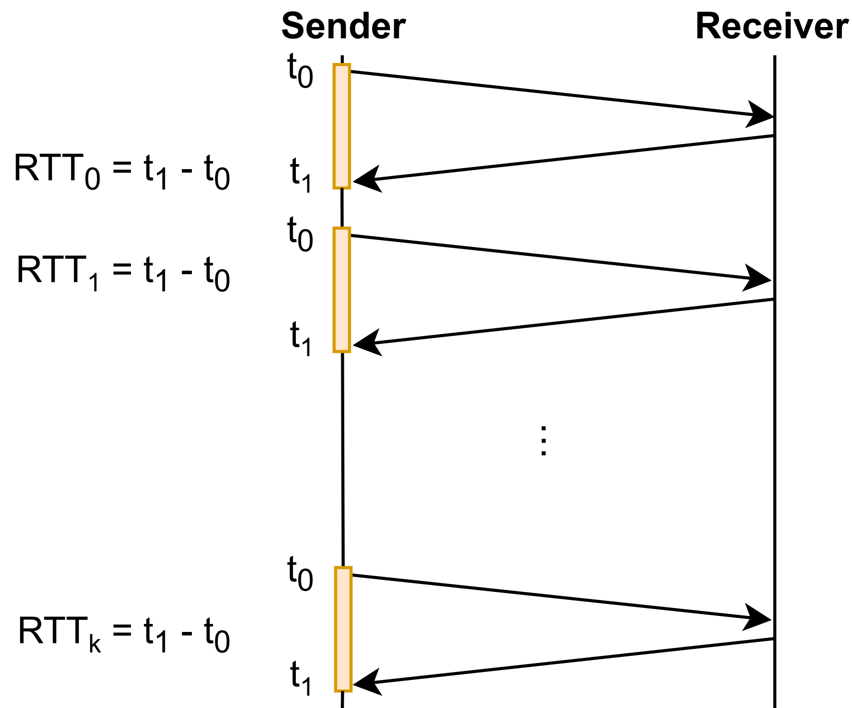


Figure 4.6: The sequence diagram of a MECPerf-active latency measure.

different ways. For example, they may be created by a packet sniffer able to intercept all the application traffic. When a pcap file is received, the MO searches for packets belonging to the target application. Then, for each selected packet, the MO collects the timestamp and the size of the payload, using them to compute the bandwidth of the target application at the application level. This methodology can be used to compute bandwidth metrics for both TCP- and UDP-based applications.

Instead, latency metrics are calculated as the difference between the timestamp of the ACK packet and the timestamp of the ACKed packet. Unfortunately, this means that latency metrics can be computed only for TCP-based applications since the UDP protocol lacks a mechanism for linking incoming and outgoing packets.

Finally, note that MECPerf-passive measures are computed from the MO, which has no knowledge about the application behavior. This means that low throughput metrics may be generated by legitimate idle periods. Consequently, applications with intermittent transmission periods are not suitable for MECPerf-passive measurement methods, while MECPerf-passive methods are more accurate in acquiring metrics for applications with constant data stream transmissions.

Self-active and self-passive measurement methods

Finally, self-measured methods let third-party applications store their metrics to the MA, the only component involved. This means that the application developers have

to decide the metrics that need to be collected, implement their measurement methods, and whether to store them into the MECPerf collection system. Measurements can be stored into the MA with HTTP POST requests, using the following JSON object as payload.


```
{
    "client_ip": "x.x.x.x",
    "client_port": xxxx,
    "server_ip": "y.y.y.y",
    "server_port": yyyy,
    "service": "<service_id>",
    "protocol": "<protocol>",
    "mode": "self-active"/"self-passive",
    "uplink": {
        "<timestamp>": value,
        ...
    },
    "downlink": {
        "<timestamp>": value,
        ...
    }
}
```


As can be seen, the JSON object contains the IP addresses and the ports used by the client and the server, an ID that identifies the application to which the metrics refer, the transport protocol used during the communication, and two arrays of metrics. The first one contains the metrics for the uplink segment, while the second contains the metrics for the downlink segment.

Note that self-passive measurement methods are more accurate than MECPerf-passive ones as the application can compute the metrics only in relevant periods, discarding the other ones. However, some developers may decide not to include any self-passive metrics. Hence, MECPerf-passive measurement methods are also needed.

Implementation details

The code of the MECPerf-active measurement methods is provided as a Java library, shared between the MC, MO, and the MRS. Hence the code is the same for all the components involved in measurements. Then, two different implementations of the MCs have been provided. The first one is a command-line Java application that can be used to perform automatic tests using dedicated hosts belonging to the network operator. The second one is an Android app that requires human interaction. How-

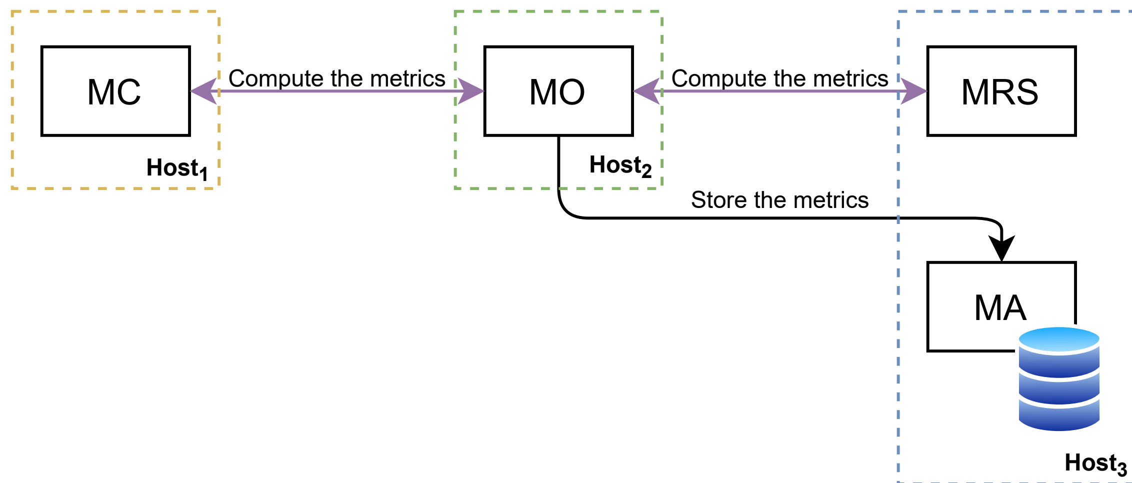


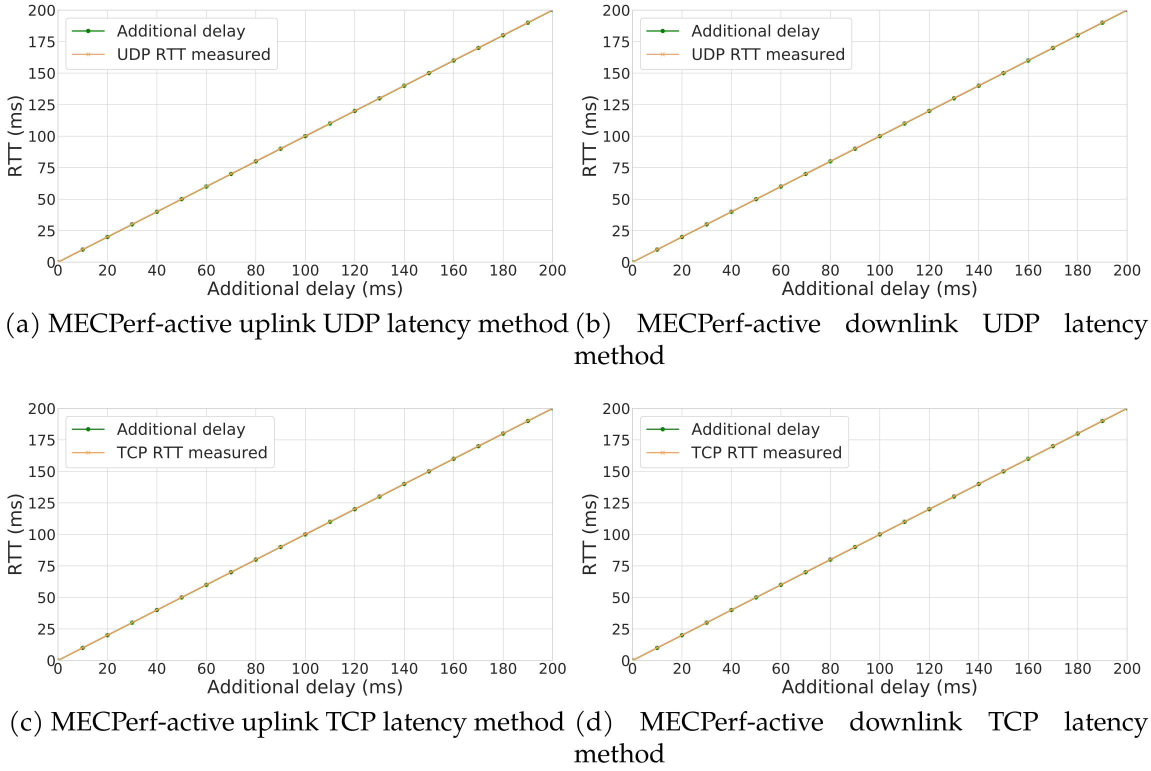
Figure 4.7: The setup used during the validation of the MECPerf collection system.

ever, the application can be easily adapted to be embedded into mobile applications of network providers running on TNs. Finally, the MA is implemented as a python Flask server(flask, 2021), while metrics are stored in a MySQL database. The source code of the MECPerf components can be found on GitHub (The source code of the MECPerf Collection system, 2021).

4.2 Validation

MECPerf has been validated in terms of both MECPerf-active and MECPerf-passive methods. Instead, self measurements methods have not been validated as they are under the application responsibility.

The validation of the MECPerf-based measurement methods has been conducted by installing the MECPerf components on three machines belonging to the network of the University of Pisa. Precisely the MC was placed on $Host_1$, the MO was hosted on $Host_2$, while the MRS and the MA were deployed on $Host_3$. The setup used during the validation phase is depicted in Figure 4.7. Artificial restrictions were applied on the network interfaces of $Host_2$ and $Host_3$ employing tc-netem (tc (Traffic Control), 2020; netem, 2020). These restrictions are always applied to the outgoing interface. This means that they have to be configured on the node that sends the measured traffic. Once the restriction was applied, we measured bandwidth and latency metrics on the links that connected the MO and the MRS. Consequently, to validate uplink and downlink bandwidth metrics computed on the link that connects the MO and the MRS, rate-limiting constraints were applied on the outgoing interface of $Host_2$ and $Host_3$, respectively. Instead, for both uplink and downlink latency metrics computed on the MEC-cloud link, the artificial delays were applied only on the interface of $Host_3$. In fact, configuring the delay on a single host is suf-



(a) MECPerf-active uplink UDP latency method (b) MECPerf-active downlink UDP latency method

(c) MECPerf-active uplink TCP latency method (d) MECPerf-active downlink TCP latency method

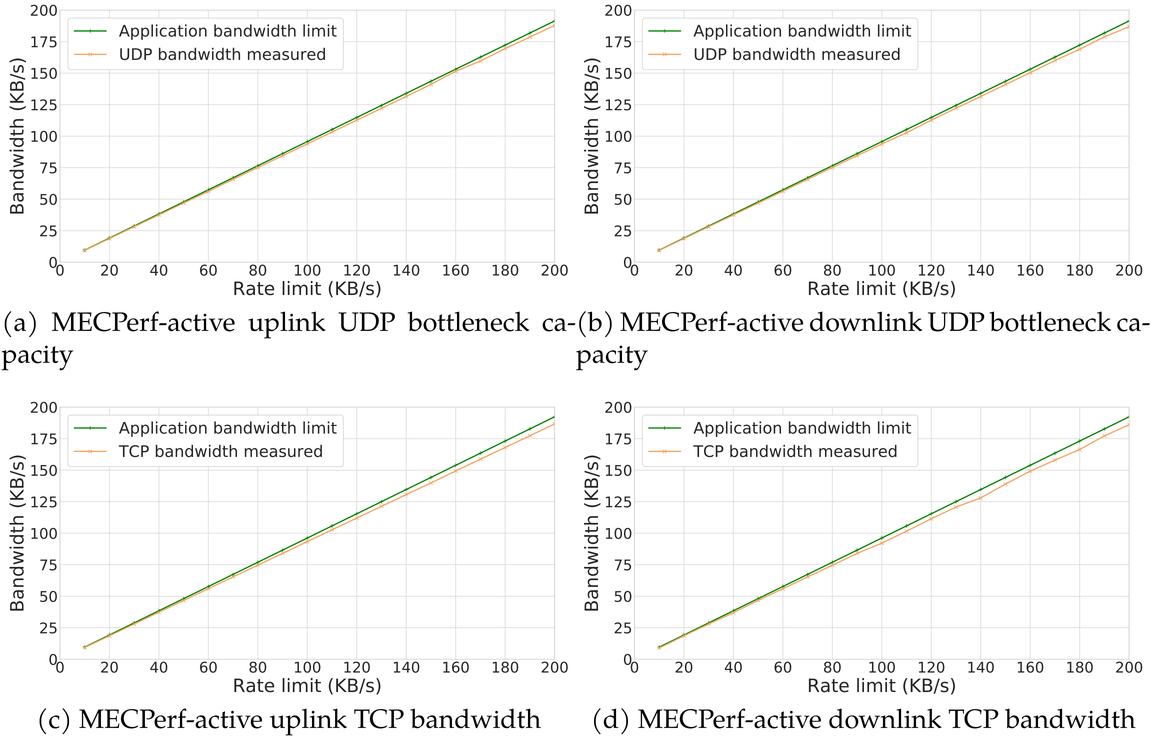
Figure 4.8: The results of the validation for MECPerf-active latency measurement methods. RTT values have been averaged among 10 repetitions.

ficient to validate the two directions since both hosts must send a packet during a latency measure. Finally, we compared the collected result and the expected one.

Validation experiments for MECPerf-active latency, TCP bandwidth, and UDP bottleneck capacity have been repeated 10 times. The MECPerf-active latency values, measured on the link between the MO and the MRS for increasing delay restrictions (from 0 ms up to 200 ms), are shown in Figure 4.8. As can be seen, tangible differences between the delays applied and the collected latency metrics cannot be observed. Hence, the validation of latency methods succeeded. Instead, the observed MECPerf-active bandwidth metrics and the bandwidth restriction applied cannot be directly compared. In fact, MECPerf computes the bandwidth at the application level, while tc restrictions are applied at the data link layer. Let r_{pkt} be the rate limit applied at the data link layer, L_{pkt} be the total length of data link packets, and L_{app} the amount of application-layer data in each packet. Then we can compute the expected application-level bandwidth as

$$r_{app} = \frac{L_{app}}{L_{pkt}} \cdot r_{pkt} \tag{4.3}$$

Let H_{IP} and H_{UDP} be the lengths of the IPv4 and the UDP headers. We considered



(a) MECPerf-active uplink UDP bottleneck capacity

(b) MECPerf-active downlink UDP bottleneck capacity

(c) MECPerf-active uplink TCP bandwidth

(d) MECPerf-active downlink TCP bandwidth

Figure 4.9: The results of the validation for MECPerf-active TCP bandwidth and UDP bottleneck capacity measurement methods. TCP bandwidth and UDP bottleneck capacity values have been averaged among 10 repetitions.

Parameter	Size (bytes)
$L_{app_{UDP}}$	1024
MTU	1500
H_{IP}	20
H_{TCP}	20
H_{UDP}	8
H_{ETH}	14
T_{ETH}	4

Table 4.1: The values of the operational parameters used during the validation of both MECPerf-active TCP bandwidth and MECPerf-active UDP bottleneck capacity methods.

L_{app} values that are smaller than $MTU - H_{IP} - H_{UDP}$. Then, it is possible to assume that the Operating System (OS) used a single packet to send the UDP data. Consequently, the packet length L_{pkt} of each UDP datagram can be computed as

$$L_{pkt} = L_{app} + H_{UDP} + H_{IP} + H_{ETH} + T_{ETH} \quad (4.4)$$

where H_{ETH} and T_{ETH} are the lengths of the Ethernet header and trailer, respec-

tively.

When TCP measures are considered, a stream of bytes is sent in a short amount of time. This means that TCP packets can be assumed to be filled at their maximum capacity. Then, the amount of application-level data contained in each packet can be computed as the Maximum Transmission Unit (MTU) minus the size of the headers. In other words, L_{app} and L_{pkt} can be computed as

$$L_{app} = MTU - H_{IP} - H_{TCP} \tag{4.5}$$

$$L_{pkt} = MTU + H_{ETH} + T_{ETH} \tag{4.6}$$

where H_{TCP} is the length of the TCP header.

The value of each parameter is reported in Table 4.1. Then, the application level rate limit can be computed as

$$r_{app_UDP} = 0.957 \cdot r_{pkt} \tag{4.7}$$

$$r_{app_TCP} = 0.962 \cdot r_{pkt} \tag{4.8}$$

for UDP and TCP traffic, respectively.

Figure 4.9 compares the MECPerf-active bandwidth metrics collected by MECPerf during the validation and the correspondent expected r_{app}, considering r_{pkt} values that go from 10 KB/s to 200 KB/s. As can we see, some differences can be noted only for high r_{app} values. Moreover, more differences can be appreciated comparing the TCP and the UDP case. This can be explained as the effect of the TCP slow-start mechanism, which slightly reduces the average throughput. However, differences between the measured bandwidth and the applied r_{app} limits are below 3%.

Finally, the MECPerf-passive measurement methods have been validated a single time using an iperf TCP data transfer as target application and from 10 Mbps to 50 Mbps of rate limits. Note that the amount of data transferred using iperf is much greater than the amount of data used to validate MECPerf-active measurement methods. Hence, since both the iperf and the MECPerf estimations are based on a non-negligible amount of data, we decided not to repeat MECPerf-passive experiments. Instead, latency measurement methods were validated using additional 10, 50, and 100 milliseconds of delay. The results of both MECPerf-passive bandwidth and latency validation tests are summarized in Table 4.2. As expected, bandwidth results are coherent with both the applied restriction and bandwidth measured computed by iperf. Additionally, the latency measurements obtained matched the additional delay imposed plus the latency computed when tc-netem is disabled. Hence, also the validation of MECPerf-passive measurement methods succeeded.

tc bandwidth restriction (Mbps)	MECPerf result (Mbps)	iperf result (Mbps)
10.0	9.60	9.60
20.0	19.1	19.1
30.0	28.7	28.7
40.0	38.3	38.2
50.0	47.4	47.4

(a) MECPerf-passive TCP bandwidth results

tc artificial latency (ms)	MECPerf result (ms)
0	0.8
10	10.8
50	50.8
100	100.8

(b) MECPerf-passive TCP latency results

Table 4.2: The results obtained during the validation for MECPerf-passive measurement methods.

Chapter 5

Data collection and experimental results

The MECPerf collection tool explained in the previous Chapter has been used to collect MECPerf-active, MECPerf-passive, and self-passive metrics in an edge environment. To evaluate an edge environment was used the NITOS testbed, while some machines deployed at the University of Pisa have been used as cloud servers. NITOS is a testbed facility hosted at the University of Thessaly (Greece) participating in the Fed4Fire+ European federation of Next- Generation Internet testbeds (Fed4Fire+, 2017). NITOS makes available nodes equipped with Wi-Fi and LTE interfaces. This allows comparing edge and cloud performances, considering TNs connected with different access technologies.

In the following will be explained the setup employed to conduct the experiments. Then, edge and cloud performance will be compared considering different access technologies and network conditions. The dataset containing the metrics collected during the experiments is available on Zenodo(MECPerf experimentation results, 2020).

5.1 Network setup

This section will explain the setup used to collect MECPerf-active, MECPerf-passive, and self-passive metrics. Table 5.1 summarizes the characteristics of both NITOS nodes and cloud servers belonging to the University of Pisa.

MECPerf-active measurement setup

The first set of experiments pointed at collecting MECPerf-active metrics for the access-MEC, MEC-cloud, and access-cloud network segments. To this purpose, two networks were built inside the testbed. The first one was a wireless access network

NITOS Indoor RF Isolated Testbed details	
Nodes	50 Icarus nodes
LTE connectivity	8 nodes equipped with LTE dongles
LTE dongles	Huawei E392, Huawei E3272, and Huawei E3372. LTE experiments were performed using the Huawei E3272 dongle characterized by a maximum speed of 150 Mbps in download and 50 Mbps in upload.
Topology	Grid topology with adjacent nodes separated by 1.2 meters

Icarus nodes details	
OS	Ubuntu 14.04.1 LTS for LTE nodes and Ubuntu 12.04.1 LTS for the remaining nodes
CPU	Intel® Core™ i7-2600 Processor, 8M Cache, at 3.40 GHz
RAM	8GiB DDR3
Wireless Interfaces	Atheros 802.11a/b/g and Atheros 802.11a/b/g/n (MIMO)

University of Pisa cloud infrastructure details	
Guest OS	Ubuntu 18.04.3 LTS
Host CPU	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
Number of allocated cores	2
Guest RAM	4GiB

Table 5.1: The technical characteristics of the devices used during MECPerf-active, MECPerf-passive, and self-passive experimental data collection.

that hosted an MC and a cross-traffic generator. Instead, the second one was a wired MEC network, which hosted a MO and a cross-traffic receiver. Depending on the access technology employed, the hosts within the access network received network connectivity from a Wi-Fi Access Point (AP) or an LTE BS. Precisely, for experiments based on Wi-Fi connections, a NITOS node was configured to act as a Wi-Fi AP. Instead, during LTE-based experiments, a dedicated LTE BS provided by the testbed gave connectivity to a set of NITOS nodes equipped with LTE dongles. Finally, a second MO, an MRS, and an MA were hosted at the University of Pisa. Note that the edge and the cloud networks were deployed in two different countries. This setup is coherent with a MEC scenario composed of an edge network deployed near the user and a far cloud network that involves transit on the public Internet. Then,

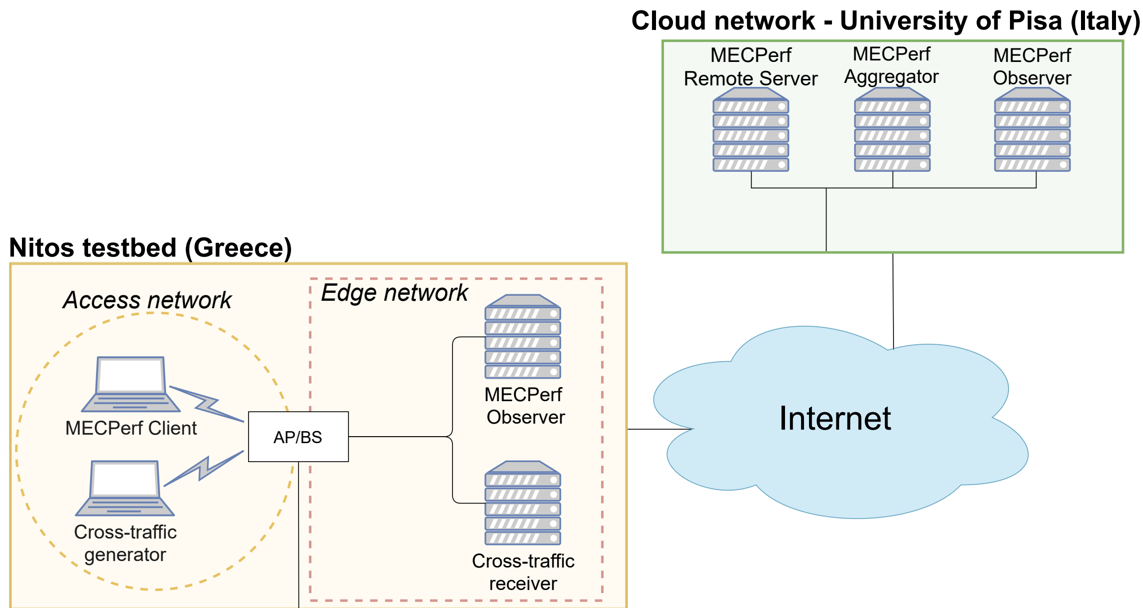


Figure 5.1: The setup used during the collection of MECPerf-active metrics.

it should also be noted that the setup used during the experimental phase slightly differs from the generic MECPerf architecture depicted in Figure 4.1 as two MOs were employed. The first MO, hosted inside the edge network, was used to collect the metrics on the access-MEC (i.e., between the MC and the MO) and the MEC-cloud segments (i.e., between the MO and the MRS). Instead, the MO hosted at the University of Pisa collected the access-cloud metrics (i.e., between the MC and the MO). Figure 5.1 depicts the setup used to collect MECPerf-active metrics for the considered segments.

```
configure the Wi-Fi AP/LTE BS;
connect the MC and the cross-traffic generator to the AP/BS;
for c ∈ [0, 10, 20, 30, 40, 50] do
    configure the cross-traffic generator to inject c Mbps of cross-traffic;
    for i ∈ [1, ..., 10] do
        collect MECPerf-active metrics using the MO deployed in the edge
          network;
        collect MECPerf-active metrics using the MO deployed in the cloud
          network;
    end
end
```

Algorithm 1: The setup used to collect MECPerf-active metrics using the NITOS testbed.

Each experiment was organized as shown in the pseudo-code of Algorithm 1. At the beginning of a Wi-Fi experiment, a NITOS node was configured to act as an

Measurement method	Measurement parameters	
	Payload size (bytes)	Number of packets
TCP bandwidth	1420	1 TCP stream of 1024 packets
UDP bottleneck capacity	1420	25 packet pairs
TCP latency	1	25 RTT measures
UDP latency	1	25 RTT measures

Table 5.2: The operational parameters used to collect MECPerf-active metrics.

AP, providing connectivity to the MC and the cross-traffic generator. Similarly, at the beginning of each LTE-based experiment, the dedicated LTE BS provided by the testbed was connected with NITOS nodes equipped with LTE dongles. Then, the cross-traffic generator used iPerf3 to inject cross-traffic into the access network. Once the cross-traffic generator was configured, the MC started to use the MO hosted in the edge network to collect uplink and downlink MECPerf-active metrics on both the access-MEC (between the MC and the MO) and the MEC-cloud segment (between the MO and the MRS). Then, the MC repeated the same measures using the MO hosted at the University of Pisa as target. As a result, access-cloud metrics were collected, while metrics computed between the cloud MO and the MRS were discarded as they measured the performance of the wired University of Pisa's network. For each amount of cross-traffic, the MECPerf-active measurement methods were executed 10 times. Then, the procedure was repeated considering an increasing quantity of cross-traffic to evaluate the network metrics under different workload levels. Note that the cross-traffic is consumed near the base station. This means that only the wireless segment was affected by cross-traffic.

TCP bandwidth metrics were collected using a stream composed of 1024 packets with 1420 bytes of payload each. UDP capacity metrics were collected using 25 packet pairs, using packets with 1420 bytes of payload. Instead, TCP and UDP latency metrics were computed using 25 RTT and packets with a payload of 1 byte. The details of the MECPerf-active experiments are summarized in Table 5.2.

MECPerf-passive measurement setup: DASH application

MECPerf-passive experiments were conducted considering a scenario where several Dynamic Adaptive Streaming over HTTP protocol (DASH)(Sodagar, 2011) clients and a DASH server are running on TNs and third-party application servers, respectively. Basically, a DASH application behaves as follows. The DASH server provides a video streaming service to multiple DASH clients, using videos encoded at different bitrates. Instead, the goal of each DASH client is to download video chunks at the maximum bitrate that does not generate re-buffering events. This task is accom-

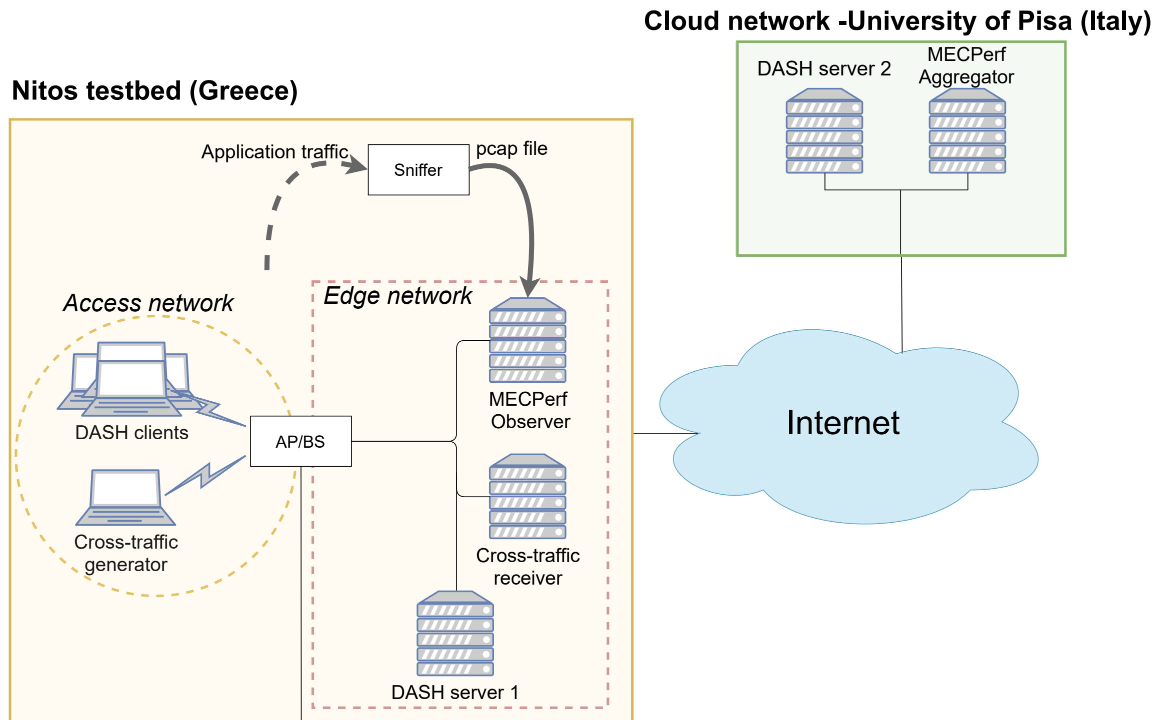


Figure 5.2: The setup used during the collection of MECPerf- and self-passive metrics for a DASH application.

plished using an Adaptive Bitrate (ABR) algorithm that can dynamically adapt the bitrate of the requested video chunks to the status of the network. We had chosen a DASH application for two reasons. Firstly, the video streaming application field is one of the six relevant use cases identified by the European Telecommunications Standards Institute (ETSI) for MEC (Hu et al., 2015). Secondly, a video streaming application is well suited to be evaluated through MECPerf-passive measurements, as it can generate a constant stream of data.

The setup used to collect MECPerf-passive metrics is shown in Figure 5.2. As expected, the only MECPerf components involved in MECPerf-passive metrics collection were the MO and the MA. Within the testbed, a wireless access network and a wired MEC network were built similarly to the MECPerf-active experiments. The access network hosted the cross-traffic generator and multiple TNs running the DASH client. In addition, the Wi-Fi AP and the LTE BS have also been configured as previously described. Instead, the MEC network hosted an MO, a cross-traffic receiver, and a DASH server, while the cloud network contained a second application server and the MA. Note that, also in this case, two application servers are used. The first application server was used to compute the performance of the access-MEC segment, while the second one was used to collect access-cloud metrics. Finally, to generate the pcap file needed to compute MECPerf-passive metrics, a packet snif-

fer run on a network point where all the traffic between DASH clients and the application server can be captured. For Wi-Fi experiments, the sniffer was a process running into the AP. Instead, for LTE experiments, the sniffer process was executed directly on the application server since the LTE BS cannot be configured to run such software.

```
configure the Wi-Fi AP/LTE BS;
connect the DASH client and the cross-traffic generator to the AP/BS;
for each video v do
    for c ∈ [0, 10, 20, 30, 40, 50] do
        configure the cross-traffic generator to inject c Mbps of cross-traffic;
        start the packet sniffer;
        for s ∈ [DASH sever 1, DASH server 2] do
            for i in [0, NUM_Clients] do
                the i-th DASH client starts to download the video v from the
                  s-th DASH server;
            end
            waits for all the DASH clients to complete their download ;
        end
    end
end
```

Algorithm 2: The setup used during the collection of MECPerf- and self-passive metrics using the NITOS testbed.

In detail, each experiment was performed as illustrated in the pseudo-code of Algorithm 2. At the beginning of each experiment, the Wi-Fi (LTE) AP (BS) was configured to give connectivity to the hosts inside the access network. At this point, for each video hosted on the application servers and for each amount of cross-traffic considered, the cross-traffic generator and the packet sniffer were started. Then, the clients began to download the selected video from the edge server producing access-MEC metrics. Once each client completed the first download, the clients started downloading the video from the cloud server, collecting access-cloud metrics. Then, the cross-traffic generator is configured to inject a new level of cross-traffic into the access network.

Passive experiments were repeated using up to 10 clients for Wi-Fi experiments and 2 clients for LTE-based experiments. Moreover, from 0 Mbps (i.e., no cross-traffic is injected into the access network) to 50 Mbps of cross-traffic were considered. The two DASH servers provided four videos of 1, 5, 12, and 25 minutes each. Each video is encoded using 3 different bit rates (i.e., 200 Kbps, 500 Kbps, and 700 Kbps). The operational parameters of MECPerf-passive experiments have been reported in Table 5.3.

Operational parameters parameters	
File length	1, 5, 12, and 25 minutes
Encoding bitrate	200, 500, and 700 Kbps
Cross-traffic	0, 10, 20, 30, 40, and 50 MBps
Number of clients	up to 2 for LTE-based experiments and up to 10 for Wi-Fi-based experiments

Table 5.3: The operational parameters used to collect MECPerf- and self-passive metrics.

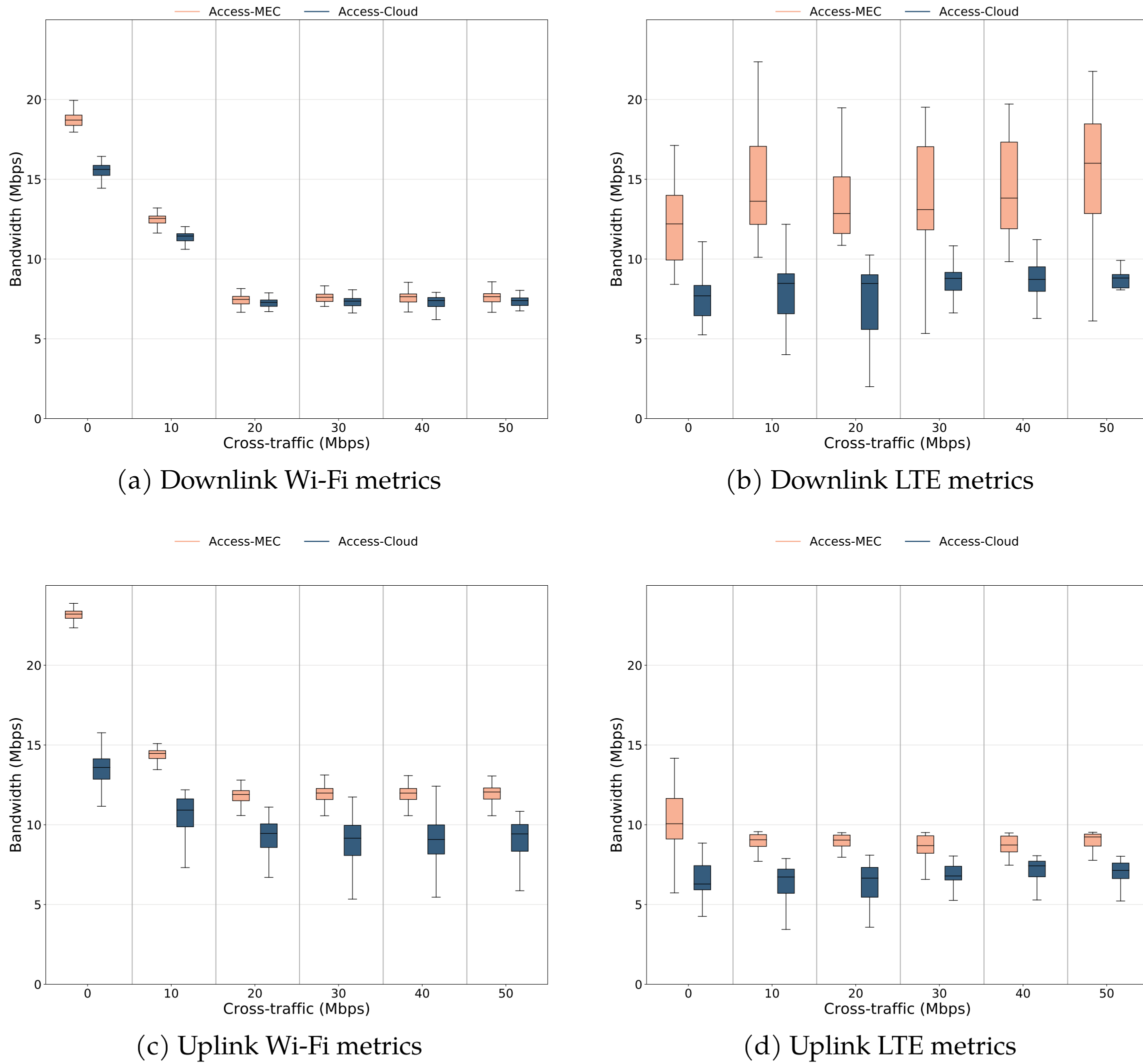
Self-passive measurement setup: DASH application

DASH-based network metrics based on self-passive measurement methods had been collected with the MECPerf-passive ones as the two methods do not interfere with each other. In fact, when a video is downloaded, self-passive metrics are computed directly by the client node. In contrast, MECPerf-passive methods are calculated by the MO using pcap files generated on the AP (or on the application server for LTE-based experiments). The only shared node is the MA, which only stores the metrics and is not involved in any measure. This means that MECPerf- and self-passive measured metrics could be gathered together within the same download. This diminished the number of experiments that need to be performed, and consequently, the time required to collect MECPerf- and self-passive metrics. Moreover, MECPerf- and self-passive metrics can be compared as they refer to the same download. Hence, the setup of Figure 5.2, the Algorithm 2, and the parameters of Table 5.3 are still valid.

5.2 MECPerf-active measurement results

TCP bandwidth results

TCP MECPerf-active bandwidth metrics are shown in Figure 5.3. The results are expressed by means of boxplots where the median values are indicated by a horizontal line inside the box, the interval Q1-Q3 is delimited by the borders of the box, and IQR 1,5 interval is delimited by the whiskers. First of all, it can be seen that the bandwidth metrics collected on the access-MEC segment are always higher than those gathered on the access-cloud segment. This difference is particularly noticeable for low cross-traffic values, while it is barely perceptible starting from 20 Mbps of cross-traffic. This result was expected, and it can be mainly ascribed to the TCP protocol. In fact, it is known that the throughput of a TCP connection is inversely proportional to the RTT between the two endpoints. Hence, it is reasonable to as-



(a) Downlink Wi-Fi metrics

(b) Downlink LTE metrics

(c) Uplink Wi-Fi metrics

(d) Uplink LTE metrics

Figure 5.3: The boxplots of MECPerf-active TCP bandwidth metrics computed during Wi-Fi and LTE experiments.

sume that the segment with the lower latency is the one with the higher bandwidth values.

Secondly, we can see that the metrics collected during Wi-Fi and LTE experiments show different behavior. The highest bandwidth values are collected for Wi-Fi experiments when no cross-traffic is injected into the access network. Then, when a higher level of cross-traffic is considered, the bandwidth metrics decrease. These results can be explained by the fact that the MC and the cross-traffic generator shared the same wireless link. Hence, higher cross-traffic levels could lead to the saturation of the wireless connection. Instead, the LTE results show different behavior. As can be seen, access-MEC uplink metrics (Figure 5.3d) are similar to the metrics collected during the Wi-Fi experiments. When no cross-traffic is injected into the ac-

cess network, the access-MEC segment obtains higher metrics than those collected for the access-cloud segment. When the cross-traffic was considered, the access-MEC metrics stabilized on slightly lower values. Instead, metrics gathered on the access-cloud segments appear to be wholly unaffected by the cross-traffic. Moreover, if we consider the downlink experiments of Figure 5.3b, it is possible to see that the access-MEC bandwidth metrics increase when the level of cross-traffic increases. Several reasons may contribute to this counter-intuitive result. Firstly, LTE has a higher nominal capacity than Wi-Fi. This means that 50 Mbps of cross-traffic may not be sufficient to saturate the link. Secondly, LTE scheduling mechanisms may isolate the MC traffic and the cross-traffic. Moreover, other fluctuations may be ascribed to other optimization mechanisms.

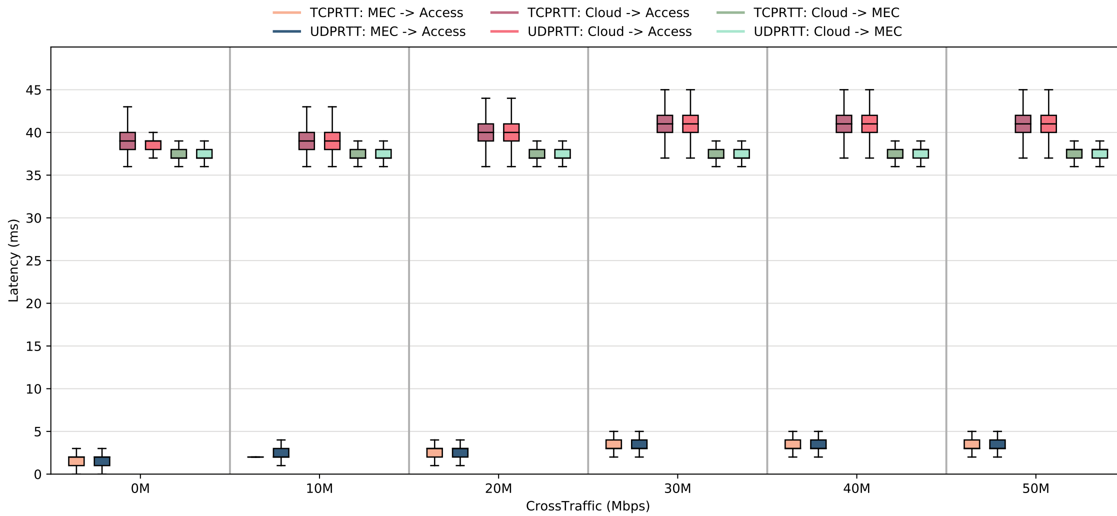
Latency results

The results of MECPerf-active latency experiments are shown in Figures 5.4 and 5.5 for Wi-Fi and LTE experiments, respectively. First, for both Wi-Fi and LTE results, it can be noted that TCP and UDP metrics show similar behavior. During Wi-Fi experiments, the latency metrics computed on the access-MEC segment are equal to a few milliseconds and, as expected, seem to be affected by cross-traffic. Additionally, it is possible to note that high cross-traffic values seem not to affect the MEC-cloud segment. However, this last result is reasonable as the cross-traffic saturates only the wireless link. Finally, it can be noted that latency values computed on the access-cloud segments are just equal to the sum of access-MEC and MEC-cloud metrics.

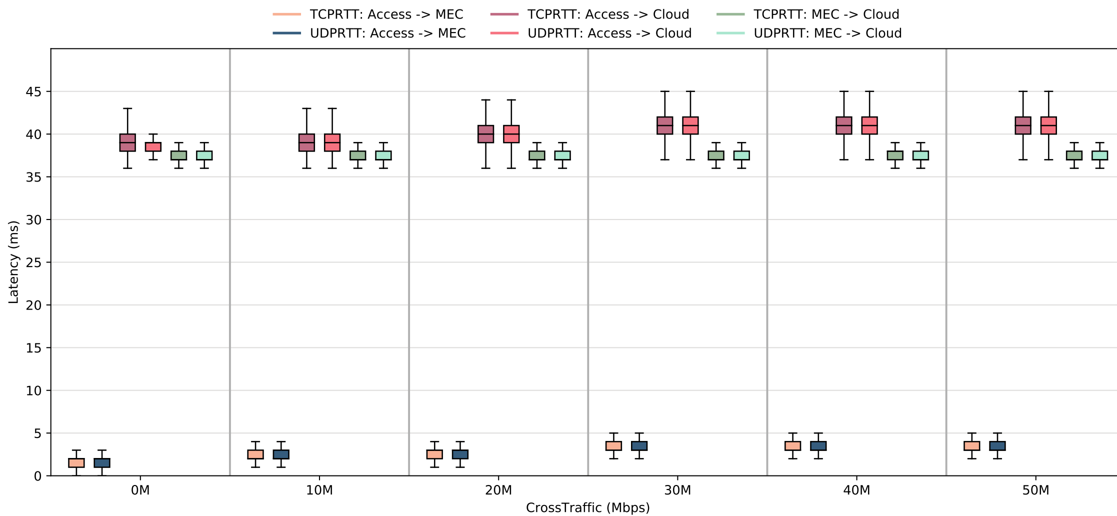
When the LTE connection is involved, the access-MEC and the access-cloud segments obtain higher latency than those obtained during Wi-Fi experiments. Instead, the latency values collected on the MEC-cloud segment are similar to those collected during Wi-Fi experiments. This is an expected result as the MEC-cloud segment did not include the wireless link. Then, it can be noted that LTE metrics are higher when there is no cross-traffic injection into the access network. Instead, when there is some cross-traffic into the access network, latency metrics appear to be not affected by the amount of cross-traffic. This result is coherent with the metrics collected during TCP measurements. In fact, higher latency metrics are obtained when there is no cross-traffic into the access network. Consequently, since the TCP throughput is negatively affected by the RTT between the two endpoints, lower bandwidth values are obtained. Furthermore, these results can confirm that LTE traffic optimization mechanisms can introduce additional delays to packets in low traffic conditions.

UDP bottleneck capacity results

Finally, the results of UDP bottleneck capacity measures will be discussed in the following. As previously stated in Chapter 4, UDP bottleneck capacity measures aimed



(a) Downlink metrics

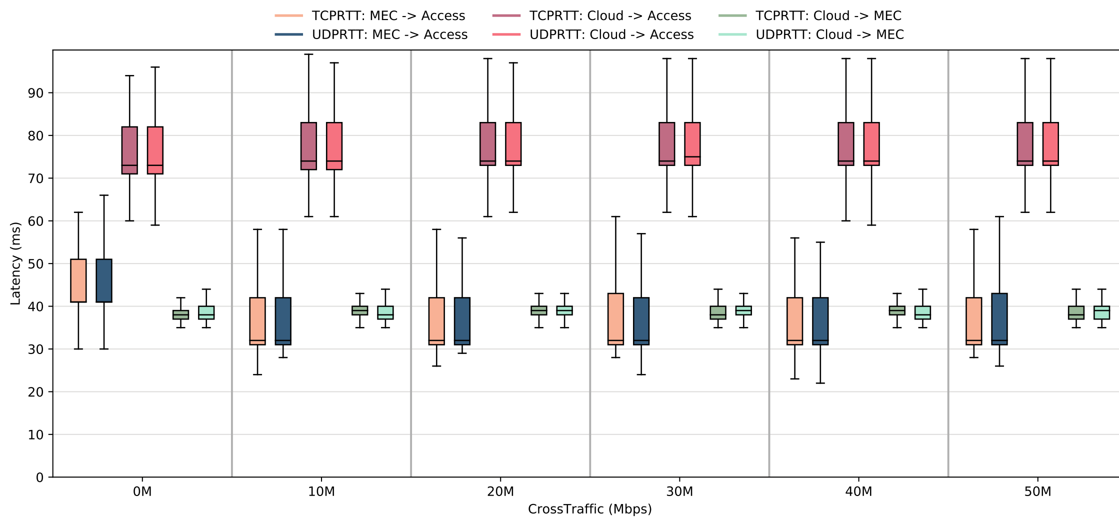


(b) Uplink metrics

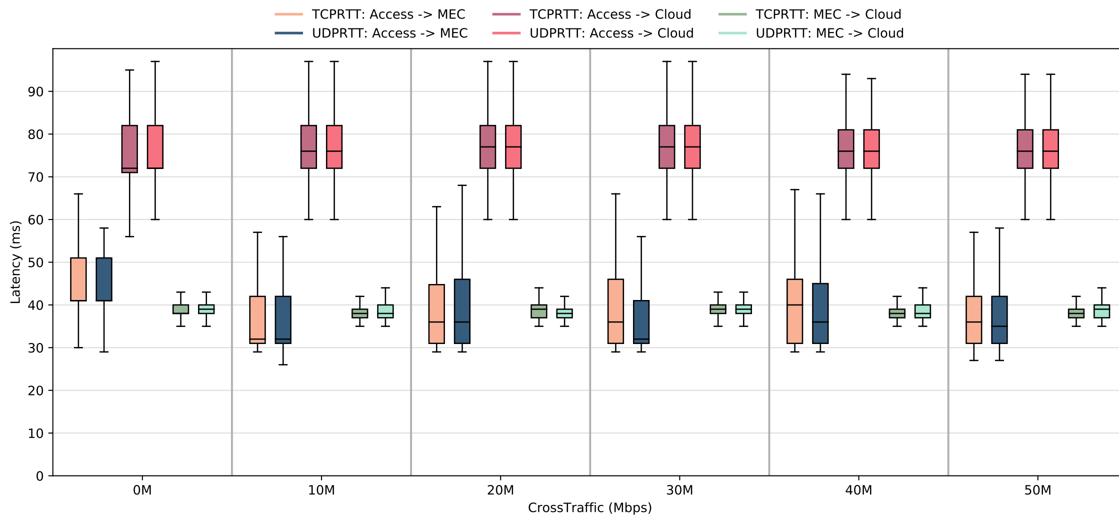
Figure 5.4: The boxplots of MECPerf-active UDP and TCP latency metrics computed during Wi-Fi experiments.

to measure the capacity of the narrowest link of the network. To further investigate UDP capacity metrics, we adopted a statistical approach known in the literature (Dovrolis et al., 2004). Basically, the metrics collected using the packet pair technique follow a multimodal distribution, and in a low workload scenario, the highest peak represents the best estimation of the bottleneck capacity.

Then, let us analyze the results of Wi-Fi experiments, considering a scenario where there is no cross-traffic within the access network. The distribution of UDP bottleneck capacity downlink metrics is illustrated in Figure 5.6. As can be seen, the distribution of access-MEC and access-cloud metrics show their highest peaks



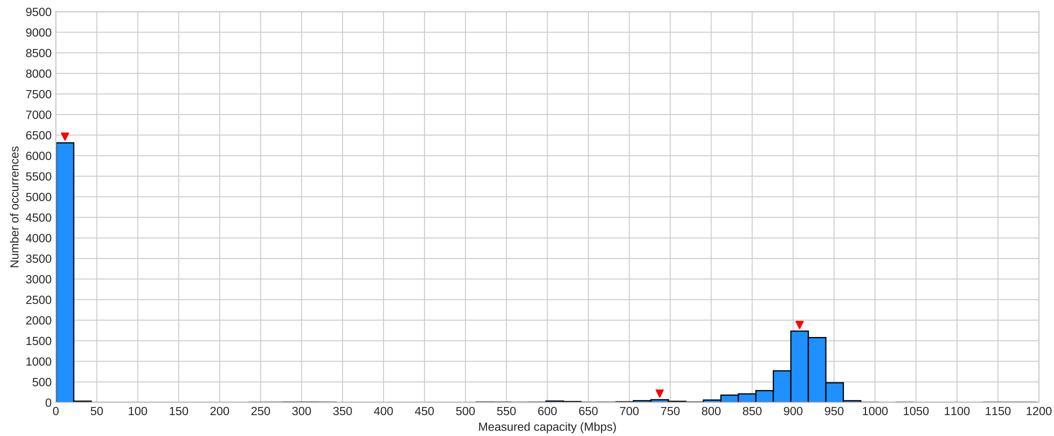
(a) Downlink metrics



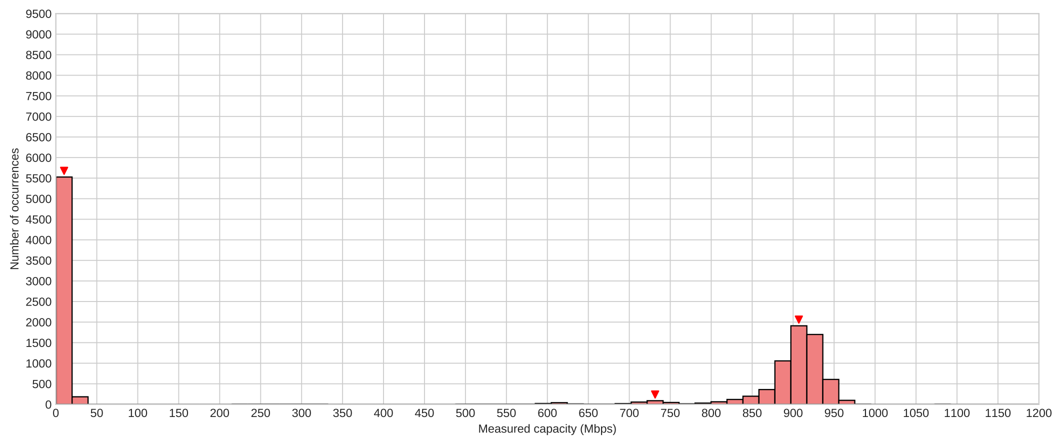
(b) Uplink metrics

Figure 5.5: The boxplots of MECPerf-active UDP and TCP latency metrics computed during LTE experiments.

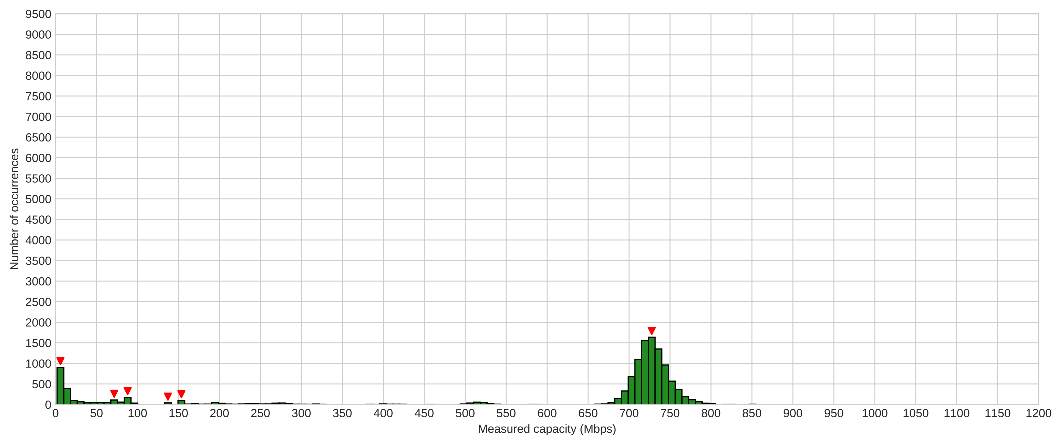
between 8 and 20 Mbps. This result is consistent with the TCP MECPerf-active bandwidth metrics of Figure 5.3a. Then, it is possible to note that the distribution has additional peaks around approximately 800 and 1000 Mbps. This result is not coherent with the capacity of a Wi-Fi link. However, the presence of multiple rightmost peaks can be ascribed to buffering events that reduce the separation of the pair of packets. Precisely, uplink packets may be buffered at the access point and subsequently sent over a wired path characterized by higher capacity. Instead, for the downlink traffic, the rightmost peaks can be generated by interrupt coalescence events on the receiver interface. In other words, the incoming packets may be buffered at the receiving in-



(a) Access-MEC metrics



(b) Access-cloud metrics



(c) MEC-cloud metrics

Figure 5.6: The results collected during UDP bottleneck capacity MECPerf-active downlink Wi-Fi experiments considering a scenario where no cross-traffic is injected into the access network.

terface in order to reduce the number of interrupt events generated on the receiver interface.

Finally, it can be noted that the MEC-cloud segment reached its highest peak at about 750 Mbps. This result is much higher than that achievable through a TCP connection. However, it must be remarked that TCP bandwidth and UDP bottleneck capacity measure two different properties of the path. In fact, TCP bandwidth metrics measure the available bandwidth and they are subjected to slow-start, congestion, and flow control mechanisms. Instead, the UDP bottleneck capacity metrics measure the capacity of the narrowest link of a path. This means that it is reasonable to assume that the TCP throughput is smaller than the bottleneck capacity of the path. Moreover, both the NITOS hosts and the machines belonging to the University of Pisa are furnished with 1 Gbps network cards. Hence, a bottleneck capacity of about 750 Mbps is coherent with a cabled high-speed network scenario. Finally, it should be noted that there is an additional peak at about 10 Mbps. This behavior is generally caused by cross-traffic packets interleaving the ones belonging to the measure. Note that not considering the access network does not mean that there is no cross-traffic as the MEC-cloud segment is affected by the interference caused by legitimate Internet packets.

5.3 DASH measurement results

The results of self-passive and MECPerf-passive experiments will be illustrated below, considering the impact on network performance of both the cross-traffic level and the number of users.

Passive results will be evaluated in terms of both TCP latency and bandwidth metrics. For a DASH application, the downlink traffic consists of video fragments going from the server to the client. On the contrary, uplink traffic consists of only a few sporadic DASH requests going in the opposite direction. Therefore, the uplink traffic is not suitable for computing MECPerf-passive bandwidth metrics due to the too limited amount of traffic generated. For this reason, TCP bandwidth metrics will be illustrated only for the downlink direction, while latency metrics will be analyzed in both directions. Instead, self-passive results will consist only of bandwidth metrics as the DASH client adopted did not compute any latency metric.

Evaluating the impact of different levels of cross-traffic

TCP bandwidth: MECPerf-passive results

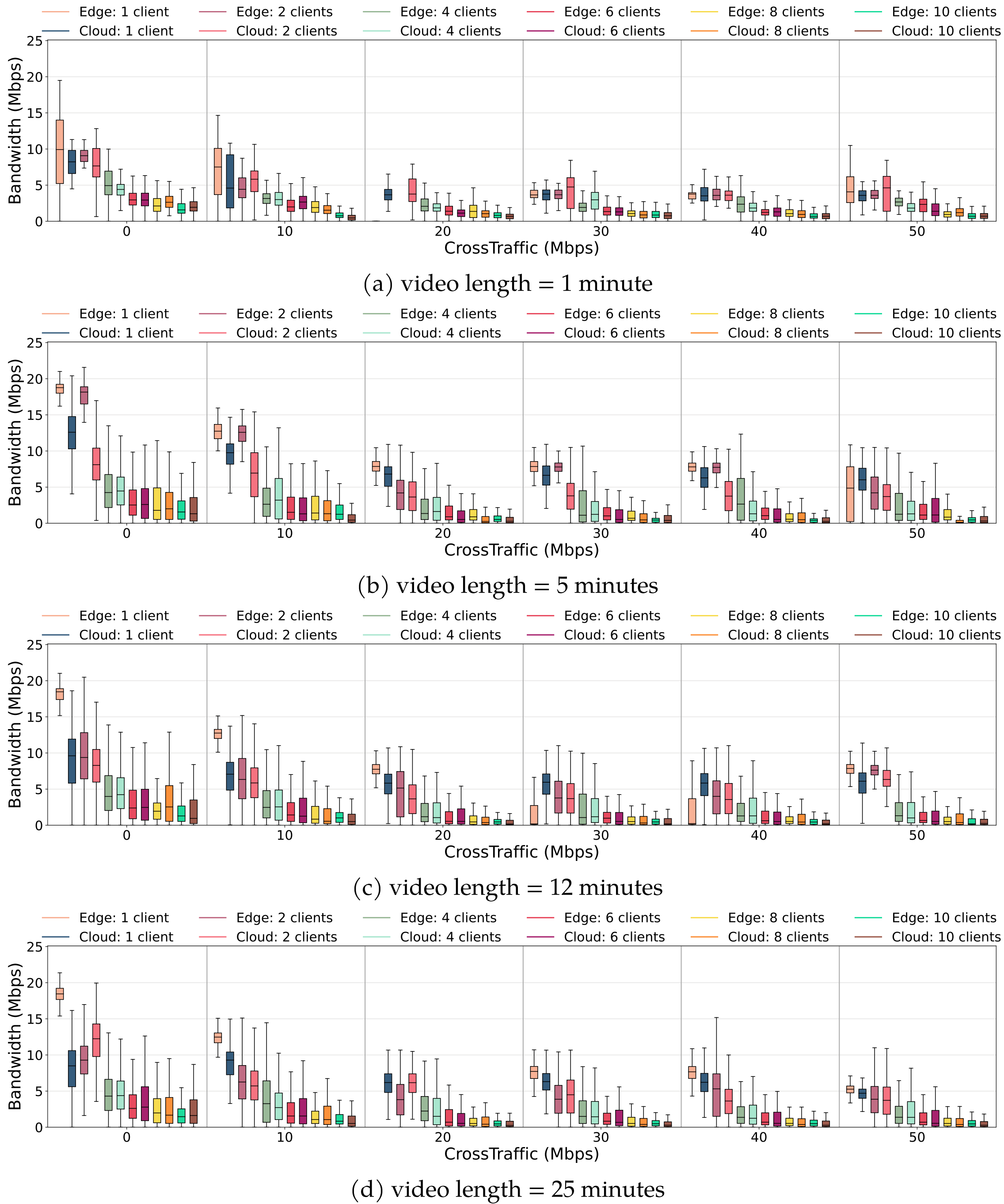
TCP MECPerf-passive bandwidth metrics have been generated at the MECPerf observer, considering the packet size and the time elapsed between the current packet and the previous one. In other words, each bandwidth metric is computed upon a

single packet. However, a bandwidth estimation based on a single packet could lead to estimation errors. To solve this problem, we grouped the raw bandwidth metrics into buckets of 0.5 seconds. Then, for each bucket, a single bandwidth metric has been computed.

Figure 5.7 shows the boxplot of MECPerf-passive bandwidth metrics obtained during Wi-Fi experiments, considering from 0 to 50 Mbps of cross-traffic and the four videos provided by the server. First, it can be noted that files of 5, 12, and 25 minutes present similar results, while the shortest video shows lower performance. Note that the DASH protocol operates over TCP connections. Consequently, the download starts with a modest bitrate due to the slow start mechanism. The first file contains only 1 minute of video, and it generally is downloaded in a few seconds, which is not sufficient to let the TCP connection reach the steady-state throughput. Then, the results obtained for this file will be omitted from now on.

Then, for the remaining files, it can be seen that the highest performance is obtained when there are a small number of clients, and there is no additional cross-traffic injected into the network. When moderate levels of cross-traffic are considered, the bandwidth metrics start to decrease. For cross-traffic values greater than 20 Mbps, the performance generally remains stable. This result is similar to the one obtained during MECPerf-active experiments, and it can be ascribed to the utilization level of the wireless link. Note that, even when no cross-traffic is injected into the access network, the bandwidth metrics decrease also when the number of clients increases. For example, let us consider the bandwidth metrics obtained when 12 minutes of video is downloaded (Figure 5.7c), and no cross-traffic is injected. As can be seen, when a single user is considered, median values of approximately 18 Mbps and 10 Mbps are obtained for the edge and cloud servers, respectively. Instead, the median values decrease to around 9 Mbps and 8 Mbps for two clients, while the median values decrease to approximately 4 Mbps for three clients. Two factors may contribute to this result. First, when the number of clients increases, the server has to manage a higher number of simultaneous downloads. This means that some requests may incur additional delays depending on the server's workload. Secondly, the traffic generated by each client can be considered as cross-traffic for the others. In other words, a higher number of clients leads to a higher utilization level of the wireless link and, consequently, to higher interference between the clients.

Instead, Figure 5.8 shows the MECPerf-passive results for LTE experiments. LTE experiments have been computed using up to 2 clients. This does not allow us to evaluate how the server workload affects the performance. However, it was impossible to collect data for a higher number of clients as they started to disconnect from the BS. As can be noted access-MEC metrics are higher than those collected for the access-cloud segment. Moreover, bandwidth values are generally slower when there is no cross-traffic injected into the access network and they do not seem to be affected



(a) video length = 1 minute

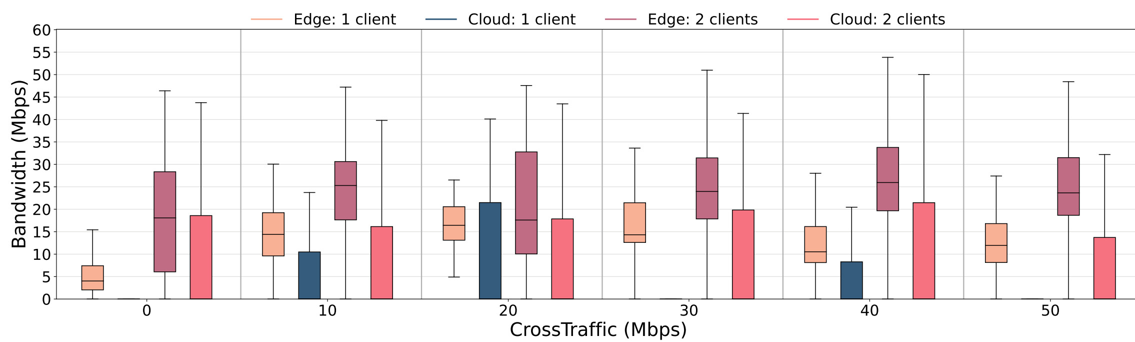
(b) video length = 5 minutes

(c) video length = 12 minutes

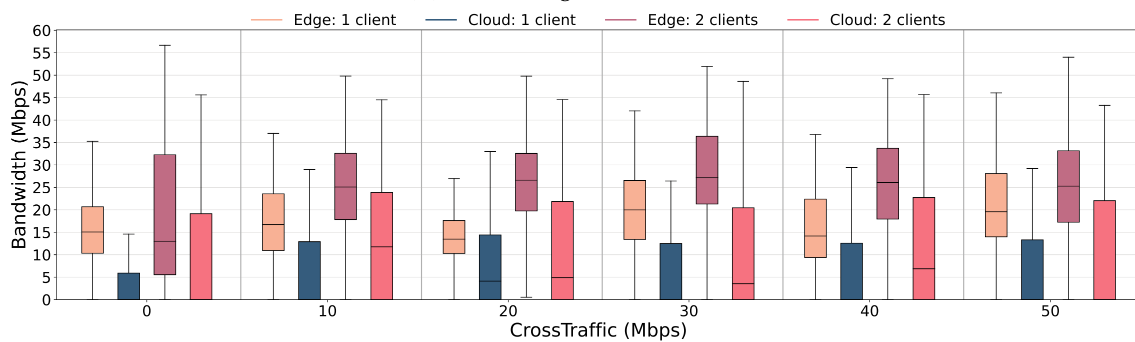
(d) video length = 25 minutes

Figure 5.7: The results collected during MECPerf-passive downlink bandwidth Wi-Fi experiments considering different levels of cross-traffic injected into the access network. The bandwidth values have been computed considering buckets of 0.5 seconds.

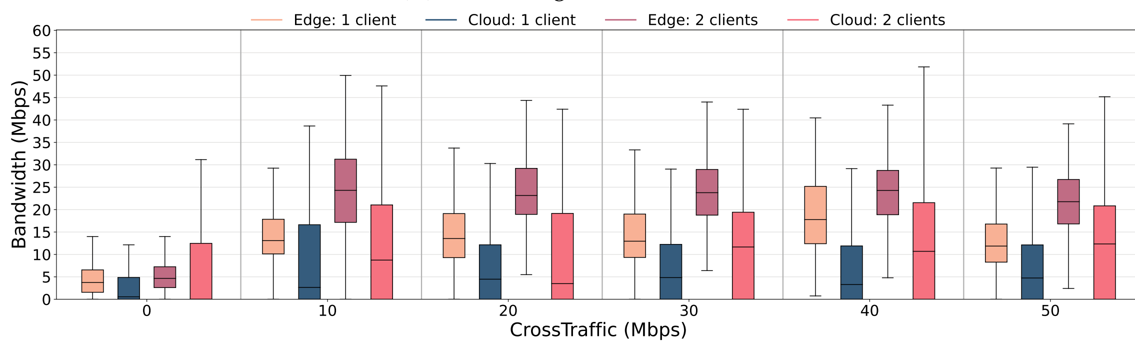
by other amounts of cross-traffic. This result is consistent with the MECPerf-active ones of Figure 5.3b.



(a) video length = 5 minutes



(b) video length = 12 minutes



(c) video length = 25 minutes

Figure 5.8: The results collected during MECPerf-passive downlink bandwidth LTE experiments considering different levels of cross-traffic injected into the access network. The bandwidth values have been computed considering buckets of 0.5 seconds.

TCP bandwidth: self-passive results

In this section, the bandwidth metrics collected during self-passive experiments will be illustrated. Figure 5.9 shows the metrics collected during Wi-Fi experiments, considering from 0 to 50 Mbps of cross-traffic and up to 10 DASH clients. As stated before, self-passive and MECPerf-passive metrics have been collected during the same experiments. Hence, Figures 5.9 and 5.7 can be directly compared. As can be seen, the self-passive metrics are always higher than the MECPerf-passive ones. This difference reflects the differences between the two measurement methods. Self-passive

(a) video length = 5 minutes

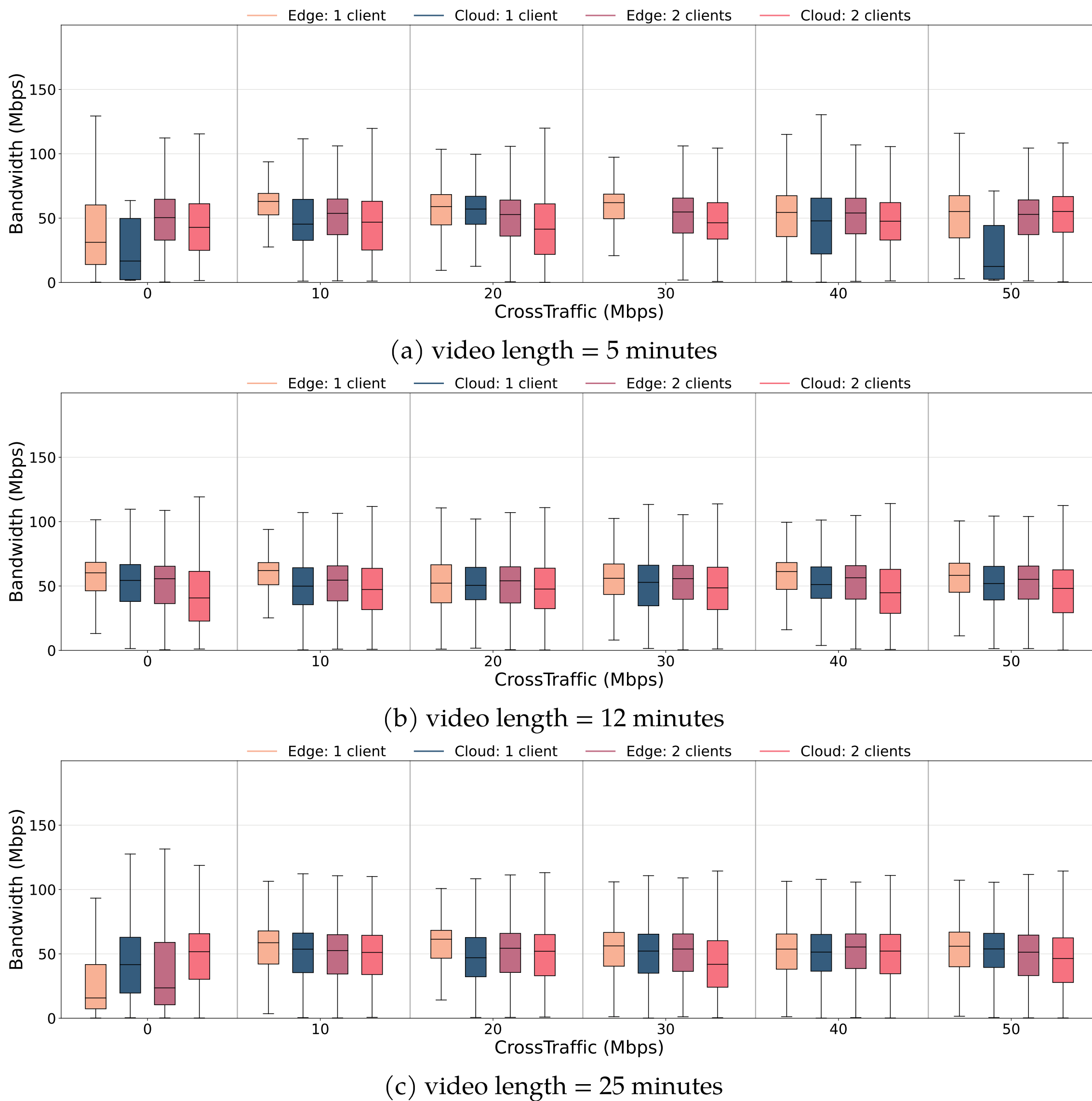
(b) video length = 12 minutes

(c) video length = 25 minutes

Figure 5.9: The boxplots of self-passive bandwidth metrics. The results are based on Wi-Fi downlink experiments and consider a scenario where a cross-traffic generator injects different levels of cross-traffic into the access network.

measurement methods are collected directly from the client application, which has the complete knowledge of the download status. This means that the client can compute the metrics only when the application effectively performs network operations, discarding idle periods. Instead, MECPerf-passive metrics are calculated by the MO and based on a network trace. The MO, which did not have any knowledge about the application logic, computes the metrics upon the whole execution as it cannot identify and discard idle periods. This explains why MECPerf-passive bandwidth metrics are smaller than the self-passive ones. Moreover, this highlights how self-passive measurement methods are required to compute bandwidth metrics, especially when applications with sporadic transmissions are considered.

As can we see from Figure 5.9 the median values of self-passive bandwidth metrics decrease as the amount of cross-traffic increases. This is coherent with the satu-



(a) video length = 5 minutes

(b) video length = 12 minutes

(c) video length = 25 minutes

Figure 5.10: The boxplots of self-passive bandwidth metrics. The results are based on LTE downlink experiments and consider a scenario where a cross-traffic generator injects different levels of cross-traffic into the access network.

ration of the wireless link. Instead, considering a fixed amount of cross-traffic and a high number of users, it can be noted that higher median values can be appreciated when a cloud server is adopted. For example, let us consider a scenario in which 10 clients download the file of 25 minutes (Figure 5.9c). When no cross-traffic is injected into the access networks and the edge server is involved, a median bandwidth value of approximately 5 Mbps can be appreciated. Instead, when the DASH server hosted into the cloud network is considered, the same scenario leads to a median value of approximately 18 Mbps. We believe that the higher computational capabilities of the machines located at the University of Pisa lead to this result. Finally, Figure 5.10 shows the results of LTE self-passive experiments. Also in this case,

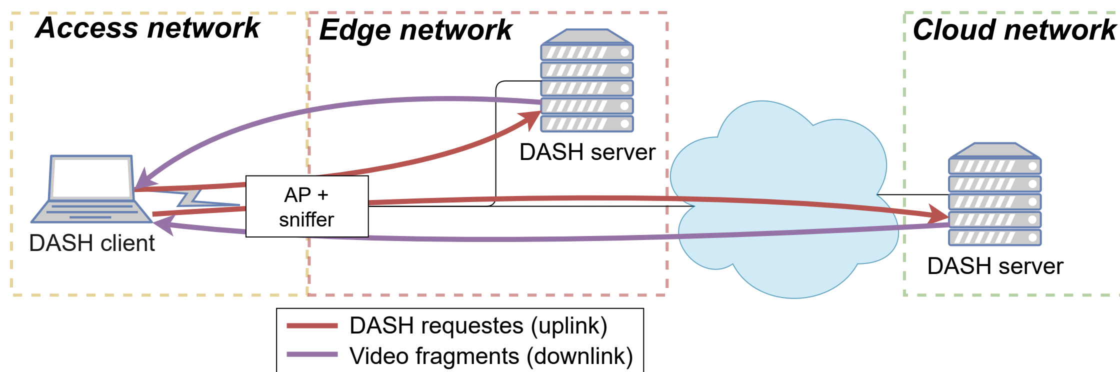


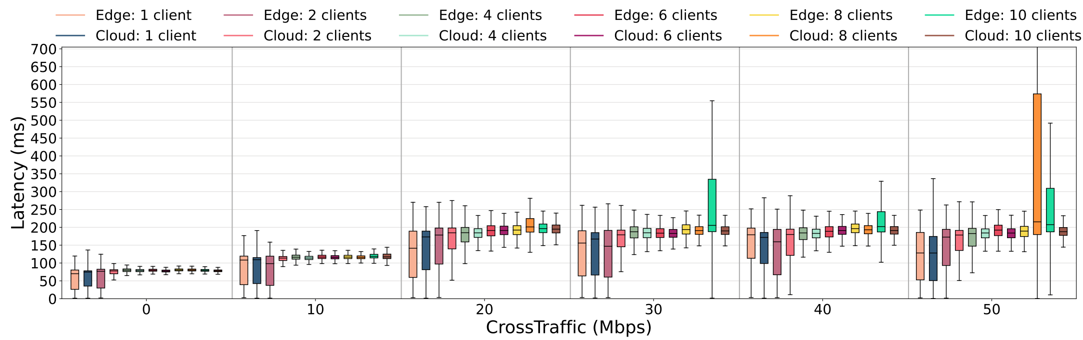
Figure 5.11: The uplink and downlink traffic during DASH Wi-Fi experiments.

bandwidth metrics are higher than those obtained during the MECPerf-passive ones (Figure 5.8). However, it can be noted that during self-passive experiments the metrics collected on the access-MEC and the access-cloud metrics show similar median values. This behavior, which is different from the one observed during MECPerf-passive experiments, can be explained by the client's ability to discard idle times. Finally, it can be noted that this behavior can also be observed for self-passive Wi-Fi experiments when a limited number of clients are involved.

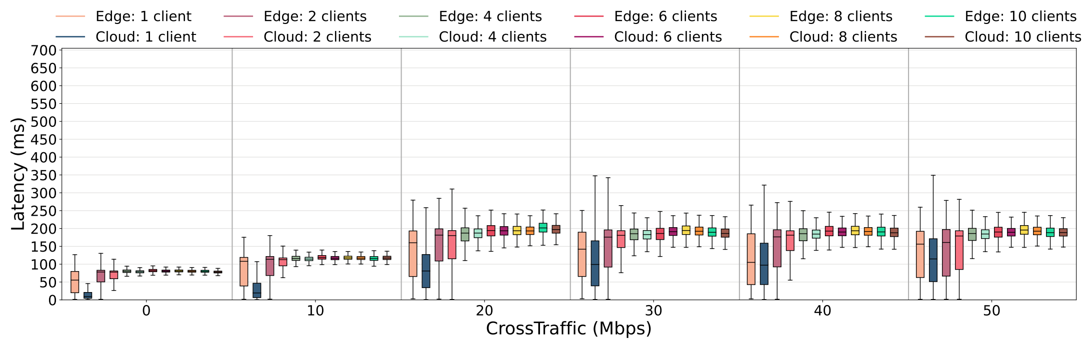
TCP latency: MECPerf-passive results

As stated in Section 4.1, MECPerf-passive latency metrics are computed as $t_{ACK} - t_{ACKED}$, where t_{ACK} and t_{ACKED} are the timestamps at which the AP sees the ACK and the ACKed packets, respectively. For Wi-Fi experiments, the traffic sniffer was deployed on the AP as shown in Figure 5.11. Then, uplink and downlink latency metrics can be computed as follows. Uplink latency metrics had been computed as the time elapsed between the timestamp in which the AP sees the DASH request sent from the client (t_{ACKED}) and the instant in which it receives the correspondent acknowledgment from the server (t_{ACK}). Hence, they can compute the performance on the wired segments connecting the AP with the MEC and the cloud DASH servers. Instead, downlink latency metrics were computed using video fragments sent from the server to the client. Precisely, they are computed as the time elapsed between the timestamp in which the AP sees the packet that contains the video fragment (t_{ACKED}) and the instant in which it receives the corresponding ACK from the client (t_{ACK}). Hence, they can be used to evaluate the performance of the wireless link that connects the client and the AP.

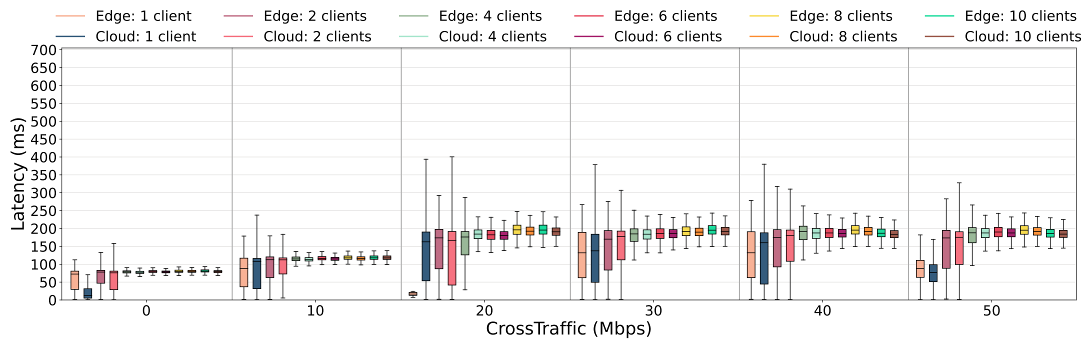
The results of downlink and uplink metrics for the Wi-Fi experiments are illustrated in Figures 5.12 and 5.13, respectively. As can be seen, downlink and uplink metrics present two different behaviors. In fact, downlink metrics increases as the amount of cross-traffic increase. Moreover, when more than 2 clients are consid-



(a) video length = 5 minutes



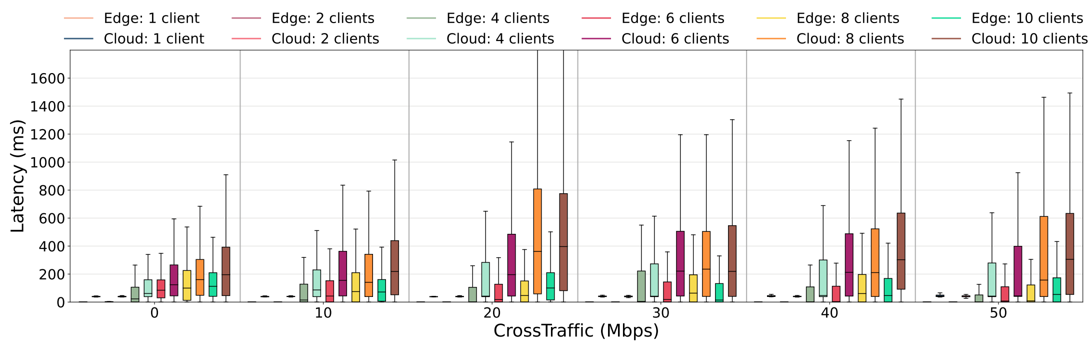
(b) video length = 12 minutes



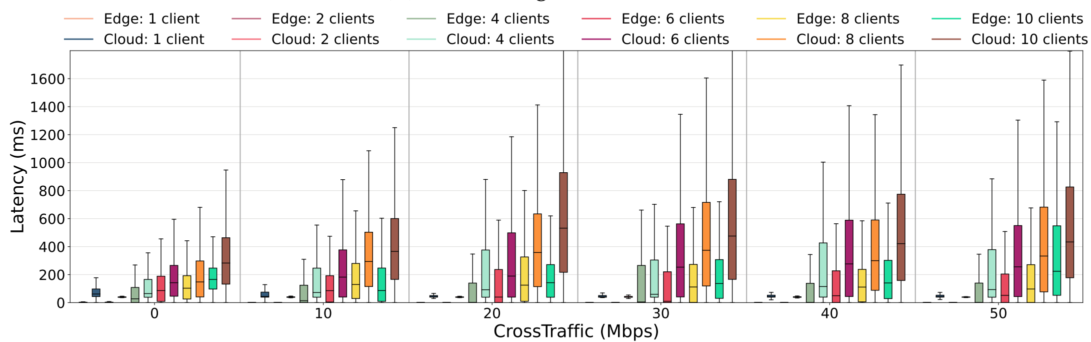
(c) video length = 25 minutes

Figure 5.12: The boxplots of MECPerf-passive downlink latency metrics, representing the performance of the wireless segment that connects the clients and the AP. The results are based on Wi-Fi experiments and consider different scenarios where a cross-traffic generator injects different levels of cross-traffic into the access network.

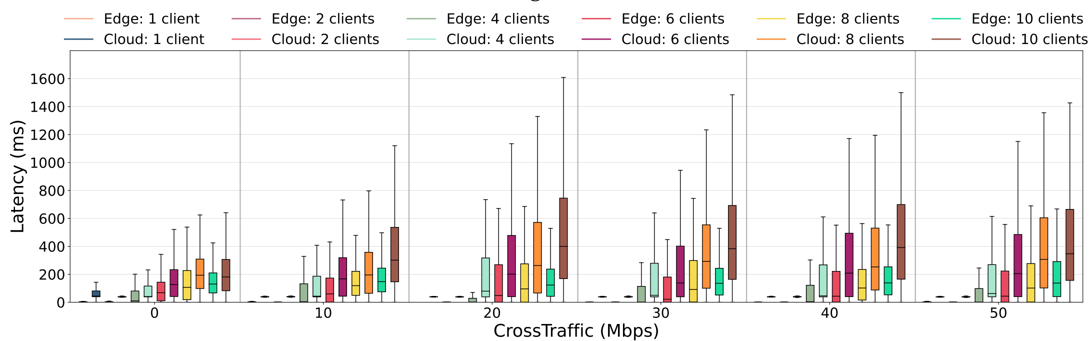
ered, their values become slightly higher and less variable. Conversely, uplink metrics increase when more clients are involved, while they appear to be marginally affected by the amount of cross-traffic considered. Moreover, uplink latency metrics are more variable than those computed using downlink traffic. Note that downlink uplink and downlink metrics compute latency metrics on different parts of the network. Downlink metrics are computed as the time needed to send a packet from the AP to the client and receive back the relative acknowledgment. Basically, downlink metrics reflect the performance of the access-MEC wireless segment and do not consider the server workload. As a consequence, this type of metric is highly affected



(a) video length = 5 minutes



(b) video length = 12 minutes



(c) video length = 25 minutes

Figure 5.13: The boxplots of MECPerf-passive uplink latency metrics, representing the performance of the wired network segments that connect the AP with both MEC and cloud servers. The results are based on Wi-Fi experiments and consider different scenarios where a cross-traffic generator injects different levels of cross-traffic into the access network.

by the utilization level of the wireless link but does not consider delays that can be attributed to a higher workload level on the server. Therefore, as can be noted in Figure 5.12, this type of metric is strongly influenced by the utilization level of the wireless link. Instead, uplink latency metrics evaluate the performance between the AP and the two servers. Therefore, since cross traffic does not involve the wired segment, it is easy to understand why the collected metrics are marginally affected by cross-traffic. Moreover, uplink shows a strong dependence on the number of customers and an increasing variability. This last behavior can be ascribed to two

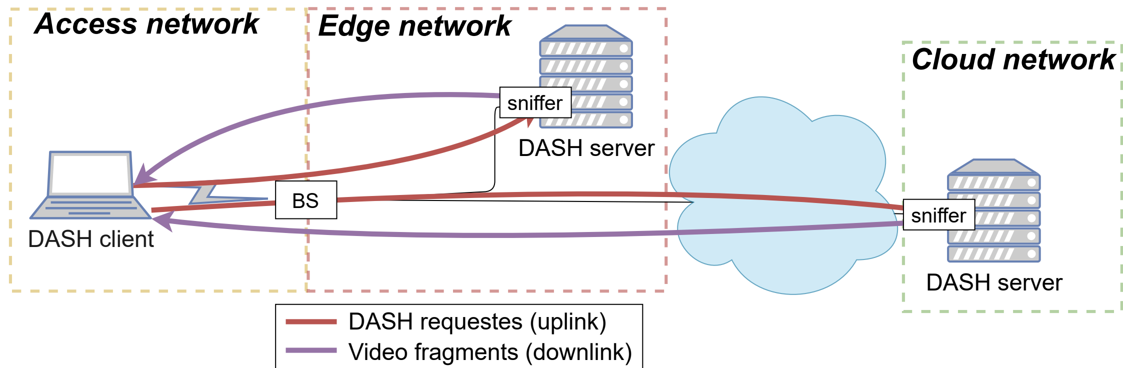
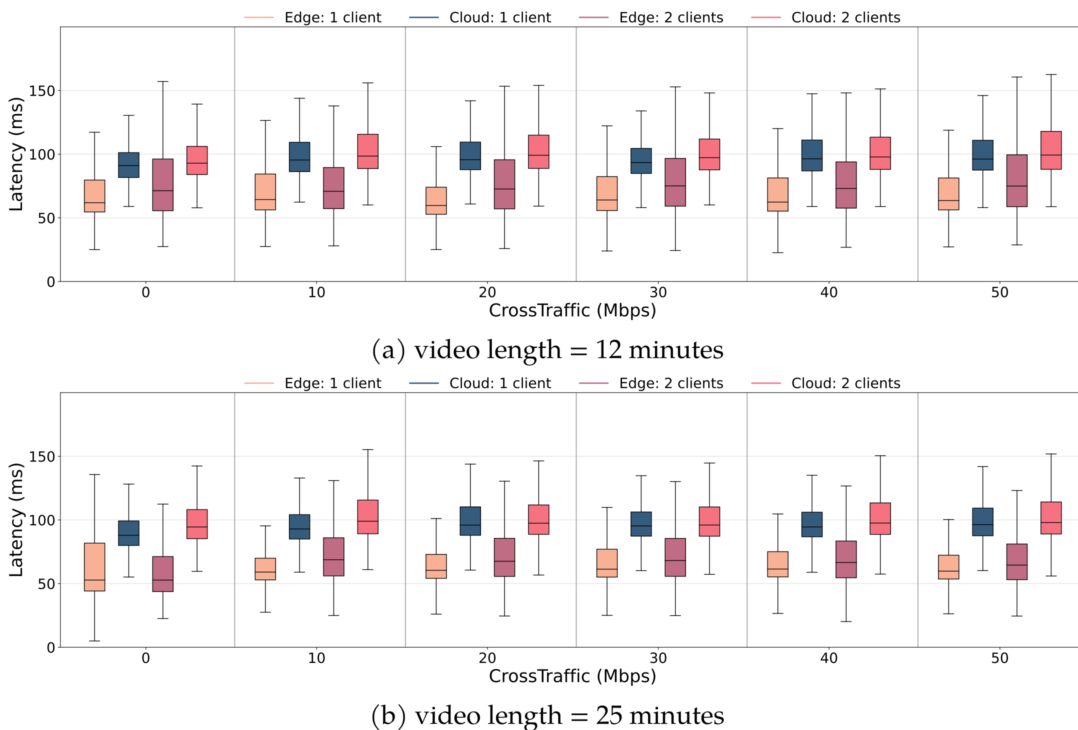


Figure 5.14: The uplink and downlink traffic during DASH LTE experiments.



(a) video length = 12 minutes

(b) video length = 25 minutes

Figure 5.15: The boxplots of MECPerf-passive downlink latency metrics, representing the performance of the wireless segment that connects the clients and the AP. The results are based on LTE experiments and consider a scenario where a cross-traffic generator injects different levels of cross-traffic into the access network.

different reasons. First, a higher amount of clients generates a higher workload on the server, introducing additional delays on incoming requests. Second, the traffic generated increases when the number of clients increases, saturating some links in the wired path.

As previously described, during LTE experiments the sniffer was placed in the destination server. Consequently, downlink latency metrics represent the time elapsed from the timestamp in which the AP sees a fragment of the video (t_{ACKED}) and the

instant in which it receives the corresponding ACK from the client (t_{ACK}). Hence, they can be used to evaluate the performance of both the access-MEC and the access-cloud segment. Conversely, uplink metrics are computed as the time that goes from the timestamp in which the sniffer hosted on the server sees the request for a fragment (t_{ACKED}) to the timestamp in which it detects the corresponding ACK (t_{ACK}). Hence, it simply represents the time needed to generate an acknowledgment packet on the server machine. Thus, they will be omitted. Figure 5.14 depicts the setup used.

Figures 5.15 depicts the latency metrics observed during the LTE experiments. As can be seen, the LTE latency metrics are generally stable for all the considered amounts of cross-traffic considered. This result should not surprise as the cross-traffic affect only the wireless part of the path, and also the MECPerf-active latency metrics show the same trend.

5.4 Lessons learned

To conclude, in the following the main results displayed in this Chapter will be outlined.

As regards the MECPerf-active metrics, it was observed that the performance of the network is strongly influenced by the placement, the state of the network, and the technology used. The results of the experiments based on MECPerf-active latency metrics demonstrated that access-MEC metrics are not only lower than those collected on the access-cloud server but also less affected by the presence of cross-traffic. Therefore, applications with stringent latency requirements should be placed within the edge networks, whenever possible. In addition, WI-Fi-based latency metrics showed values below 5 milliseconds, while latency metrics collected on the MEC-cloud segments are approximately equal to 40 milliseconds. These results revealed the benefits introduced by edge nodes but they also showed the high cost of propagating context data between edge servers and centralized remote repositories. Similarly, as expected, the access-MEC segment performs better than the MEC-cloud segment in terms of MECPerf-active bandwidth metrics. Therefore, it would be preferable to deploy application servers on edge nodes wherever possible. However, it can also be noted that the difference in the bandwidth metrics collected on the access-MEC and the access-cloud network segments is often smaller than the differences exhibited by the latency metrics on those segments. Thus, in the presence of congested edge nodes, it may be preferable to relocate bandwidth-intensive applications. Finally, the results of experiments based on MECPerf- and self-passive measurement methods revealed that cloud servers provided higher performance in terms of both bandwidth and latency metrics when a large number of clients was considered. Fundamentally, this result is due to the higher computational capa-

bilities of cloud servers, which succeeded in compensating for the increased communication latency. This result indicates how important it is to consider the server workload during the deployment of application servers.

Chapter 6

Use MECPerf experimental results to build a simple trace-based network simulator

As stated in Chapter 2, most of the edge simulators currently available make use of simple network models. For instance, some simulators consider bandwidth and/or latency metrics as fixed characteristics of a link. Other frameworks are based on the simulation of just a part of the network stack. However, these approaches may not reflect the complexity of a real system. To fix this issue a trace-based simulator can be employed. Essentially, a trace-based network simulator retrieves network performance metrics from pre-generated traces. Therefore, to obtain noteworthy results, it is necessary to use traces that are relevant for the scenario under analysis and as complete as possible. This approach offers some advantages.

- The use of traces allows to include in the simulation phenomena that are difficult to consider. For example, they can introduce unexpected interactions between hosts belonging to the system or they can reflect the influence of traffic generated by hosts outside the system. Therefore, trace-based approaches allow the performance evaluations of applications and placement strategies in a real-world scenario.
- A trace advances over time in a totally deterministic way. This enables the evaluation of different applications and orchestration strategies under the same network conditions.
- Trace-based simulators are less computationally demanding than other types of simulators.

The rest of this Chapter is organized as follows. First, it will be presented the MECPerf Library. Basically, the library provides an API to produce bandwidth and

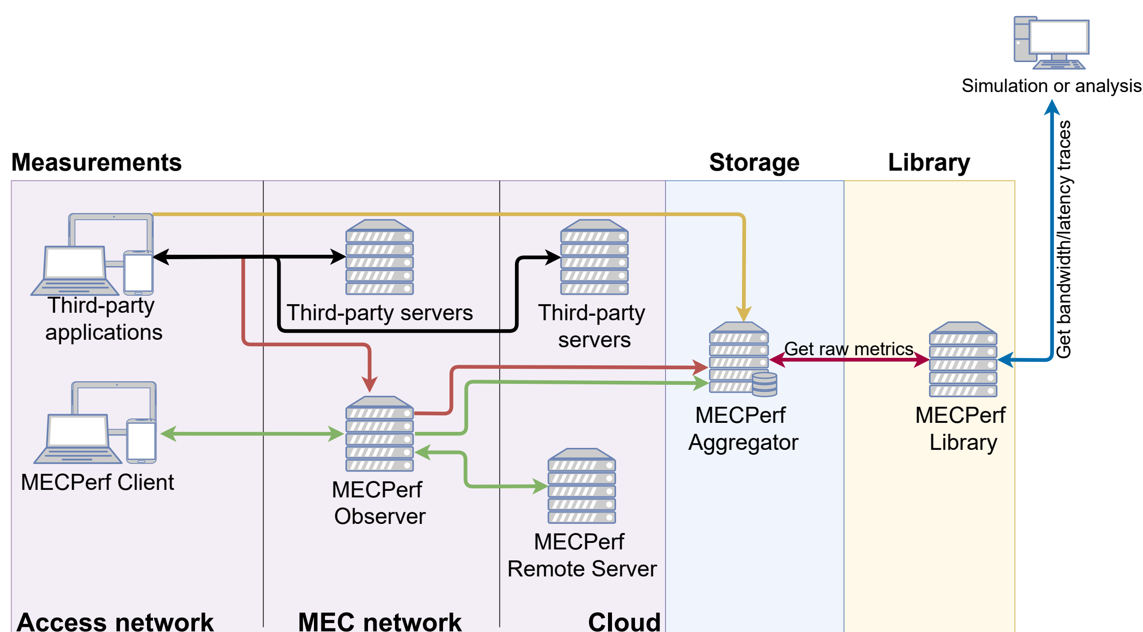


Figure 6.1: The interaction between the MECperf collection tools, the MECPerf Library, and third-party software that requires metrics to compute simulation or analysis activities.

latency traces starting from the metrics stored in the dataset containing the results of the experiments collected using MECPerf. Then, some insights about the usage of the MECPerfLibrary will be provided.

6.1 Using the experimental results to generate input traces

MECPerf was used to collect a set of network metrics in an edge environment under different network conditions. The setup used to collect the metrics and the results of the experimental campaign have been provided in Chapter 5. Moreover, a dataset containing the results of the experiments conducted using MECPerf has been made publicly available. However, for programmers can be challenging to access directly to the raw metrics stored since this operation required a complete knowledge of the dataset's internal structure. This makes the metrics difficult to be used. Hence, the MECPerfLibrary was developed with the goal of simplifying the access to the metrics contained within the dataset as much as possible.

The MECPerfLibrary architecture

The library interacts with MECPerf as shown in Figure 6.1. The left side of the graph shows the MECPerf components (i.e., the MC, the MO, the MRS, and the MA) and how they interact to collect MECPerf-active, MECPerf-passive, and self-passive metrics. The raw metrics collected during the MECPerf experiments have been stored in a MySQL database administrated by the MECPerf Aggregator. Further details about the interactions between the components and the collection methods have been found in Chapter 4. Instead, the right side of the graph shows the MECPerfLibrary. As can be noted the library can access the raw metrics obtained through the measurement campaign. Precisely, the MECPerf Library is composed of two different components. The first one is a software utility that can be used to retrieve the raw metrics contained within the database, generating intermediate bandwidth and latency input files. Instead, the second one is the NetworkTraceMenager class. Basically, this last component takes the intermediate input files, using them to produce bandwidth and latency traces. Once generated, the traces can be used to emulate a MEC network under specific network conditions (for example, when a particular amount of cross-traffic is injected into the access network or when a specific access technology is used to connect the TNs to the rest of the network). Finally, the NetworkTraceManager module provides a set of APIs that can be used by programmers to obtain trace-based network metrics.

The NetworkTraceManager module

The NetworkTraceManager (NTM) class is the principal module within the MECPerf library. It exposes APIs to generate the traces and obtain bandwidth and latency metrics. Whenever a software component requires a new trace, it has to instantiate a new NTM object. During the initialization, the NTM instance receives as input the path of a JSON file, a network setup descriptor, and two seeds. The JSON file is used to associate the raw metrics with the setup used during the experiment that collected them (e.g., it specifies the type of the experiment, the access technology used, the amount of cross-traffic injected into the access network, etc.). The network setup descriptor indicates the network configuration in which the caller is interested. It is composed of the type of measurements, the transport layer protocol, the measured network segment, the identity of the hosts involved in the measure, the direction of communication, the access technology used to connect hosts in the access network, and the amount of cross-traffic injected into the access network. However, not all the parameters have to be specified. Finally, the two seeds are used to guarantee the repeatability of the experiments. The Library stores the raw metrics into multiple files, where each of them contains homogeneous couples of <*raw metrics, timestamps*> pairs collected under the same experiments and network conditions. After check-

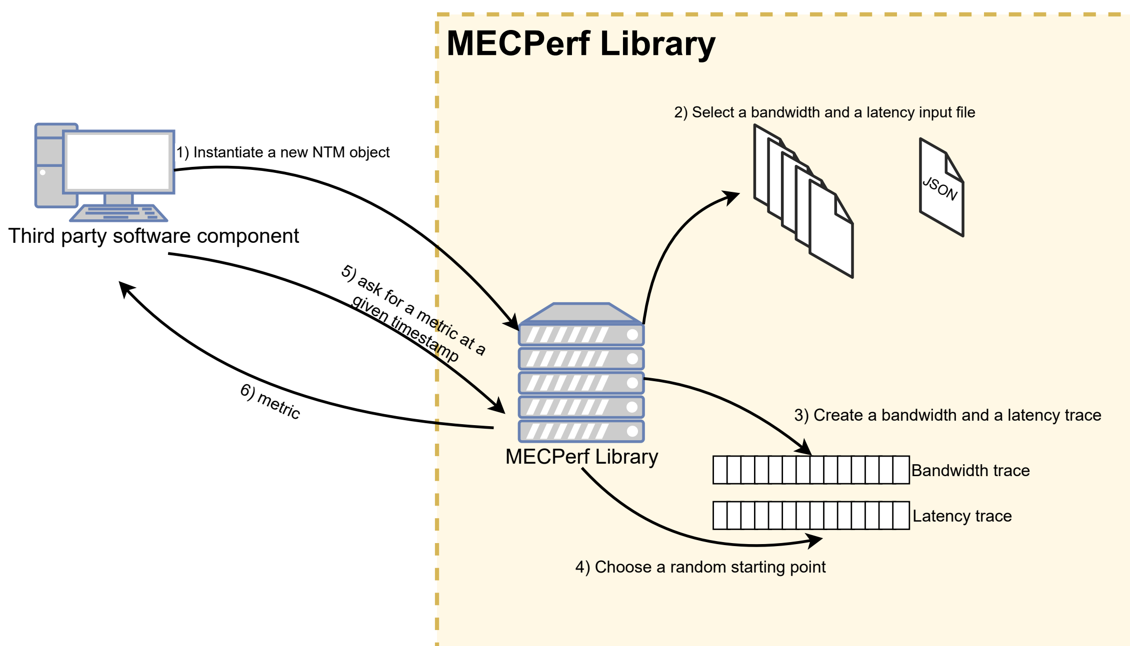


Figure 6.2: The interaction between the MECPerf Library and a generic third-party software components.

ing the validity of the received input, the NTM instance uses the JSON files to select all the input files with a setup that matches the network setup descriptor. Note that several input files may be selected at this stage as experiments have been repeated multiple times. Furthermore, the programmer may have expressed only a part of the network setup descriptor. Consequently, input files containing raw values collected under different configurations may be returned. At this point, the NTM instance uses the seeds to initialize two random number generators. The first seed is used to select one bandwidth and one latency input file among the ones available. Then, the NTM instance generates bandwidth and latency traces from the two selected files. Instead, the second seed is used to choose a starting point randomly within the trace. Finally, it must be remarked that the traces had been generated starting from real-world experiments. Thus, the collected metrics are not perfectly spaced equally. Furthermore, some metrics may be missing due to unpredictable failures during the collection phase. To bound the possible side effects introduced by protracted losses in the collection of metrics the two traces have been post-processed so that two consecutive elements cannot differ by more than $MAX_{TRACEGAP}$ unit of times. This completes the initialization of the NTM instance. From this time on, the programmer can ask for network metrics using methods provided by the library. Note that the raw metrics have been collected through experiments. Hence, only some timestamps have associated metrics. To overcome this restriction, the traces are managed following a sample-and-hold strategy. Finally, when the last trace el-

ement has been asked, the NTM shifts the timestamp associated with each value contained in the trace. In other words, <*raw metric, timestamp*> couples are managed as a circular array. The Library also aligns the bandwidth and latency traces when collected samples are not perfectly synchronized. The behavior of the NTM module is pictured in Figure 6.2.

The MECPerf library had been implemented in Python, and it is available on GitHub(The source code of the MECPerf library, 2021).

A simple example of usage

The following code shows a simple example of the usage of the library.

```
1  import configparser
2  from network_trace_manager import NetworkTraceManager
3
4  config = configparser.ConfigParser()
5  config.read("conf.ini")
6
7  network_traces = NetworkTraceManager(config["configuration_name"])
8  print(network_traces.get_rtt(1))
9  print(network_traces.get_bandwidth(3.1))
10 print(network_traces.get_networkvalues(2.3))
11
12 for e in NetworkTraceManager.get_tracelist("inputFiles/mapping.json",
13                                            command="TCPRTT",
14                                            direction="upstream",
15                                            typeofmeasure="active"):
16     print (e)
```

At first, the programmer must provide an ini file containing the targeted network setup. This operation is performed at Lines 4-5. Then, it is possible to call the NetworkTraceManager constructor (Line 7). The constructor takes as input a network configuration and returns a NetworkTraceManager object containing a bandwidth and a latency trace. From this time on it is possible to ask for network metrics. The library provides three different methods that can be used by programmers to retrieve network metrics.

- Line 8 - ***get_rtt***(***sec***): gets as input a floating number *sec* and push forward the simulated time by *sec* seconds. Returns a negative number if an error occurs.

Otherwise, it returns a python list composed of the latency metric associated with the current simulated time, a datetime object containing the current simulated time, and a datetime object containing the timestamp at which the metric was collected.

- Line 9 - ***get_bandwidth*(*sec*)**: gets as input a floating number *sec* and push forward the simulated time by *sec* seconds. Returns a negative number if an error occurs. Otherwise, it returns a python list composed of the bandwidth metric associated with the current simulated time, a datetime object containing the current simulated time, and a datetime object containing the timestamp at which the metric was collected.

- Line 10 - ***get_networkvalues*(*sec*)**: gets as input a floating number *sec* and push forward the simulated time by *sec* seconds. Returns a negative number if an error occurs. Otherwise, it returns a python list composed of both the latency and the bandwidth metric associated with the current simulated time, a datetime object containing the current simulated time, and a datetime object containing the timestamp in which the metric was collected.

Finally, the library provides the *get_tracelist*() method, which takes as input the path of the mapping file and other parameters. The method scans the mapping file and returns the list of configuration that matches the provided input. Basically, this method can be used by programmers to explore the set of available configurations. An example of the usage of this method can be found in Lines 12-16.

6.2 Use the NetworkTraceManager to implement a simple simulator

The MECPerf Library had been used in a paper to implement a simple network simulator in an edge environment (Caiazza et al., 2021).

The work considered the scenario depicted in Figure 6.3 where a TN was capable of obtaining network connectivity from two different edge networks. In this scenario, the MC was supposed to be running on the TN, while each edge network is supposed to contain an MO and a MA. Finally, there was supposed to be present an orchestrator capable of retrieving network metrics from the two MAs in real-time and capable of migrating a fraction of clients γ with the poorest performance from one edge to the other one.

The paper presented a time-discrete simulation, performed considering 100 independent clients. Each client interleaved active and inactive periods and the duration of each period was randomly chosen following an exponential distribution with a mean of 10 slots. Precisely, at the beginning, each client obtained two latency

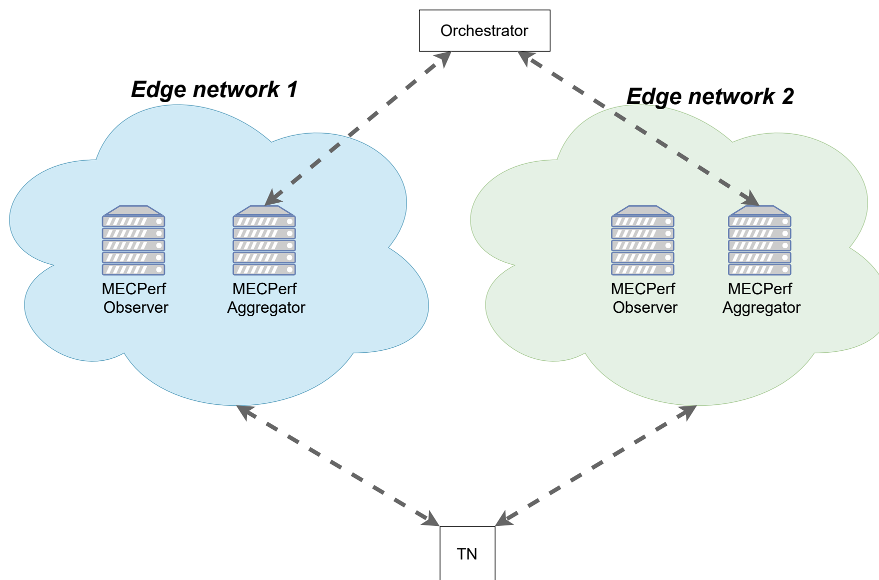


Figure 6.3: The simulated architecture.

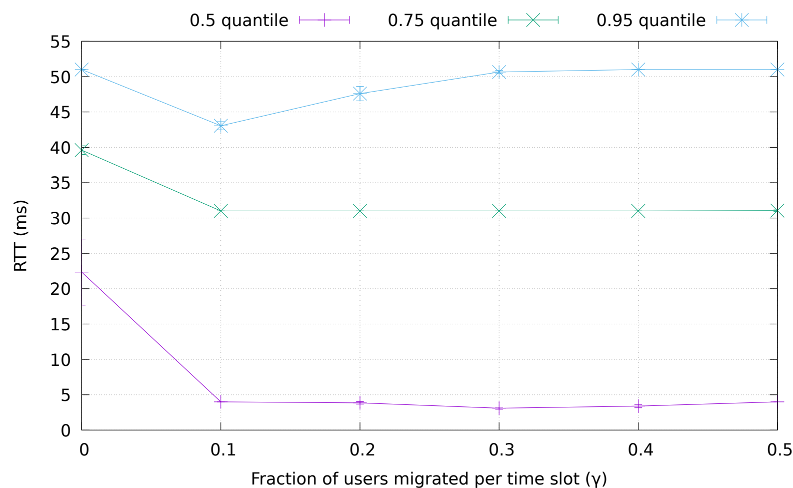


Figure 6.4: The 0,5, 0,75, 0,95 RTT quantiles for γ values that goes form 0 to 0,5. Values are plotted considering a 95% confidence interval.

traces for the two edge networks. The setup used to generate each trace considered both LTE and Wi-Fi experiments and values of cross-traffic that go from 0 Mbps to 50 Mbps. Then the TN was connected to an edge network randomly selected. Finally, at the end of every slot, the orchestrator collected the latency metrics from the subscribed clients. Then, a fraction γ of users characterized by high RTT metrics were forced to migrate to the other operator, while the clients with smaller RTT values were retained.

Figure 6.4 shows the 0,5, 0,75, and 0,95 quantiles of RTT computed upon 20 independent repetitions of each scenario. Values have been plotted considering a 95%

confidence interval and γ values that goes from 0 (i.e., no migration) to 0,5 (i.e., at every time slot half clients are migrated to the peer operator). First, it is possible to see that the RTT shows a non-monotone trend in each quantile. This indicates that a low migration can keep the RTT low, while high migration obtains the opposite result. For example, considering the 0.95 quantile, the RTT decreases from 50 milliseconds to approximately 43 milliseconds where γ goes from 0 to 0,1. Then, when γ increases, the RTT increases as well. For γ equal to or greater than 0,3 the RTT returns to approximately 50 milliseconds. In general, the optimal values of γ are equal to 0,1 for all the considered quantiles.

The results presented in (Caiazza et al., 2021) make clear how edge orchestration strategies can benefit from the collected network metrics. The paper also demonstrates how traces collected in the real world can be used to feed a simple simulator.

Chapter 7

Estimating the energy consumption of terminal nodes in edge/cloud scenarios

In this Chapter the energy consumption of a client-server application operating in a MEC environment following a request-response scheme will be evaluated. Let us consider the reference architecture of Figure 7.1. The application is assumed to be running on TNs, which are hosted in the access network and connected to both edge and cloud servers using LTE connections. Assessing the client-side energy consumption helps to understand the server's location's impact on TN's energy consumption. As stated in Section 2.2, several offloading strategies have been proposed in the literature. Most of them have been evaluated with the aim of minimizing the energy consumption of the server infrastructure, while the energy consumption of TNs is generally not sufficiently investigated in those works. However, it is reasonable to assume that the energy consumption of a TN can be affected by the communication latency between the device and the application server. Therefore, a deeper understanding of the relationships between communication latency and power consumption of TNs would allow improving the design of energy-efficient application placement strategies.

The rest of this Chapter is organized as follows. First, the FSM of an LTE interface is provided. Then, two models will be given. The first one is an analytical model suited to evaluate the energy consumption of a connectionless application. Instead, the second one is a hybrid model used to assess the energy consumption of a connection-oriented client application. Finally, the energy consumption of the two models will be estimated considering both edge and cloud servers.

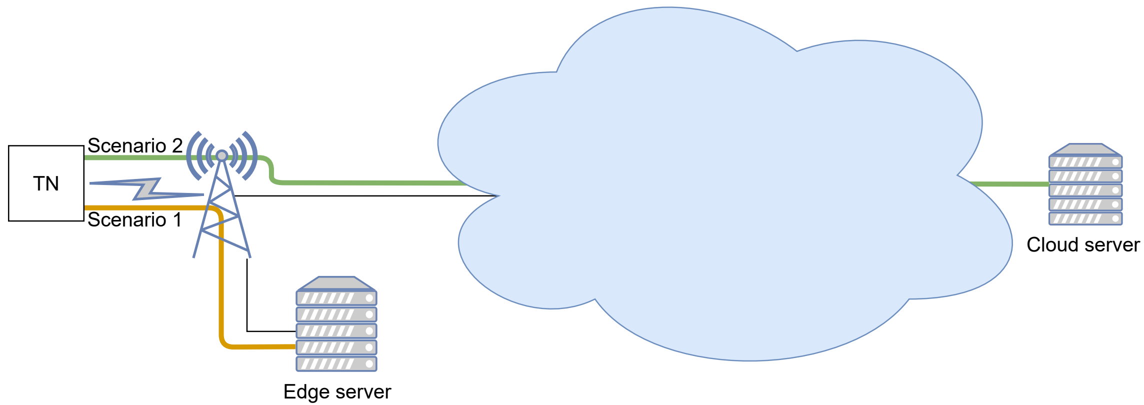


Figure 7.1: The reference architecture considered in this Chapter. The client application is running on TNs hosted in the access network using LTE connections to communicate with servers located on both edge and cloud networks.

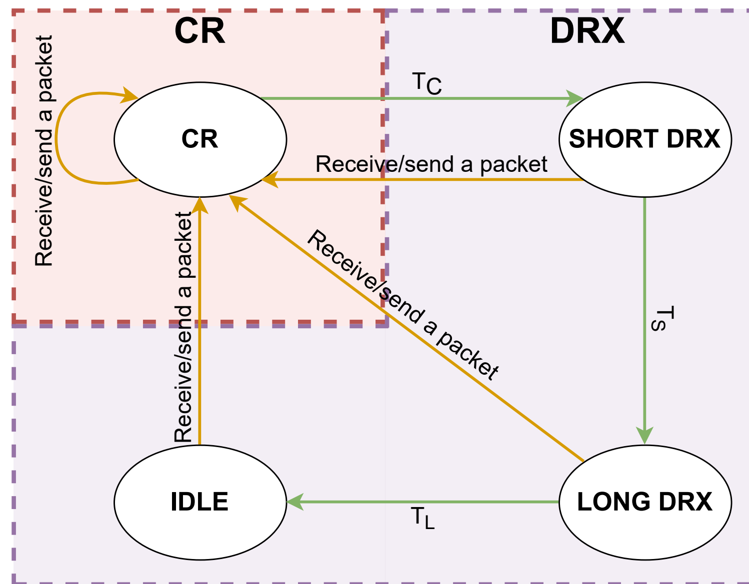


Figure 7.2: The finite state machine of an LTE module.

7.1 Modeling the LTE interface as a finite state machine

The LTE interface of TNs has been modeled using the FSM shown in (Huang et al., 2012) and (Chen et al., 2015). Fundamentally, an LTE interface operates according to two different modes. The former is the *Continuous Reception* (*CR*) mode, while the latter is the *DRX* mode. During CR, the interface is continuously switched on, waiting to send or receive packets. Therefore, CR is characterized by the highest energy consumption level. Instead, during DRX, the interface alternates sleeping

Phase	Symbol	Mean power consumption (mW)
CR when sending	P_{TX}	1200
CR when receiving	P_{RX}	1000
CR when idle	P_C	1000
SHORT DRX	P_S	359.07
LONG DRX	P_L	163.23
IDLE	P_I	14.25
Promotion	P_{PROM}	1200

Table 7.1: The mean power consumption of the LTE interface adopted.

and wake-up periods. The sleeping periods reduce the energy consumption of the network interface, but they also introduce additional delays since no packet can be received during sleeping periods. The DRX mode is composed of three different states characterized by different duty cycles. The first state is called SHORT DRX, which is characterized by small sleeping periods. Hence, energy consumption is lower than the one experienced during CR, however additional delays are also been introduced. The second state is called LONG DRX. During this state, the sleeping periods are longer than those experienced during SHORT DRX. Consequently, less energy is needed to remain in the LONG DRX, but higher delays are introduced. Finally, the third state is called IDLE. In this case, the interface sleeps most of its time. As a result, the energy consumption is minimized and the highest delays are introduced.

Figure 7.2 illustrates the FSM of the LTE interface. As can be seen, every time a packet needs to be sent or received, the interface goes into CR. Once the packet has been sent (or received), the interface remains in CR for an additional time T_C. If no transmission occurs during TC, the interface goes into SHORT DRX and starts to interleave sleeping and wake-up periods. The interface remains into SHORT DRX for at most T_S units of time. If a packet is sent or received during SHORT DRX, the interface returns to CR. Otherwise, it goes into LONG DRX. This state is similar to SHORT DRX as it also operates according to a duty cycle. However, sleeping periods are longer. Consequently, the energy consumption decreases while higher delays may be applied on incoming packets. Finally, if no packets are sent or received for T_L, the interface enters into IDLE. In this last state, the interface sleeps for most of the time. Consequently, this state is characterized by the smallest consumption, but it can also introduce higher delays. Note that timers do not trigger any transition from IDLE. Thus, the interface goes back to CR only when a new packet needs to be sent or received.

In the rest of this Chapter, we will consider the LTE interface described by Chen et al. (2015) and characterized by the following operational parameters. Let P_C,

Phase	Symbol	Duration (ms)
CR when idle	T_C	200
SHORT DRX	T_S	400
LONG DRX	T_L	11000
Promotion	T_{PROM}	200

Table 7.2: The maximum time spent in each state of the adopted FSM.

P_S, P_L, and P_I the power consumption needed to stay in CR without transmitting any data, in SHORT DRX, in LONG DRX, and in IDLE, respectively. T_C, T_S, and T_L are equal to 200, 400, and 11000 milliseconds, respectively. When the interface is idle in CR, then P_C is equal to 1000 mW. Instead, the energy needed to stay in CR during a data transmission depends on the Reference Signal Received Power (RSRP). However, for simplicity, we considered a mean power consumption of 1200 mW for the sending phase and 1000 mW for the receiving phase. During SHORT DRX, the interface required 788 mW during a wake-up period of 41 ms and 61 mW during the sleeping period. The interface wakes up with a period of 100 ms. During LONG DRX, the interface required 788 mW during a wake-up period of 45 ms and 61 mW during the sleeping period. The interface wakes up with a period of 320 ms. Finally, in IDLE, the interface consumed 570 mW during the wake-up period of 32 ms, while the energy consumption during the sleeping phase is negligible. During IDLE, the interface woke up with a period of 1280 ms. For simplicity, we considered the mean power consumption of 359.07 mW, 163.23 mW, and 14.25 mW for P_S , P_L, and P_I, respectively. To conclude, it should be said that the transitions from SHORT DRX and LONG DRX to CR are performed immediately. In contrast, transitions from IDLE are completed only after a *promotion* period. The power consumption needed to promote the interface from IDLE to CR is the same for both the sending and the receiving phases. Hence, $P_{PROM_{TX}} == P_{PROM_{RX}} == P_{PROM}$. The considered interface has a P_{PROM} equal to 1200 mW for 200 ms. Tables 7.1 and 7.2 summarized the parameters of the considered interface.

7.2 Estimating the energy consumption of a connectionless application

Modeling a connectionless request-response schema

The energy consumption of a connectionless application has been evaluated considering the request-response schema illustrated in Figure 7.3. Let T_I be the application period. As can be seen, at the beginning of each period, the client sends to the server

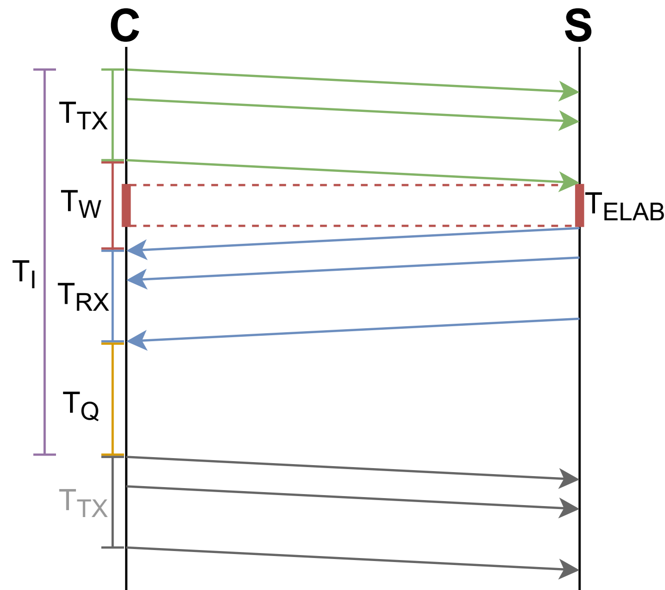


Figure 7.3: The interaction between the client and the server considering a connectionless application operating on an application period T_I.

a message. This message can represent a small request for a resource that needs to be downloaded, or it can represent the upload of some data to a centralized repository. In general, the message requires a time T_{TX} to be transmitted. For a connectionless communication, T_{TX} depends on the amount of data that needs to be sent and the bandwidth available on the wireless segment. Once the transmission has been completed, the client remains idle for T_W, waiting to receive a response from the server. In other words, T_W is the time between the end of T_{TX} and the beginning of T_{RX}. We assumed to have an idle server capable of immediately processing incoming messages, generating a reply in a time T_{ELAB}. Then, T_W can be rewritten as

$$T_W = T_{ELAB} + RTT \tag{7.1}$$

This suggests that T_W depends on the location of the destination server. Consequently, the energy consumption E_W, needed during T_W, depends on the server's location. Finally, once the response has been fully received in a time T_{RX}, the client remains idle for a time T_Q, waiting for the beginning of a new period. In other words, T_Q can be written as:

$$T_Q = T_I - T_{TX} - T_{RX} - T_W - T_{PROM_{TX}} - T_{PROM_{RX}} \tag{7.2}$$

Then, given Equation 7.1 also T_Q depends on the RTT. This means that also the energy consumption E_Q, needed during T_Q, depends on the server's location.

Modeling the client-side energy consumption

Let E_{TX}, E_W, E_{RX}, and E_Q be the energy spent during T_{TX}, T_W, T_{RX}, and T_Q, respectively. Instead, let $E_{PROM_{TX}}$ and $E_{PROM_{RX}}$ be the energy required during the promotion period eventually needed before T_{TX} and T_{RX}, respectively.

The energy E_I can be estimated as the sum of the energy spent during the periods that form T_I plus the energy employed to promote the interface from IDLE to CR eventually. Then, E_I can be computed as:

$$E_I = E_{TX} + E_W + E_{RX} + E_Q + E_{PROM_{TX}} + E_{PROM_{RX}} \tag{7.3}$$

First of all, E_{TX} can be computed considering both the time needed to send the message to the server and the power required to send data.

$$E_{TX} = T_{TX} \cdot P_{TX} \tag{7.4}$$

Similarly, E_{RX} can be computed as:

$$E_{RX} = T_{RX} \cdot P_{RX} \tag{7.5}$$

Note that when packets need to be sent or received, the interface remains in CR. Hence, E_{TX} and E_{RX} are always characterized by high power consumption.

To compute E_W, two things have to be noted. Firstly, at the beginning of T_W, the interface is still in CR. Secondly, there are no packets transmitted during T_W. Consequently, depending on the length of T_W, the FSM may perform several transitions. Precisely, if T_W is smaller than TC, the interface remains in CR for the whole interval T_W. Hence, E_W can be computed as:

$$E_W = T_W \cdot P_C$$

Otherwise, the interface stays in CR for T_C. Then it goes in SHORT DRX for the residual time. If T_W is smaller than $T_C + T_S$ the interface completed its T_W in SHORT DRX. In this case, E_W can be computed as

$$E_W = T_C \cdot P_C + (T_W - T_C) \cdot P_S$$

Instead, if T_W is larger than $T_C + T_S$, the interface goes from SHORT DRX to LONG DRX. Then, if T_W is shorter than $T_C + T_S + T_L$ the interface completes its T_W when it is still into LONG DRX. Then, E_W can be computed as

$$E_W = T_C \cdot P_C + T_S \cdot P_S + (T_W - T_C - T_S) \cdot P_L$$

Finally, if $T_C + T_S + T_L$ is not sufficient to complete T_W the interface enters into IDLE. Note that the interface remains in IDLE until T_W is concluded as the interface exits

from IDLE only when a packet is sent or received. In this last case, E_W is computed as

$$T_C \cdot P_C + T_S \cdot P_S + T_L \cdot P_L + (T_W - T_C - T_S - T_L) \cdot P_I$$

To summarize, E_W can be computed as

$$E_W = \begin{cases} T_W \cdot P_C & \text{if } T_W \leq T_C \\ T_C \cdot P_C + (T_W - T_C) \cdot P_S & \text{if } T_C < T_W \leq (T_C + T_S) \\ T_C \cdot P_C + T_S \cdot P_S \\ \quad +(T_W - T_C - T_S) \cdot P_L & \text{if } (T_C + T_S) < T_W \leq (T_C + T_S + T_L) \\ T_C \cdot P_C + T_S \cdot P_S + T_L \cdot P_L \\ \quad +(T_W - T_C - T_S - T_L) \cdot P_I & \text{if } (T_C + T_S + T_L) < T_W \end{cases} \tag{7.6}$$

Finally, T_Q is the time that goes from the end of T_{RX} to the beginning of a new period, and it can be computed as previously shown in Equation 7.2. At this point, it must be remarked that T_W and T_Q are similar as there are no network operations during these intervals. Hence, the argumentation that leads to Equation 7.6 can also be applied to compute E_Q.

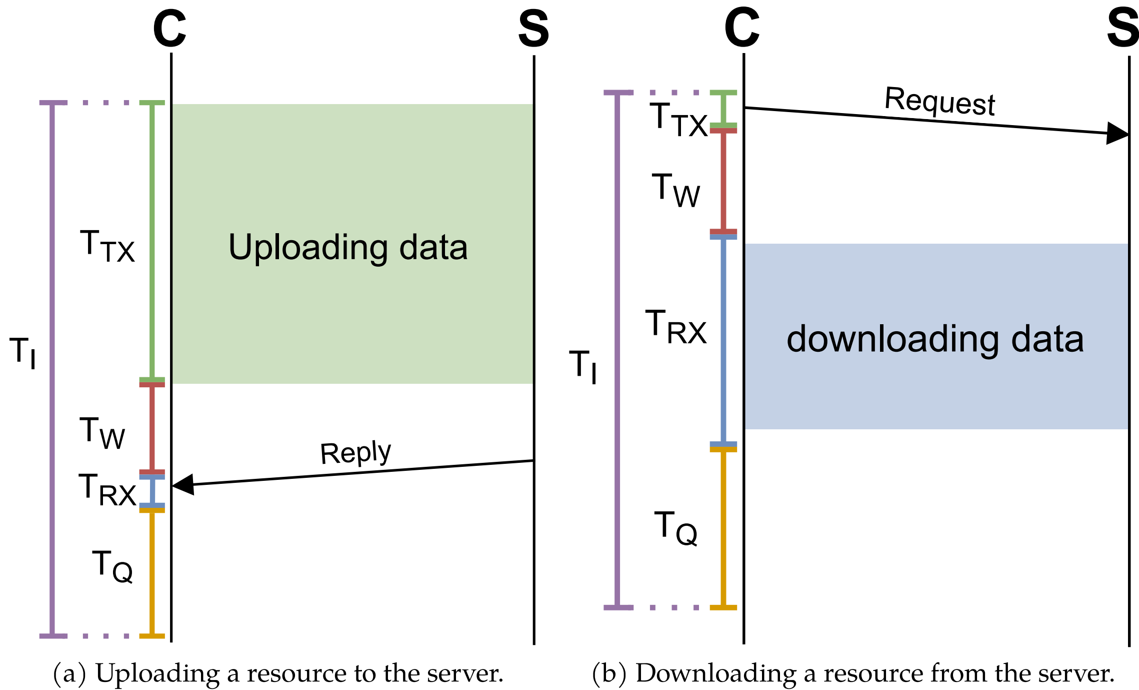
$$E_Q = \begin{cases} T_Q \cdot P_C & \text{if } T_Q \leq T_C \\ T_C \cdot P_C + (T_Q - T_C) \cdot P_S & \text{if } T_C < T_Q \leq (T_C + T_S) \\ T_C \cdot P_C + T_S \cdot P_S \\ \quad +(T_Q - T_C - T_S) \cdot P_L & \text{if } (T_C + T_S) < T_Q \leq (T_C + T_S + T_L) \\ T_C \cdot P_C + T_S \cdot P_S + T_L \cdot P_L \\ \quad +(T_Q - T_C - T_S - T_L) \cdot P_I & \text{if } (T_C + T_S + T_L) < T_Q \end{cases} \tag{7.7}$$

Finally, $E_{PROM_{TX}}$ and $E_{PROM_{RX}}$ are different from 0 only if the interface is into IDLE at the beginning of T_{TX} and T_{RX}.

7.3 Estimating the client-side energy consumption of a connection-oriented application

Modeling a connection-oriented request-response scheme

The energy consumption of a client application that periodically exchanges data with a server using an existing TCP connection has been computed accordingly to

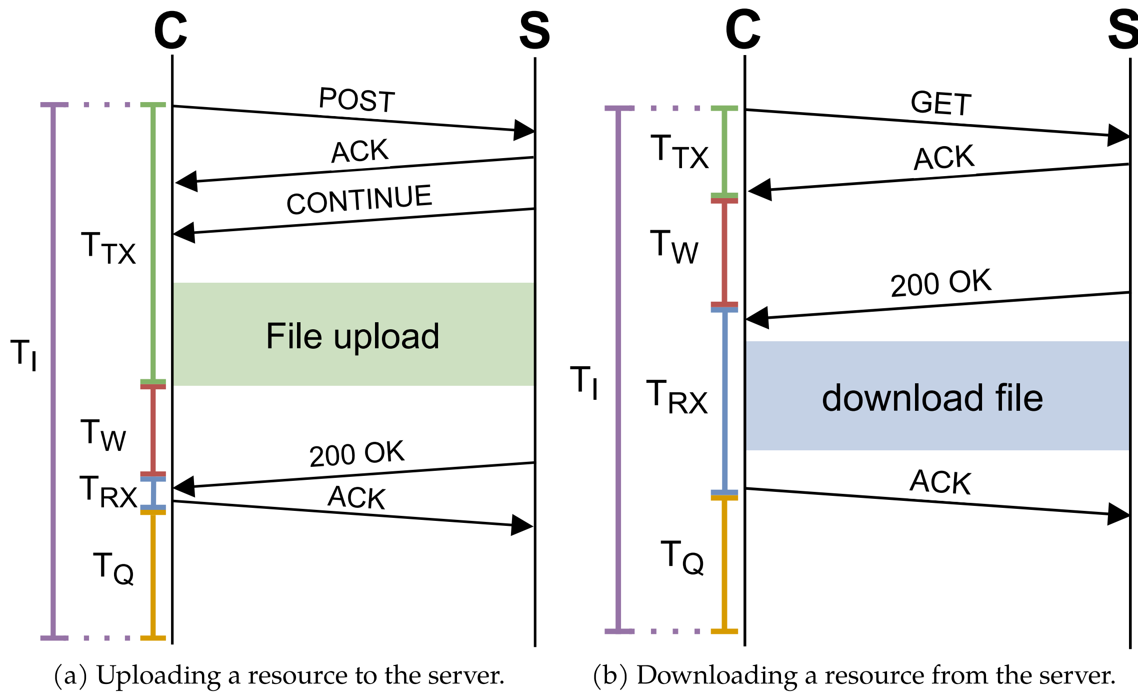


(a) Uploading a resource to the server. (b) Downloading a resource from the server.

Figure 7.4: The interaction between the client and the server, considering a connection-oriented application operating on an application period T_I.

two different scenarios. The first one examines an application that periodically sends a moderately large amount of data to the server, receiving a response of a few bytes. The second scenario considers the opposite situation where the application sends a small request, receiving a larger quantity of bytes. In both scenarios, the two applications operate with a period T_I. These two applications can represent an IoT scenario where some nodes upload collected data receiving a confirmation message (scenario 1), and others periodically ask a server for some data (scenario 2). Figure 7.4 depicts the considered scenarios.

The energy consumption for the two scenarios was evaluated using a hybrid approach based on the integration of the analytical model shown in Section 7.2 and experimental results collected in a realistic setting. This strategy was chosen to deal with the TCP protocol, which is much more complex than UDP. In fact, multiple factors affect the TCP throughput. For instance, the flow control mechanism will limit the throughput if the receiver cannot manage the incoming traffic. In other cases, the congestion control mechanism dominates the TCP throughput at the steady-state. Various congestion control strategies have been shown in the literature (e.g., Reno, Vegas, CUBIC, etc.), and multiple models have been proposed to estimate the maximum achievable TCP throughput (Mathis et al., 1997; Padhye et al., 1998; Cardwell et al., 1998; Bao et al., 2010). However, these models are valid only under rigid assumptions. Moreover, they make use of parameters whose values are not constant



(a) Uploading a resource to the server. (b) Downloading a resource from the server.

Figure 7.5: The TCP-based application used to collect T_{TX}, T_{RX}, and T_W values in a real environment.

over time and can only be collected through network measurements (such as the packet loss probability and the RTT). Consequently, providing a reliable estimation of T_{TX} and T_{RX} is not trivial.

Collecting T_{TX}, T_{RX}, and T_W from a real environment

We gathered T_{TX} and T_{RX} values from a set of experiments based on HTTP requests. Precisely, the first application is accustomed to an HTTP POST request as depicted in Figure 7.5a. For this case, T_{TX} has been computed as the times between the HTTP POST request and the last TCP acknowledgment received from the server. Instead, T_{RX} has been calculated as the time between the HTTP 200 OK message received from the server and its acknowledgment. Conversely, the metrics for the second scenario have been collected considering the HTTP GET request of Figure 7.5b. In this case, T_{TX} is the time that goes from the HTTP GET request to the reception of the acknowledgment for the request packet, while T_{RX} is the time needed to receive the response from the server, considering the download of the resource, the HTTP reply, and the acknowledgment packets. Finally, T_W is computed as the time that goes from the end of T_{TX} and the beginning of T_{RX} for both applications.

The experiments were performed using the setup of Figure 7.6. The client application was hosted on a Raspberry Pi. We chose such a device as it has poorer

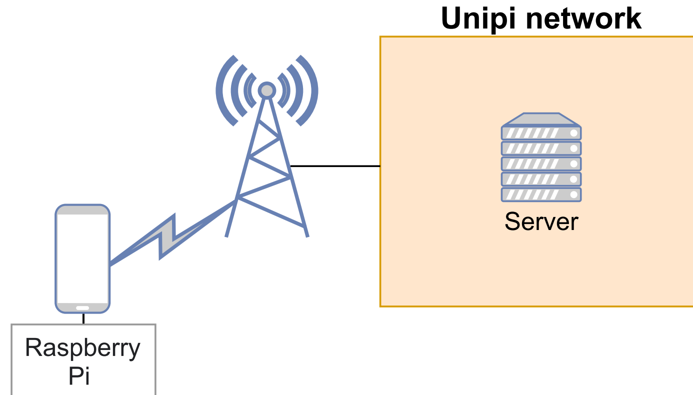


Figure 7.6: The setup used to collect T_{TX}, T_{RX}, and T_W values in a real environment for a connection-oriented application.

computational capabilities with respect to a PC. Consequently, it is more similar to a generic constrained TN. The Raspberry Pi gets the LTE connectivity from an Android smartphone, connected using a USB cable. Instead, an NGINX server was deployed on a machine belonging to the University of Pisa, which is considered the edge server. To mimic a cloud server, 100 milliseconds of delay had been applied on the server machine for both incoming and outgoing packets using tc. In other words, for the cloud configuration, a ΔRTT equal to 200 ms had been used.

```
for i ∈ [1, ..., 10] do
    for f ∈ [2, ..., 12] MB do
        for s ∈ [server_EDGE, server_CLOUD] do
            start the tshark session;
            send an HTTP POST request, uploading a file of size f on server s;
            send an HTTP GET request, downloading a file of size f from
              server s;
            stop the tshark session and store the cap file ;
        end
    end
end
```

Algorithm 3: The setup used to collect T_{TX}, T_{RX}, and T_W values.

At the beginning of each experiment, the client starts a tshark(tshark, 2021) session to sniff the packets between the client and the server. Then, the client application uses curl to send an HTTP POST request to the server, uploading to the edge a file of known size f. Once the first request is completed, the client uses curl to send the GET requests, downloading a file of identical size. Finally, the client stops the packet sniffer and stores the sniffed traffic trace into a cap file. At this point, the procedure is repeated using the cloud server as target. The HTTP requests were re-

peated using files of different sizes (from 2 MB to 12 MB) and performed using up to 10 repetitions. The collection of the experimental data is outlined in the pseudocode of Algorithm 3.

We performed 2 experiments. The first one was conducted during the day and aimed to assess transfer times in a scenario where multiple devices are sharing the radio resources. Instead, the second one was performed overnight to reduce the probabilities of cell congestion interference. In other words, it aimed to assess a low congestion scenario.

To conclude, the cap files had been post-processed to retrieve T_{TX}, T_{RX}, and T_W values. Then, to compute the energy consumption of a TCP-based application, the collected values were used as input parameters for the model presented in Section 7.2.

7.4 The Energy evaluator module

The code used to assess the energy consumption of connectionless and connection-oriented applications can be found on GitHub (The source code of the energy evaluator, 2021).

The EnergyEvaluator and LTEEnergy classes are the main modules within the developed software. Basically, the EnergyEvaluator object is in charge of computing all the components that compose the overall energy consumption and updating the LTEEnergy object, which maintains the interface status. The code was developed in C ++ and it uses the MATLAB Engine API for C ++ (MATLAB Engine API for C++, 2021) for generating the graphs.

7.5 The energy consumption of an ideal connectionless application

In the following, the consumption of an application that uses UDP sockets to exchange data accordingly to the schema of Figure 7.3 was evaluated. E_I values will be computed considering two different scenarios. The former involves an edge server, while the latter includes a cloud server. Then, results will be expressed in terms of the ρ index, which is computed as

$$\rho = \frac{E_I^E}{E_I^C} \tag{7.8}$$

where E_I^E is the energy consumption within a period when the edge server is considered, and E_I^C is the energy required to run a period when the cloud server was involved. Then, ρ values can be used to compare the two configurations. ρ values

smaller than 1 indicate that a lower amount of energy is required when the edge server is used. Conversely, ρ values greater than 1 indicate that the cloud server is the one that minimizes the energy consumption.

The analysis has been performed varying the application period T_I, the elaboration time T_{ELAB}, the RTT toward the cloud (hereafter RTT^C), and the amount of data transferred. Instead, the RTT toward the edge (RTT^E) of 40 ms was adopted, which is the median latency value obtained using the MECPerf-active LTE measurement method when no cross-traffic is injected into the access network (see Figure 5.5). We will define ΔRTT as the difference between the cloud and the edge RTT. In other words, ΔRTT can be computed as

$$\Delta RTT = RTT^C - RTT^E \tag{7.9}$$

Consequently, we can use Equation 7.1 to compute

$$T_W^C = T_W^E + \Delta RTT \tag{7.10}$$

where T_W^E and T_W^C are T_W values computed when the edge and the cloud server are involved, respectively. Then, if we consider Equation 7.2 we can compute

$$T_Q^C = T_Q^E - \Delta RTT \tag{7.11}$$

where T_Q^E and T_Q^C are T_Q values computed when the edge and the cloud server are involved, respectively. Then from Equations 7.10 and 7.11 it can be seen that the cloud configuration is characterized by higher T_W and lower T_Q values. Consequently, when the cloud is involved, higher E_W and lower E_Q will be obtained.

Finally, it should be noted that T_{TX} and T_{RX} are computed starting from the amount of data transferred and the bitrate of the interface. Precisely, T_{TX} has been computed as

$$T_{TX} = \frac{8 \cdot B_{TX}}{bitrate_{uplink}} \tag{7.12}$$

where B_{TX} is the number of bytes sent during T_{TX}, including both the application data and the overhead introduced by the underlying network stack levels. Instead, $bitrate_{uplink}$ is the bitrate of the interface in the uplink direction. It must be noted that Equation 7.12 does not consider the RTT between the client and the server as the UDP protocol lacks of any rate control mechanism. Hence, the time required to send the data does not depend on the RTT. Then, considering Equation 7.4, it can be easily concluded that also E_{TX} is independent from the RTT. To compute T_{TX} and T_{RX} an uplink bitrate $bitrate_{uplink}$ equals to 1 Mbps and a downlink bitrate $bitrate_{downlink}$ equals to 0.8 Mbps were adopted, respectively. These bitrates comply with those offered by a well-known Italian operator (Tim: Le tecnologie abilitanti per l'IoT, 2021). Since low performance characterizes the adopted interface, it can

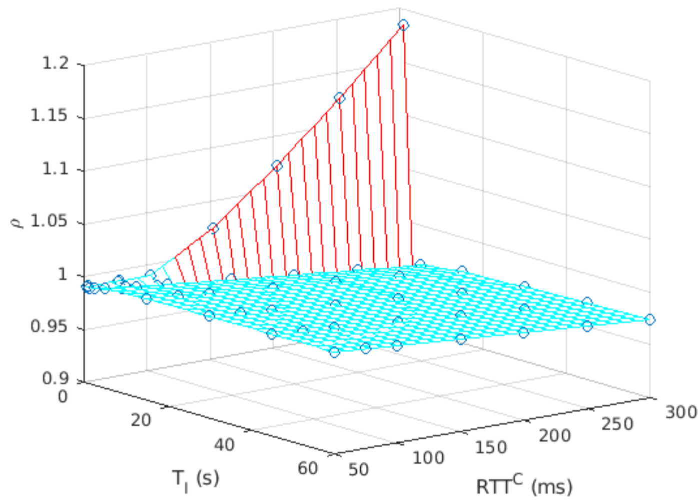


Figure 7.7: The values of ρ obtained setting the amount of transmitted data and the elaboration time, while variable RTT^C and T_I are adopted. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge.

Sent data	16 000 B
Recv data	16 000 B
RTT^E	40ms
RTT^C	from 50ms to 300ms
T_{ELAB}	150ms
T_I	from 750ms to 60s

Table 7.3: The setup used to compute the ρ values of Figure 7.7.

be assumed that the bottleneck of the path between the client and the server will be located on the wireless link that connects the client interface with the LTE BS. Consequently, it is reasonable to assume that incoming packets are received back-to-back during T_{RX}. Then T_{RX} can be considered independent from the RTT, and it can be computed similarly to T_{TX}.

$$T_{RX} = \frac{8 \cdot B_{RX}}{bitrate_{downlink}} \tag{7.13}$$

where B_{RX} is the amount of bytes received within a single period, including the data introduced by the underlying network stack levels.

Evaluating ρ considering different combinations of application periods and RTT^C

First, the energy consumption of both the edge and the cloud configuration has been evaluated considering a scenario where the amount of transmitted data and the elaboration time are constant and the application period (T_I) and the RTT toward the cloud (RTT^C) vary.

Figure 7.7 shows a 3D surface of ρ values, while Table 7.3 summarizes the setup adopted. The red area of the surface identifies those combinations of parameters

that produce ρ values greater than 1. Thus, for the red area, the interaction with the remote server is less energy demanding than the interaction with the edge server. Conversely, the blue area of the surface indicates combinations with ρ values smaller than 1. Hence, interacting with the edge server is the most convenient choice for this area.

As can be seen, when low T_I values and high RTT^C values are considered, the energy needed to interact with the remote server is lower than the energy required to communicate with the edge server. Instead, when T_I is greater than 750ms, the edge server is always the best choice. Note that when the RTT increases E_{TX} and E_{RX} do not change as they depend only on the amount of data transmitted and on the bitrate of the interface. This means that all the differences between the two configurations can be ascribed to E_W and E_Q. Let T_W^E and T_W^C be the T_W computed for the edge and the cloud configurations, respectively. As stated before, Equation 7.1 can be used to compute T_W. With a T_{ELAB} of 150 milliseconds and a RTT^E of 40 milliseconds, we obtain a T_W^E equal to 190 milliseconds. Note that the considered interface has a T_C of 200 milliseconds. Hence, since T_W^E is smaller than T_C, the interface spent its T_W^E entirely into CR. Instead, when the cloud server is considered, the RTT increases. Consequently, T_W^C becomes higher than T_W^E. This means that E_W^E is always smaller than E_W^C. When the cloud server is used, the interface stays in CR only for 10 milliseconds longer, while most of its ΔRTT is consumed in SHORT DRX (and eventually in LONG DRX and in IDLE). In other words, the interface spent most of its ΔRTT in a state characterized by lower power consumption. Instead, T_Q is computed as the time needed to complete the current period. Note that T_I, T_{TX}, and T_{RX} are fixed and T_W^E is always smaller than T_W^C (see Equation 7.10). Consequently, T_Q^E is always greater than T_Q^C (see Equation 7.11). For T_I equal to 750ms, T_Q^C is equal to a few milliseconds. This means that, when the edge configuration is considered, the interface spent most of its ΔRTT into CR with the highest power consumption. Consequently, when the edge server is considered, the additional energy spent in T_Q is higher than the energy saved during T_W. Hence, the E_I^E is greater than E_I^C, and ρ is higher than 1. The values E_W, E_Q, and E_I for the two configurations, computed using T_I equal to 750 and 1 000 milliseconds are summarized in Table 7.4.

Instead, when large T_I values are considered, the edge configuration is always the most convenient. Note that T_W is independent from T_I. Consequently, when a higher T_I is considered E_W^E and E_W^C do not change. Instead, when T_I increases, T_Q increases in both configurations. When T_I is equal to 1 second, T_Q^C is barely sufficient to enter into SHORT DRX. This means that the cloud configuration spends a small portion of its ΔRTT into CR, while the edge configurations spend its ΔRTT entirely into SHORT DRX. Consequently, at the end of the period, the additional energy spent into T_Q^E is lower than the additional energy spent into T_W^C for a cloud configuration. Thus, the edge configuration demonstrates to require lower energy.

T_I (ms)	RTT^C (ms)	E_W^E (mJ)	E_W^C (mJ)	E_Q^E (mJ)	E_Q^C (mJ)	E_I^E (mJ)	E_I^C (mJ)
750	50	190.0	200.0	225.9	222.3	729.5	735.9
750	75	190.0	209.0	225.9	213.3	729.5	735.9
750	100	190.0	218.0	225.9	204.3	729.5	735.9
750	150	190.0	235.9	225.9	162.0	729.5	711.5
750	200	190.0	253.9	225.9	112.0	729.5	679.5
750	250	190.0	271.8	225.9	62.0	729.5	647.4
750	300	190.0	289.8	225.9	12.0	729.5	615.4
1 000	50	190.0	200.0	315.6	312.0	819.2	825.6
1 000	75	190.0	209.0	315.6	303.1	819.2	825.6
1 000	100	190.0	218.0	315.6	294.1	819.2	825.6
1 000	150	190.0	235.9	315.6	276.1	819.2	825.6
1 000	200	190.0	253.9	315.6	258.2	819.2	825.6
1 000	250	190.0	271.8	315.6	240.2	819.2	825.6
1 000	300	190.0	289.8	315.6	222.3	819.2	825.6

Table 7.4: E_W, E_Q, and E_I values for the edge- and the cloud-based configurations considering a connectionless application with T_I equal to 750 and 1 000 milliseconds, RTT^C values ranging from 50 to 300 milliseconds, T_{ELAB} equals to 150 milliseconds, RTT^E equals to 40 milliseconds, and 16 000 bytes for both B_{TX} and B_{RX}.

Evaluating ρ considering different combinations of transmitted data and RTT^C

Figure 7.8 shows the ρ values computed considering different amounts of data transferred. The setup used to collect the results is summarized in Table 7.5. As can be

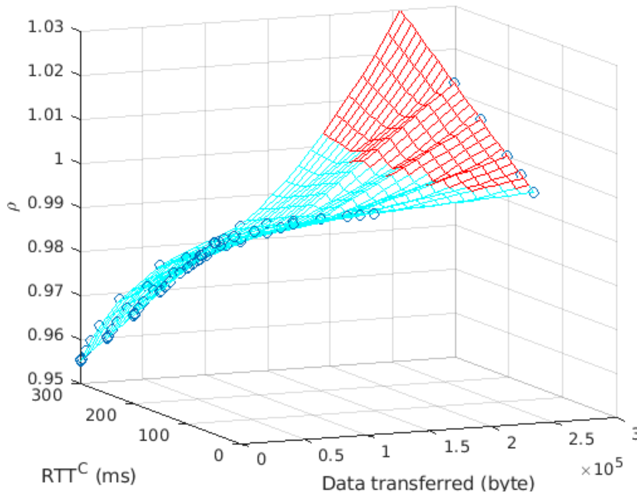


Figure 7.8: The values of ρ obtained setting the application period and the elaboration time, while a variable RTT^C and a variable amount of transmitted data are considered. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge.

Sent data	from 100B to 256 000B
Recv data	from 100B to 256 000B
RTT^E	40ms
RTT^C	from 50 to 300 ms
T_{ELAB}	150ms
T_I	5 000 ms

Table 7.5: The setup used to compute the ρ values of Figure 7.8.

seen, for a low amount of data transferred, the edge configuration demonstrates to be always the one that minimizes the energy consumption. This behavior is similar to the one explained in the previous case. Since elaboration time is equal to 150 ms and the RTT toward the edge is equal to 40 ms, the T_W^E is equal to 190ms. Since T_C is equal to 200 milliseconds, the interface spent a small fraction of its ΔRTT into CR when the cloud server is considered. If the amount of data transmitted is small, then also T_{TX} and T_{RX} are small for both the configurations. Since T_I is equal to 5 seconds, then T_Q is big enough to let the interface go into a state characterized by lower power consumption. This means that the interface spends its ΔRTT entirely out of CR during T_Q^E. Consequently, the edge configuration demonstrates lower consumption at the end of the period.

In addition, for a given amount of data transferred, it can be noted that ρ values decrease as the RTT toward the cloud server increases. This behavior can be explained as follows. When the RTT^C increases, the ΔRTT increases as the RTT toward the edge is a fixed value. This means that the gain obtained during T_W^E and T_Q^C increases. When the cloud server is considered, the interface spends a higher fraction of its ΔRTT into SHORT DRX. Instead, when the edge configuration and a small amount of data are considered, the interface spends this higher ΔRTT into LONG DRX. Note that it was considered an interface characterized by a mean power

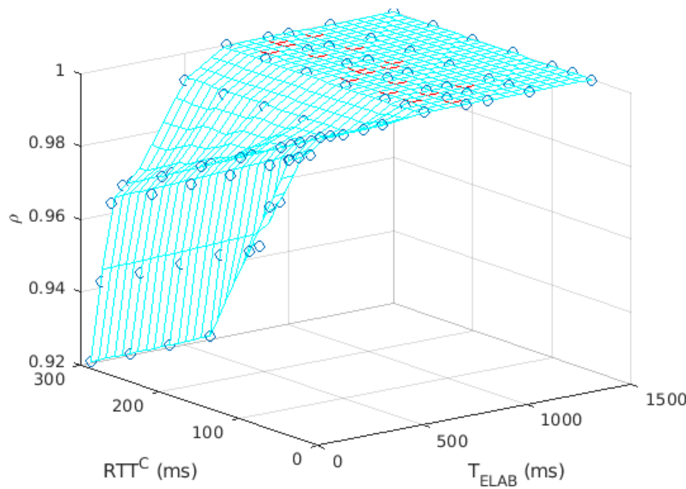


Figure 7.9: The values of ρ obtained setting the application period and the amount of transmitted data, while variable RTT^C and elaboration times are adopted. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge.

Sent data	16 000 B
Recv data	16 000 B
RTT^E	40ms
RTT^C	from 50 to 300 ms
T_{ELAB}	from 0 to 1500 ms
T_I	5 000 ms

Table 7.6: The setup used to compute the ρ values of Figure 7.9.

consumption of 163.23 mW and 359.07 mW when the interface is in LONG DRX and SHORT DRX, respectively. Since LONG DRX has smaller consumption with respect to SHORT DRX, it can be easily understood why ρ values decrease.

Finally, we can see that when a higher amount of data are considered, the cloud configuration becomes the most favorable configuration. For example, when 100 bytes are transmitted and RTT^C is equal to 150 milliseconds a ρ of 0.978 is obtained. Instead, for 256 000 bytes transmitted and a RTT^C of 150 milliseconds ρ is equal to 1.012. In fact, if the amount of data transmitted increases then T_{TX} and T_{RX} increase, while T_Q decreases. At a certain point, T_Q becomes so tiny that the interface spends a relevant part of its ΔRTT into CR when the edge is involved. As a consequence, the edge configuration becomes the worst one.

Evaluating ρ considering different combinations of elaboration time and RTT^C

Figure 7.9 show the dependency of ρ values with different combination of T_{ELAB} and RTT^C, considering the setup of Table 7.6.

First, it can be noted that the edge configuration is always convenient when low elaboration times are considered. This result can be explained as follows. When a low elaboration time is considered, T_W^E is small, and the interface stays in CR for a

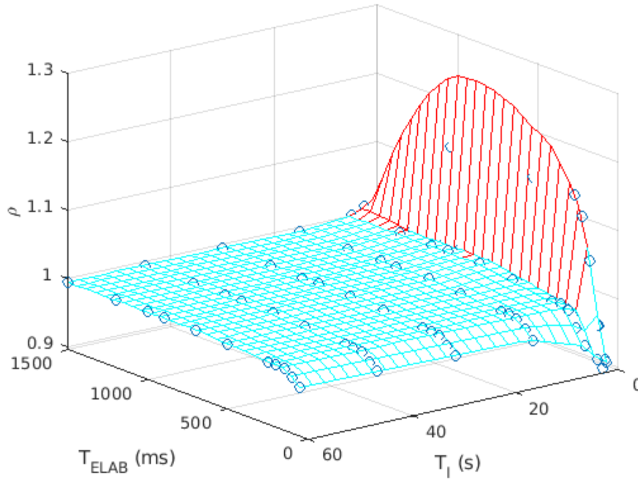


Figure 7.10: The values of ρ obtained setting the amount of transmitted data, while variable elaboration times and application periods are adopted. The red area identifies those configurations producing ρ values greater than 1. For these points, using the cloud is the most convenient choice. Conversely, for the blue area, the most convenient choice is using the edge.

Sent data	16 000 B
Recv data	16 000 B
RTT^E	40ms
RTT^C	170ms
T_{ELAB}	from 0 to 1 500 ms
T_I	from 750ms to 60s

Table 7.7: The setup used to compute the ρ values of Figure 7.10.

limited amount of time. Then, when the cloud configuration is considered, the interface spent most of its ΔRTT into CR, consuming a relevant amount of power. If we consider a T_I of 5 seconds and a transfer of 16 000 bytes, T_Q is sufficient to bring the interface outside of CR. Consequently, the interface spent its ΔRTT in a power-saving mode. This means that more energy is needed to interact with a far cloud server during a single period. When T_{ELAB} increases, T_W^E and T_W^C increase. This means that a smaller fraction of ΔRTT^C is spent in CR. Consequently, the difference between the E_W^C and E_W^E decreases, and ρ increases. Note that, if T_{ELAB} is sufficient to bring the interface LONG DRX for both the two configurations, the additional energy spent during ΔRTT^C and ΔRTT^E is the same. Consequently, ρ becomes equal to 1. In other words, the location of the destination server does not affect the energy needed to run the application.

Evaluating ρ considering different combinations of elaboration time and application period

Finally, the impact of both the elaboration time and the application period has been evaluated, considering the setup of Table 7.7. Figure 7.10 depicts the results. As can be noted, the cloud configuration demonstrated to be the best one for low T_I values. Conversely, the edge configuration brings benefits only when a short elaboration

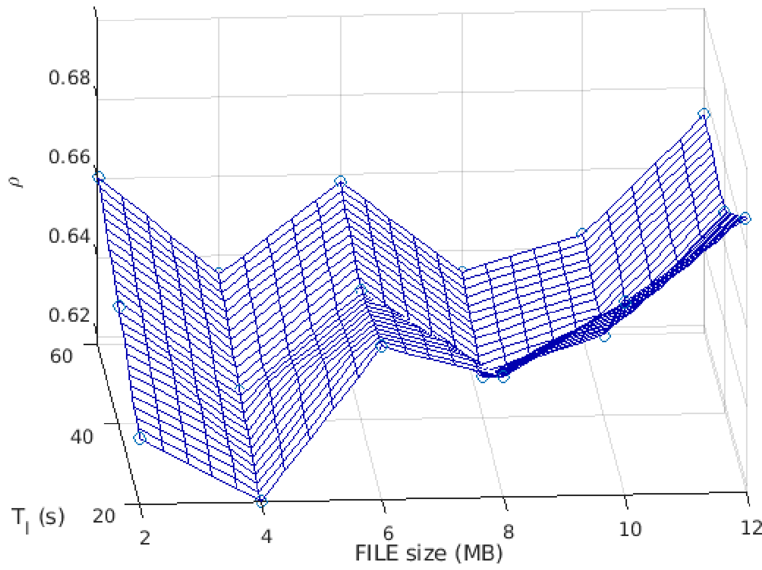
time is involved. In general, for most of the considered combinations, the location of the destination server does not affect the overall energy consumption. The reasons that led to this result have been explained in the previous part of this section. When a small T_I is considered, T_Q^C is small. Then, the interface spends a relevant fraction of the ΔRTT^E into CR, with consequently higher consumption. Instead, when the elaboration time is small, the interface spends most of its ΔRTT^C into CR. Thus, the edge configuration exhibited the best behavior at the end of the period. However, for most of the configuration, ΔRTT^E and ΔRTT^C are both spent into LONG DRX. Hence, the additional energy consumed during ΔRTT^E and ΔRTT^C is the same. Then, E_I^E is equal to E_I^C and ρ is equal to 1.

7.6 The energy consumption of a trace-based connection-oriented application

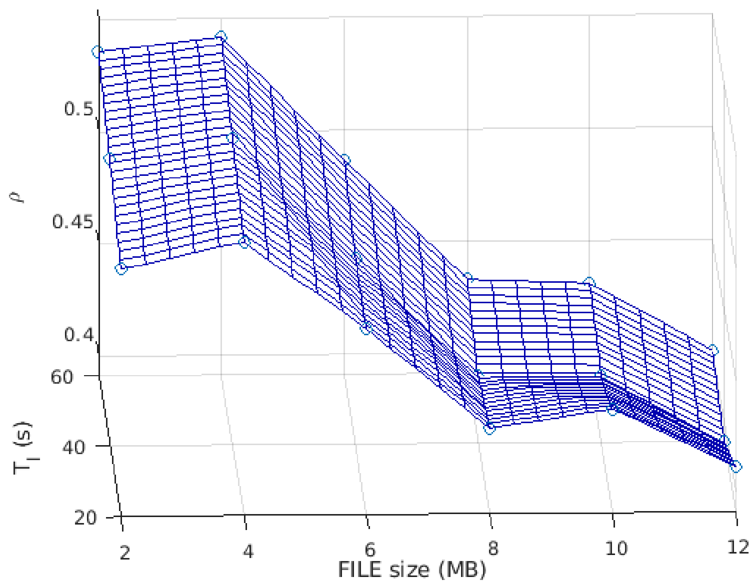
Finally, this section contains the results for the two connection-oriented scenarios. Let N be the number of experimental results collected for each configuration considered. Then, the values of ρ will be calculated as the ratio between the energy required to perform N iterations using the edge-based experimental results and the energy needed to run N iterations using the cloud-based experimental results. In other words, ρ is computed as

$$\rho = \frac{\sum_{j=1}^{N} E_I^E(j)}{\sum_{j=1}^{N} E_I^C(j)} \tag{7.14}$$

where $E_I(j)$ is the energy spent in the j-th iteration. Figure 7.11 shows the ρ values computed considering the experimental T_{TX}, T_W, and T_{RX} values gathered during the night. Moreover, ρ values have been computed considering increasing T_I values and an increasing amount of data. The setup used to collect the ρ values is summarized in Table 7.8. First of all, it can be noted that the edge configuration is always the most convenient one as the ρ values are always smaller than 1. Figure 7.12 confirmed this result. The two plots show the mean energy consumption for the two applications in both the edge- and cloud-based experiments, considering the set of experiments conducted during the night using a period (T_I) equal to 40 seconds. The 99% confidence intervals over the N repetitions are depicted. As can be seen, the mean energy consumption over a period is lower when an edge server is involved. In other words, E_I^E is always smaller than E_I^C. This was an expected result since ρ values are always smaller than 1. Then, we can note that the E_I values computed for the second application (i.e., an application characterized by a consistent download of data) have small confidence intervals in both the edge- and the cloud-based scenarios. Instead, the E_I values computed for the first application (i.e.,



(a) Application #1: uploading some data to the server during the night.



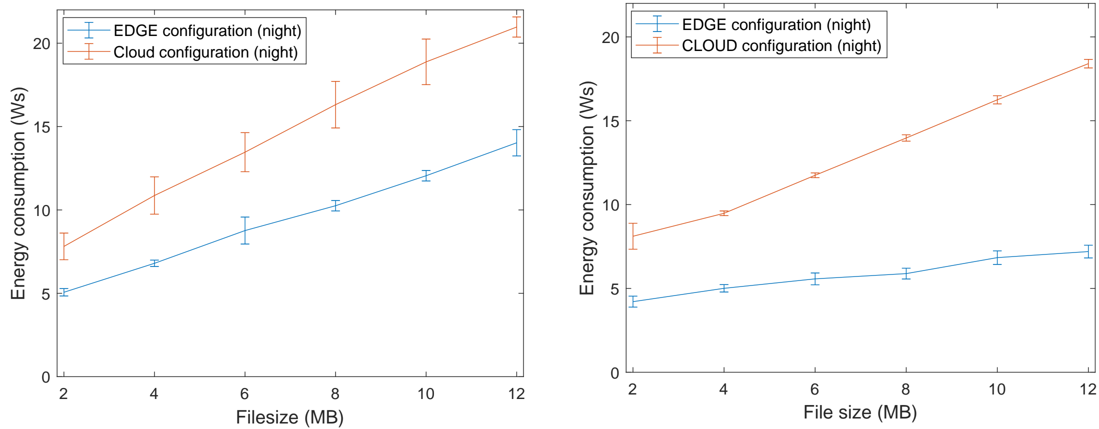
(b) Application #2: downloading some data from the server during the night.

Figure 7.11: The value of ρ collected for a connection-oriented application when an increasing amount of data transferred is considered. The results are based on T_{TX}, T_W, and T_{RX} values collected at night.

RTT^E	$\approx$ 85 ms
RTT^C	$\approx$ 285 ms
T_I	from 20 to 60 s

Table 7.8: The setup used to compute the ρ values of Figure 7.11.

an application based on a consistent upload of data) show higher variability when a cloud server is involved. Anyway, for both applications, the results for the edge and the cloud scenarios are well separated and there are no overlapping confidence intervals.

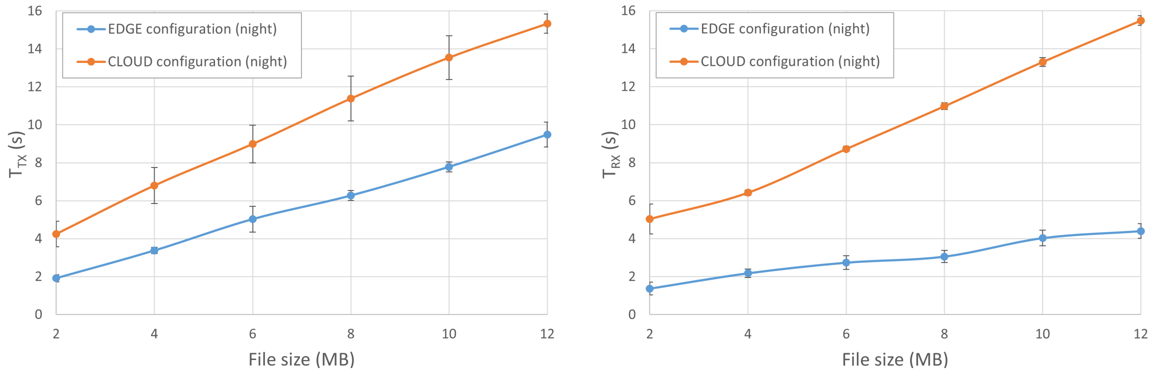


(a) Application #1: uploading some data to the server during the night.

(b) Application #2: downloading some data from the server during the night.

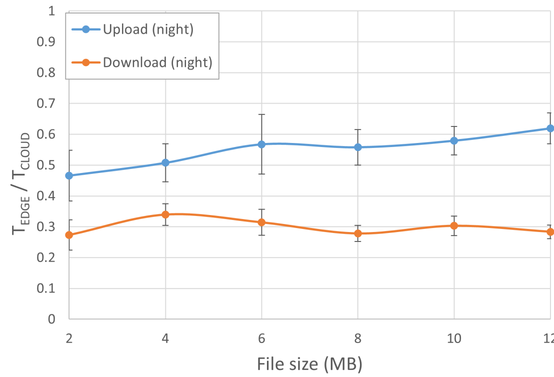
Figure 7.12: The mean energy consumption (E_I) required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds and an application period (T_I) of 40 seconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed at night. The plot shows the 99% of confidence interval of the mean values.

Moreover, it can be pointed out that the ρ values obtained for a connection-oriented application are lower than the ones obtained when a connectionless application is considered. In fact, the ρ values depicted in Figures 7.7, 7.8, 7.9, and 7.10 never go below 0.90, while connection-oriented ρ values stay always below 0.68. This means that, for a connection-oriented application, the edge brings higher benefits. This result can be attributed to the dependency between the TCP throughput and the RTT between the two end-points. In fact, when the edge server is involved, the RTT is smaller, and consequently, the throughput is higher. This means that a smaller amount of time is needed to send or receive a certain amount of data. Consequently, T_{TX}^E and T_{RX}^E are shorter, and the interface stays in CR for a lower amount of time. Note that the mean power consumption in CR is equal to 1200 mW when the interface is transmitting and to 1000 mW when the interface is receiving or idle. Instead, the mean power consumption is equal to 359.07 mW, 163.23 mW, and 14.25 mW when the interface is in SHORT DRX, LONG DRX, and IDLE, respectively. Hence, since CR is the most consuming state, the time spent in T_{TX} and in T_{RX} has a relevant impact on the overall energy consumption. To confirm this result, T_{TX} and T_{RX} values have been plotted in Figure 7.13. As can be seen, T_{TX} and T_{RX} values collected considering an edge configuration are systematically lower than those gathered considering a cloud-based scenario. For example, let us consider Figure 7.13a. When the edge server is involved, T_{TX} values go from 2 to 8 seconds. Instead, when the cloud server is considered, T_{TX} values go from 4 to 16 seconds. The sec-



(a) Application #1: uploading some data to the server.

(b) Application #2: downloading some data from the server.



(c) The ratio between the time needed to transfer a file of a given size to the edge server and to the cloud server.

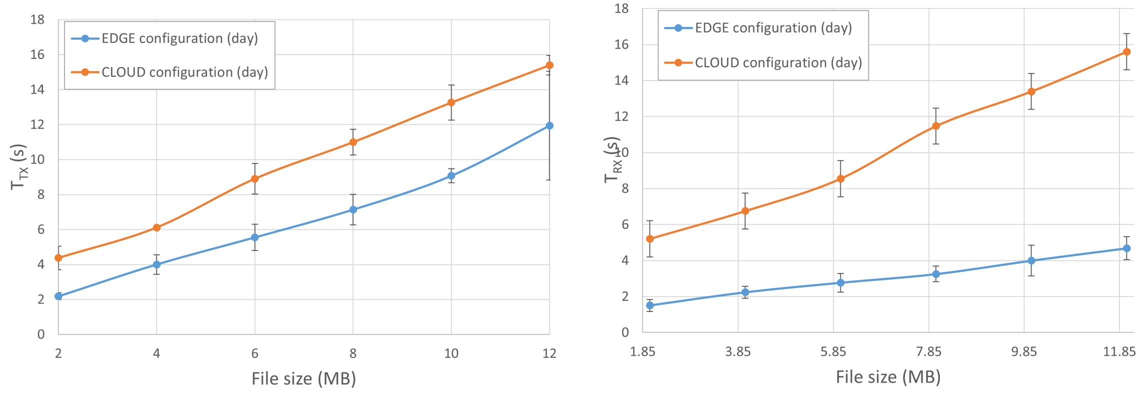
Figure 7.13: The mean time required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed at night. The plot shows the 99% of confidence interval of the mean values.

ond application (Figure 7.13b) shows a similar behavior as T_{RX} edge-based values go from 1 to 4 seconds, while cloud-based values go from 4 to 16 seconds.

Then we can note that ρ values computed for connection-oriented and connectionless applications have a different trend. This is evident if we compare Figure 7.11 and Figure 7.8. In fact, for each RTT^C, the value of ρ computed considering an application based on UDP increases as the amount of data transmitted increases. Instead, the ρ values computed considering the first application remain quite stable, while those computed considering the second application decrease. This difference can be explained as follows. First, the setup used to evaluate the two applications is slightly different. The model based on UDP considers an interface with a poorer bitrate and a smaller amount of data transmissions. Additionally, the model considers the number of bytes transmitted at the interface level. This includes the application-

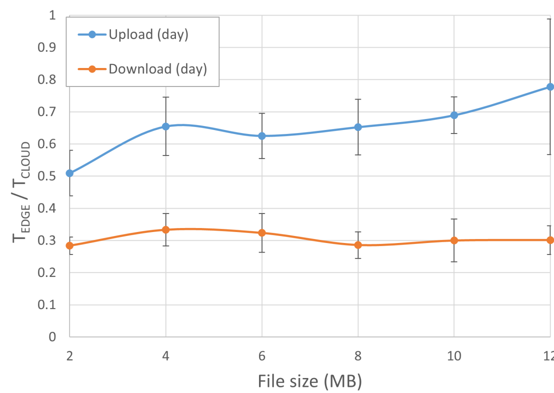
level data and all the headers and trailers introduced by the underlying stack layers. Instead, the data considered for the TCP-based model refers only to the application-level data. Secondly, the values of T_{TX} and T_{RX} retrieved during the experimental phase are based on the HTTP protocol, and they had been collected using existing tools that may introduce additional overhead on both the Raspberry and the server. Finally, the transmission rate of a TCP throughput is adapted to the status of the network by several mechanisms. Instead, UDP does not have such mechanisms. This means that T_{TX} and T_{RX} are ideally independent from the RTT. Consequently, when a connectionless application is involved, T_W and T_Q are the only sources of differences, and the value of ρ is dominated by the minor differences that occur when the interface is in a power-saving mode (i.e., SHORT DRX, LONG DRX, and IDLE). This last factor also explains why the ρ values computed for a UDP connectionless application are almost above 0.90. Instead, the application throughput is strictly related to the RTT when a connection-oriented application is considered and differences in the order of seconds can be observed during T_{TX} and T_{RX}. This means that a consistent amount of energy can be saved by choosing a closer destination server. Note that differences in the energy consumption during T_W and T_Q still arise. However, they are generally negligible when compared with those observed during T_{TX} and T_{RX}.

Finally, Figure 7.14 shows the T_{TX} and the T_{RX} values collected during the day. As can be noted, the ratio between the time needed to upload a file to the edge server and the time needed to upload a file of identical size to the cloud server is higher than those collected overnight (Figure 7.13). In fact, T_{EDGE}/T_{CLOUD} metrics go from 0.5 and to 0.8 during the day, while they go from 0.45 to 0.6 during the night. This behavior can be ascribed to a higher utilization level of the wireless link, which is generated by the higher number of clients connected to the cell during the day. Then, Figure 7.15 shows the ρ values computed using the daytime results. As expected, the uplink results depicted in Figures 7.15a and 7.11a shows a similar behavior. However, ρ values based on daytime metrics are higher. Finally, Figure 7.16 show the mean energy consumption for the edge and the cloud configurations at the 99% of confidence intervals, considering the set of experiments conducted during the day using an application period (T_I) of 40 seconds. The mean energy consumption values collected during the day and those collected during the night (Figure 7.12) show a similar trend. But daytime results show larger confidence intervals. Additionally, they are coherent with both ρ values (Figure 7.15) and both T_{TX} and T_{RX} values (7.14).



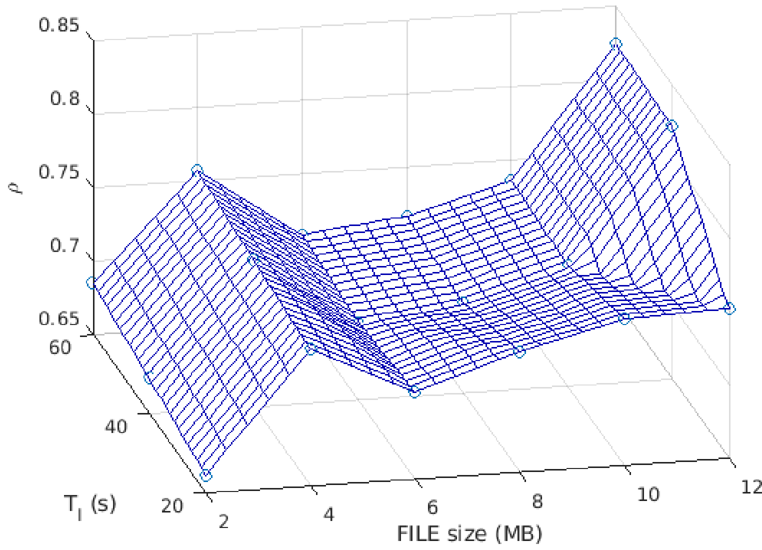
(a) Application #1: uploading some data to the server.

(b) Application #2: downloading some data from the server.

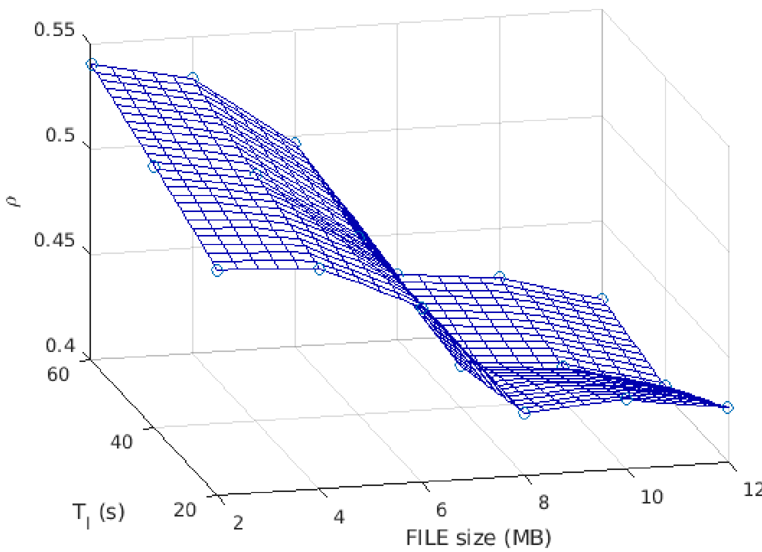


(c) The ratio between the time needed to transfer a file of a given size to the edge server and to the cloud sever.

Figure 7.14: The mean time required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed during the day. The plot shows the 99% of confidence interval of the mean values.



(a) Application #1: uploading some data to the server.

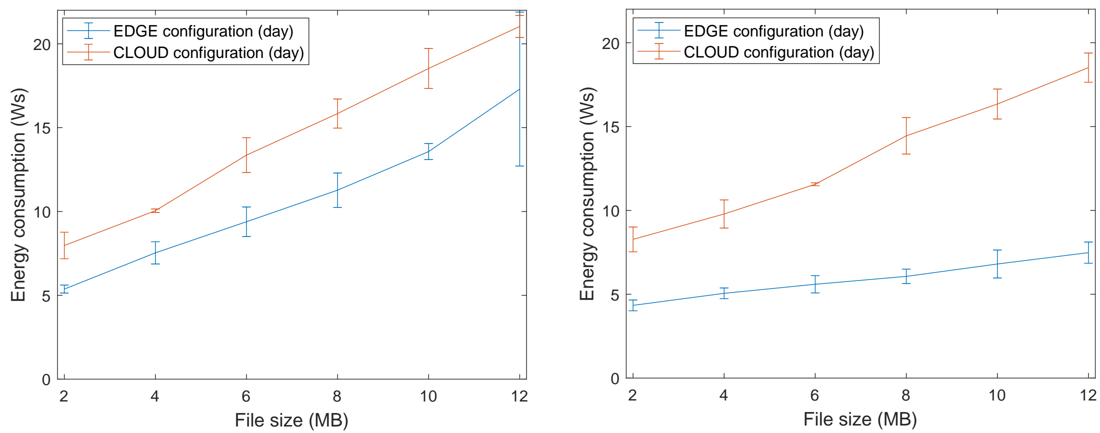


(b) Application #2: downloading some data from the server.

Figure 7.15: The value of ρ collected for a connection-oriented application when an increasing amount of data transferred is considered. The results are based on T_{TX}, T_W, and T_{RX} values collected during the day.

RTT^E	$\approx$ 75 ms
RTT^C	$\approx$ 275 ms
T_I	from 20 to 60 s

Table 7.9: The setup used to compute the ρ values of Figure 7.15.



(a) Application #1: uploading some data to the server during the day.

(b) Application #2: downloading some data from the server during the day.

Figure 7.16: The mean energy consumption (E_I) required to transmit data upon the 10 repetitions, considering a ΔRTT of 200 milliseconds and an application period (T_I) of 40 seconds. The T_{TX} and T_{RX} values were collected during a set of measurements performed during the day. The plot shows the 99% of confidence interval of the mean values.

Chapter 8

Evaluating the path to remote clouds

Finally, this Chapter will propose an approach devised for analyzing the network paths that separate a couple of hosts. A deeper understanding of the entire network path between a TN and the target application server can be used by orchestrators to adopt more complex placement strategies. For example, this knowledge can be used to discourage the deployment of an application on a server that would establish a path that involves links that are already widely used by other applications. Alternatively, this information could be used to stimulate the use of servers that would establish a path via links and ASs that have already demonstrated good performance in other applications.

Traceroute is a widely adopted tool for investigating Internet paths between a couple of hosts. Basically, traceroute sends to a target host IP packets with increasing TTL. Generally, TTL values start from 1, going up to a maximum value (hereafter MAX_DEPTH). When a probe reaches an intermediate hop, an Internet Control Message Protocol (ICMP) Time Exceeded packet is sent to the source node. Instead, when the probe reaches the target, the source node receives a message that depends on the type of the protocol used (i.e., UDP probes generate ICMP Port Unreachable messages, ICMP probes generate ICMP Echo Reply messages, and TCP probes generate TCP RST or SYN + ACK messages). However, firewalls, traffic shapers, and other similar devices placed close to the target can easily detect the traceroute data. Then, these packets can be discarded for security reasons, for reducing the traffic within the destination network, or for other arbitrary reasons. To solve this issue, the following will present a connection-oriented probing methodology. Fundamentally, the proposed method aims at concealing the probes, making them appear as legitimate HTTP traffic and masking the real intentions of the mechanism. The basic functioning mechanism of the proposed probing method is depicted in Figure 8.1.

The rest of this Chapter is organized as follows. First, the connection-based probing method will be presented. Then its performance will be compared with those of the TCP traceroute.

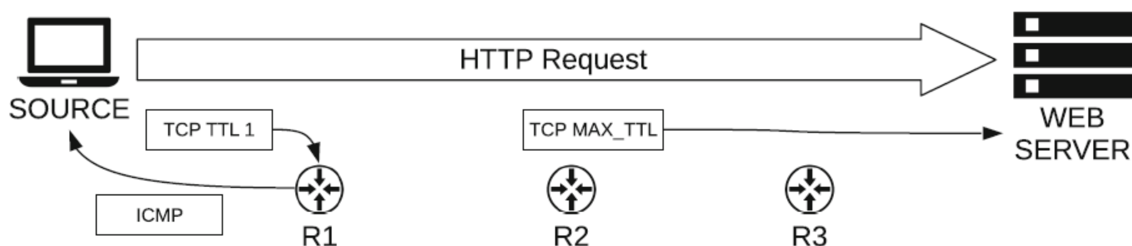


Figure 8.1: The basic operating principles of camotrace. The source node combines probes and legitimate HTTP GET requests, eliciting some ICMP Time Exceeded messages from the intermediate routers.

8.1 The camouflage traceroute software modules

Two different kinds of probes are used to discover the intermediate hops along a path. The first one is based on data segments, while the latter uses acknowledgment packets. In the following, we will refer to these two probing methods as DATA- and ACK-based methods, respectively. The source code of camotrace can be found as open-source on bitbucket (The source code of camotrace, 2021).

The Data segment-based camouflage traceroute implementation

The DATA-based method uses Hypertext Transfer Protocol (HTTP) GET requests as probes. Therefore, the target must be listening for GET requests to function properly.

As already stated, camotrace operates in two phases. The first one is the connection phase, used to establish a TCP connection with the target host. Camotrace uses the raw stream socket interface offered by the OS to make the connection handling as easy as possible. Consequently, camotrace acts on packets only at the application level for both outgoing and incoming traffic. Instead, the second phase is the probing phase, summarized in the pseudo-code of Algorithm 4. For each value of x between 1 and MAX_DEPTH, camotrace operates as follows. First, at Line 7, camotrace sets to x the TTL value associated with the socket between the sender and the target. Then, the socket is used to send the following HTTP request:

"GET / HTTP/1.1\r\nHost: <target host name>\r\nConnection: Keep-Alive\r\n\r\n"

At Line 11, camotrace starts to consume the incoming traffic. If the path between the sender and the target is longer than x hops, an ICMP time exceeded message may[1] be generated at the x-th hop. When camotrace receives an ICMP message, it associates the IP address of the x-th router with the TTL value that elicited the ICMP message. Then it immediately starts to probe the next hop. If the ICMP packet did not arrive

[1] For several reasons, some ICMP packets may not arrive (e.g., the intermediate router is configured to send no ICMP messages).

```
1  MAX_TTL ← System default TTL
2  MAX_DEPTH ← 40
3  MAX_ATTEMPT ← 3
4
5  nAttempt = 0
6  for all x ∈ {1..MAX_DEPTH} do
7  |   setTTL(x)
8  |   send an HTTP request to the target host
9  |   start timer
10 |   setTTL(MAX_TTL)
11 |   while True do
12 |   |   try:
13 |   |   |   listen for incoming traffic
14 |   |   |   if ICMP Time Exceeded packets arrives then
15 |   |   |   |   store the IP address of the x-th host
16 |   |   |   |   nAttempt = 0
17 |   |   |   |   break
18 |   |   |   end
19 |   |   catch timer expired
20 |   |   |   break
21 |   |   end
22 |   |   if the server closes the socket then
23 |   |   |   establish a new connection with the server
24 |   |   end
25 |   |   if ICMP Time Exc. not received && nAttempt < N_ATTEMPT_MAX
   |   |    then
26 |   |   |   x = x - 1
27 |   |   |   nAttempt = nAttempt + 1
28 |   |   end
29 |   |   if ICMP Time Exc. not received && nAttempt ≥ N_ATTEMPT_MAX
   |   |    then
30 |   |   |   nAttempt = 0
31 |   |   end
32 |   end
33 end
```

Algorithm 4: Data-based Camouflage traceroute probing phase.

in time, the timer associated with the socket expires (Line 19), and a new probe with TTL equal to x was sent. Camotrace sends up to MAX_ATTEMPT probes to discover the x-th hop. Then it starts to probe the next hop. Note that the TTL associated with the socket is restored to its original value immediately after sending the request (Line 10). This allows interleaving probes and requests. In fact, the source will not receive any acknowledgment for the probe. Consequently, at a certain point, the message will be retransmitted using the default TTL. Finally, it should be noted that camotrace cannot elicit any ICMP packet on the target machine. This means that it is not able to detect the target host. Therefore, the algorithm always continues until MAX_DEPTH is reached in its current implementation.

Managing unexpected connection closures

To function correctly, camotrace must have full control of the source machine. However, no assumptions can be made on the target machine that is owned by a third-party organization. This means that the target server can make arbitrary and unpredictable decisions that need to be managed. A common problem is represented by the closure of the connection by the server. To deal with the anomalous closure of the connection between the two hosts, the following two improvements have been introduced.

Managing non-persistent connections
As declared in the Request for Comments (RFC) 2616 (Nielsen et al., 1999) an HTTP 1.1 connection should be persistent. Using a limited number of TCP connections brings several advantages. For example, using fewer connections allows saving resources on the hosts, while reducing the number of TCP handshakes reduces the number of packets transmitted and the communication latency. However, a server may still close the connection immediately after completing to serve a request. Camotrace can use probes with two different payload types to manage anomalous connection closures. The former consists of a full HTTP request as previously described in Section 8.1. Instead, the latter uses only one byte of the original request as payloads. Consequently, since the request was not fully received, the server is dissuaded from closing the connection.

Managing other unexpected connection closures
Although the request has not been received in its entirety, the server can still decide to close the connection with the client. This behavior can be due to multiple reasons. For example, the connection could be closed because the server is subjected to a high workload, or server-side timers could be triggered since too much time has been elapsed between two consecutive requests. To manage these problems, camo-

trace constantly checks the status of the connection (Line 22), establishing a new connection when required.

The ACK-based camouflage traceroute

Similarly to the data segment-based version of camotrace, this version of camotrace operates using an already established connection. However, this method is based on ACK probes.

The ACK-based camotrace comprises two software modules residing in the kernel and the user space. The kernel module was developed employing the Linux Netfilter library (netfilter project home page, 2021), which provides hooks that can be used to intercept packets at different levels of the network stack. The kernel-space module is implemented as an FSM with 6 states: INITIALIZING, DISCONNECTED, CONNECTING, SEND_MODIFIED, SEND_UNMODIFIED, and CLOSING. The kernel module behaves as a filter, intercepting the packets belonging to the connection between the user code and the target web server. Then, it manipulates a part of the acknowledgment packets. The TTL applied to the forged ACK is chosen dynamically, depending on the information received from the user code during the initialization, the internal state of FSM, and the information retrieved from the stack. Instead, the user-space module always initiates the discovery procedure, and it is in charge of consuming the ICMP time exceeded messages elicited by the forged ACK packets.

Precisely, for each target, the two modules operate as follows. In the beginning, the user-space module opens a Netlink socket (netlink, 7) to communicate with the kernel filter. Then, the user-space module sends three parameters to the kernel. The first parameter is the IP address of the target host ($TARGET_IP$), the second parameter is the maximum number of hops that can be probed (TTL_MAX), and the third parameter is the maximum number of attempts that can be performed at each hop ($N_ATTEMPT_MAX$). The kernel module starts into INITIALIZING, waiting to receive the parameters from the user module. Once the parameters have been received, the $START$ command is sent to the user space. Then the module goes into DISCONNECTED. Upon receiving the $START$ command, the user code opens a TCP connection with the target web server using a raw socket. Consequently, the filter goes first into CONNECTING and, when the three-ways handshake is completed, in SEND_UNMODIFIED. At this point, the user-side code starts to send HTTP requests, while the filter begins to intercept the outgoing acknowledgments. If the filter is into SEND_UNMODIFIED, the ACK packet is sent without any modification, and the filter goes into SEND_MODIFIED. Otherwise, the filter changes the TTL of the acknowledgment packet, computes a new IP header checksum for the modified ACK, sends to the user module some information about the forged packet, and finally goes back to SEND_UNMODIFIED. In other words, the filter

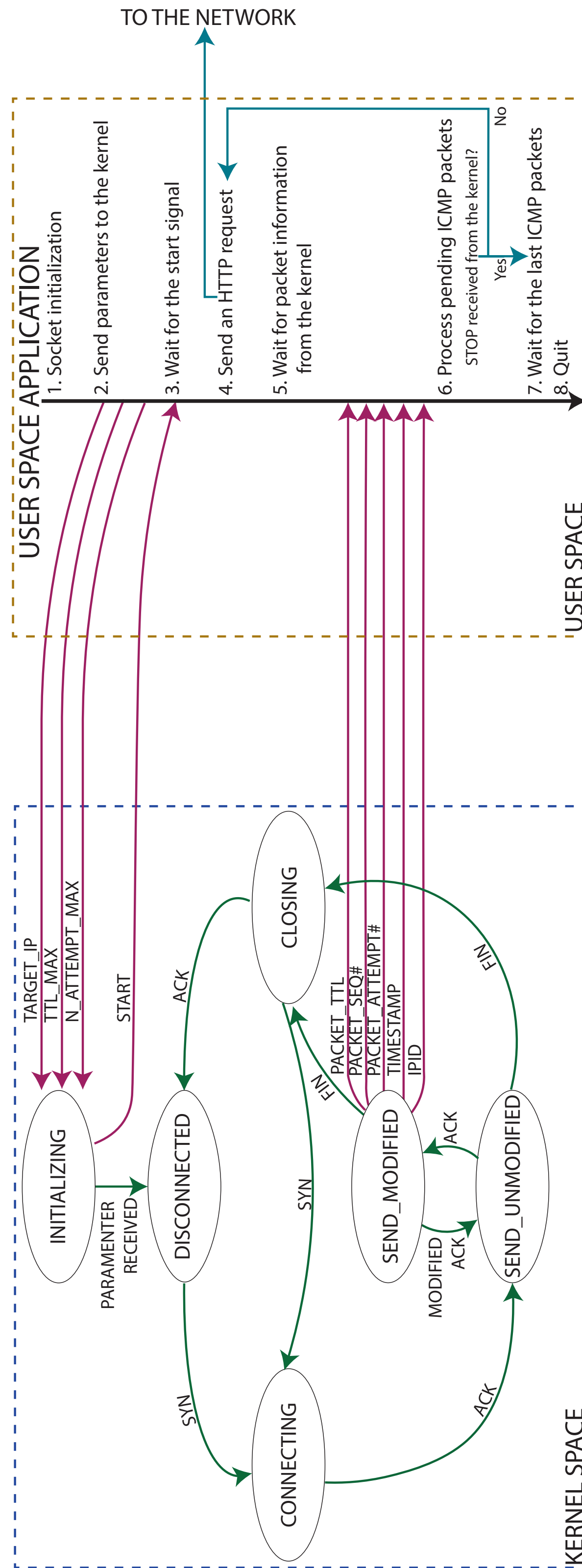


Figure 8.2: The software components of the ACK-based version of camotrace and their interactions.

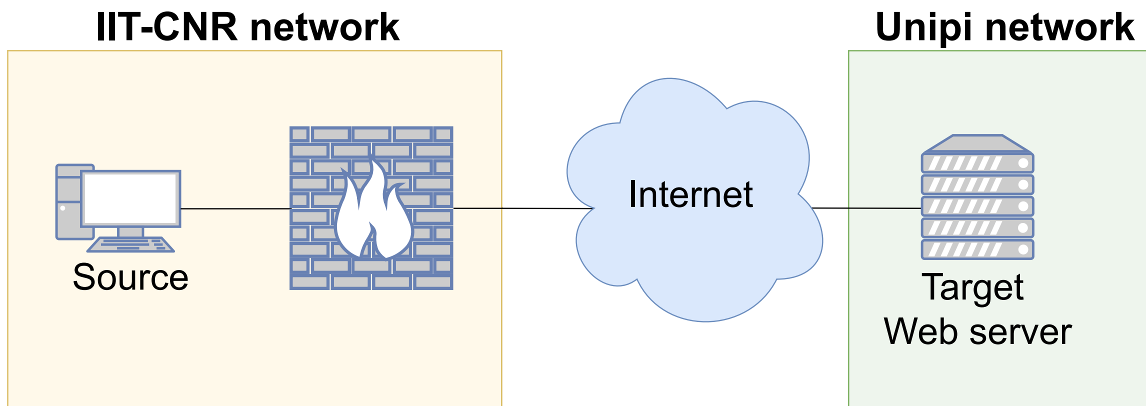


Figure 8.3: The setup used to validate the data segment-based version of camotrace.

starts to alternate transitions between the SEND_UNMODIFIED and SEND_MODIFIED states. Consequently, manipulated and regular ACKs are alternated as well. Note that the modified ACKs are likely discarded before reaching the target, eliciting ICMP packets. While the unmodified ACKs are required to maintain the TCP connection alive. The two software components and their interactions have been depicted in Figure 8.2.

8.2 The camouflage traceroute discovery capabilities

Validation

The data segment-based version of camotrace has been validated twice. The first validation test aimed to assess whether camotrace is able to discover the path between two nodes correctly. Instead, the second validation test was aimed to evaluate its ability to bypass a firewall configured to identify and filter traceroute traffic.

The first validation test involved a source node, hosted at the University of Pisa, and 17 target Web sites belonging to different Italian universities. First of all, we checked that all the target Web servers were hosted within their University networks. This step guarantees that all the hosts involved in the validation are part of the GARR network (Consortium GARR Home Page, 2021), which connects all Italian universities and provides a publicly accessible map (The map of the GARR network, 2021) of its network. Therefore, since the paths between the hosts are well known, the correctness of the results obtained by camotrace can be easily checked. As a result, all 17 paths were correctly discovered at the end of the validation.

For the second validation test, a machine belonging to the IIT-CNR network was used as source, while the target Web server was hosted on a machine located at the University of Pisa. Additionally, the IIT-CNR network hosted a Palo Alto firewall (Palo Alto Networks, 2021). This firewall can identify traffic belonging to spe-

cific applications through deep packet inspection techniques. Then, depending on the configured policies, a packet can be forwarded, shaped, or blocked. In detail, the firewall is configured to block all the traceroute traffic between the source and the target machine. Then we ran traceroute, using UDP, ICMP, TCP probes, and camotrace between the two hosts. To make a fair comparison, the TCP version of traceroute was launched using the destination port 80 since it is the same port used by camotrace. As expected, the firewall blocked all the traceroute traffic. Instead, camotrace succeeded in discovering all the hosts between the source and the target. In other words, camotrace could bypass a Palo Alto firewall configured to block traceroute. The setup used to conduct this second validation experiment is shown in Figure 8.3.

Experimental setup

We perform an experimental campaign aimed to compare traceroute (version 2.1.0 for Ubuntu) with both the DATA- and the ACK-based version of camotrace. As source, a host belonging to the University of Pisa was used. This choice makes possible to simplify the setup of the experiment as much as possible while continuing to provide reliable results[2]. Instead, the targets were selected as follows. First, approximately one million domains from the .it Top-Level Domain (TLD) were collected. This list was resolved in approximately 800 000 IPv4 addresses while the remaining 200 000 names, registered but not associated with any IP address, were discarded. Then, the duplicate addresses were removed, obtaining a new set of about 92 000 addresses. The fact that most of the IP addresses were duplicate is not surprising. In fact, most of the domains are hosted on cloud or hosting services, which generally execute multiple websites onto the same physical machine. Note that traceroute and camotrace are time-consuming applications. Therefore, the set of targets was reduced as much as possible so that experiments could be completed in a reasonable amount of time. To this purpose, each IP address was associated with its Autonomous System (AS) using the Team Cymru Whois service (Team Cymru, 2021). Then, for each AS in the list, only a single IP address was randomly selected. At the end of this step, we obtained a set of approximately 3 260 targets Web servers, each one hosted on different machines belonging to a different ASes. Note that selecting one IP for each AS significantly reduces the list of targets while the heterogeneity of the targets was preserved since it is reasonable to assume that policies are homogeneous within a single AS. Finally, for 629 targets, camotrace was unable to establish a Transmission Control Protocol (TCP) connection. Multiple reasons can cause this: there may not be a Web server running on the target machine, the Web

[2]Results presented in a similar study that not included connection-based mechanisms(Luckie et al., 2008a) showed that the vantage point does not significantly affect the results.

server may be configured not to reply to GET requests, or a firewall may block all the traffic between the source and the target. For 89.5% of these 629 targets, the TCP traceroute could not discover the target interface. However, traceroute collected information about the intermediate nodes even if the target was not discovered. In any case, the experiments were carried out in the remaining set of 2 323 targets for which a TCP connection can be successfully established.

The experiments were conducted using the TCP traceroute (hereafter SYN-based method) and both the DATA- and ACK-based camotrace. To compare all the methods fairly, we configured traceroute as follows. Firstly, the traceroute can send only one probe at a time since the DATA-based camotrace cannot send multiple probes simultaneously. Secondly, the TCP traceroute has been configured to send probes using destination port 80, the same as that used by the HTTP protocol. Generally, traceroute has a default MAX_DEPTH of 30 hops. However, preliminary tests based on traceroute showed that some paths are longer than 30 hops. For this reason, a MAX_DEPTH equal to 40 hops was configured for all the considered methods. Finally, since camotrace cannot detect the target, we excluded the destination from the traceroute results.

Experimental results

In the following, the results obtained during the experiment campaign are presented. First, to contextualize the set of targets used, the geographical location of each target was analyzed. Then the discovery capabilities of camotrace were analyzed.

Evaluate the location of the targets

First of all, the MaxMind GeoLite2 Database(Maxmind, 2021) was used to geolocate the targets. Although the target selection has been limited to Italian domains, the targets can be located anywhere in the world. Then, the results obtained show that the target machines are spread across 70 different countries. This means that some of the targets are located outside the European Union. Additionally, for each target, the distance from the source node was calculated using a delay-to-distance conversion factor of $\sim$72 km/ms (Candela et al., 2019). The bar chart of Figure 8.4 shows the ranking of the top 20 countries that hosts the larger number of targets, while the box plot shows the estimated distance between the source and the targets in the corresponding country. It can be seen that the top 3 countries hosted a similar number of targets, and only a tiny fraction of the targets were hosted in Italy. Moreover, the second country for target density is a non-European country.

At this point, to better characterize the targets, the DATA-, the ACK-, and the SYN-based methods were used to compute the empirical Cumulative Distribution

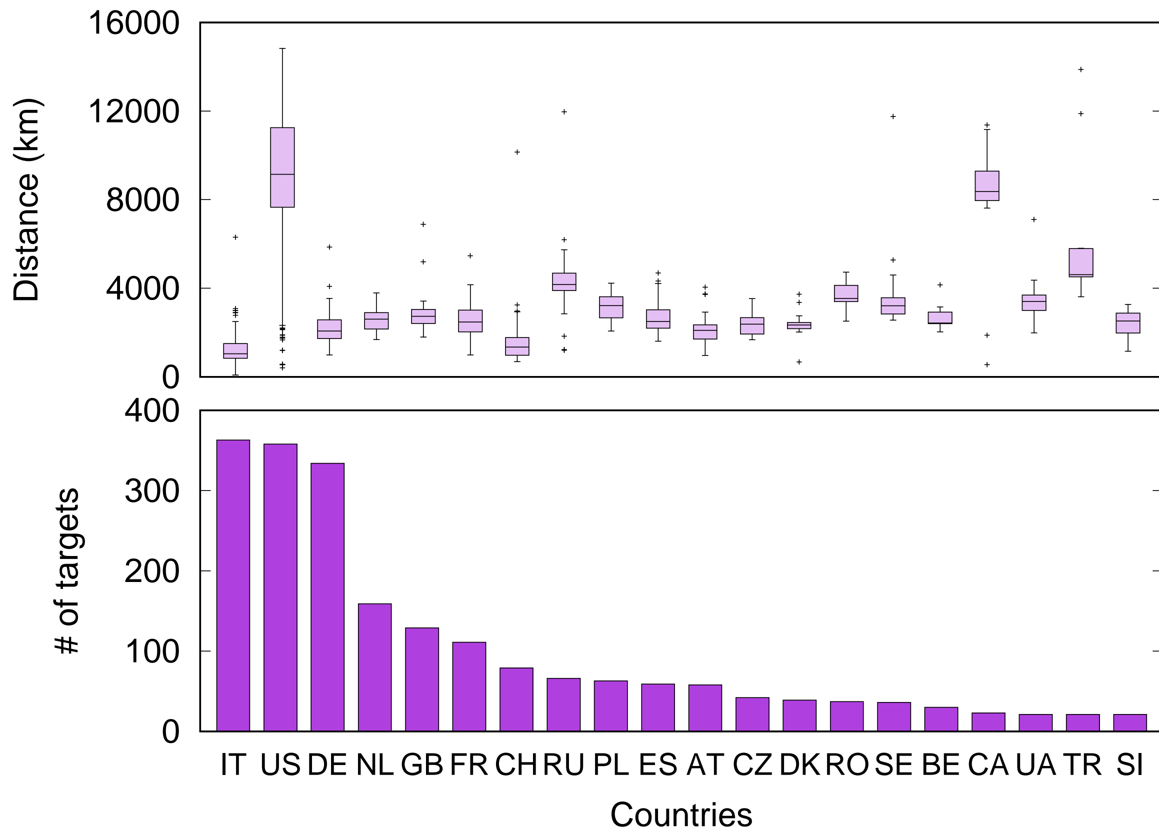


Figure 8.4: The 20 states where the majority of the targets are hosted and the distance in kilometers between the source and the target in the corresponding country.

Function (eCDF) of the AS path length and the eCDF of the number of interfaces per AS. The results are shown in Figures 8.5 and 8.6, respectively. First, we can see that the three methods show similar distributions, and a significant portion of the targets have short-length paths. For example, approximately 40% of the targets see at most 3 ASes. This result is coherent with Figure 8.4 since most of the targets are located in a European country. However, the number of long paths is not negligible and can be reasonably attributed to targets located in other continents. Finally, only one interface was identified for about half of the ASes considered. In general, almost all the ASes contain less than 100 IP interfaces, while only a minority part of them contains a relevant amount of interfaces. In particular, Cogent and Telia contain approximately ~800 and ~400 interfaces, respectively. However, this result is not surprising since these two ASes are the two upstream providers of the GARR, which contain the source hosts.

Compare the discovery capabilities of a connection-based method

At first, we analyzed the discovery capabilities of camotrace in terms of unique IP interfaces discovered. An UpSet plot(Lex et al., 2014), a visualization technique for

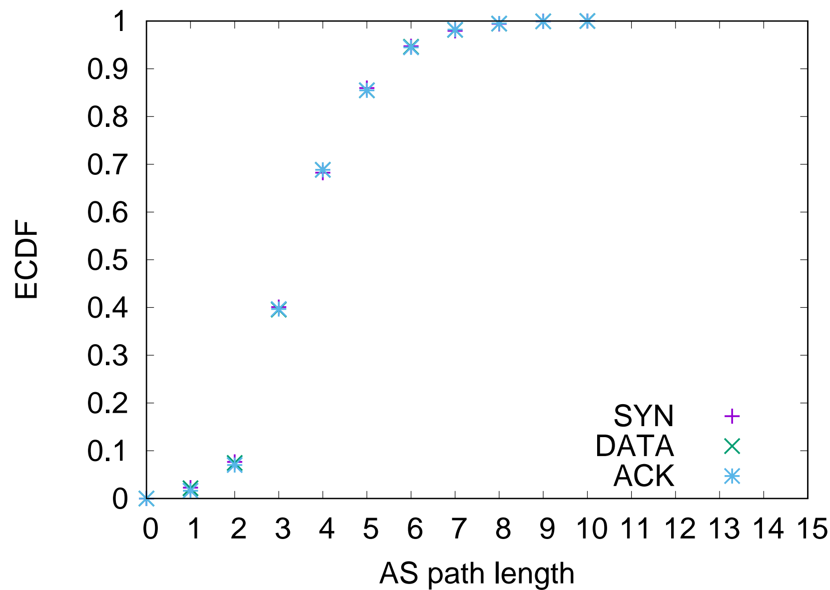


Figure 8.5: The eCDF of the AS path length using the data segment-based camotrace, the ACK-based camotrace, and the TCP traceroute probing methods.

analyzing sets and their intersections, was used to show this information. As can be seen in Figure 8.7a, the intersection between the methods includes 7 113 distinct IP interfaces. In other words, most of the interfaces are discovered by all the methods. The first three rows represent the set of unique IP interfaces discovered by only one method. Thus, we can appreciate that the SYN-based method can discover a higher number of interfaces. In fact, using SYN-based probes, 101 distinct IP interfaces were discovered, while only 33 and 51 interfaces were discovered using DATA- and ACK-based probes, respectively. However, it can be noted that the union of the set of interfaces discovered by the two connection-based methods ($(DATA \cup ACK) - SYN$) found 178 interfaces, while the SYN-based method only 101. Precisely, the 178 interfaces are divided as follows: 33 were found only by the DATA-based method, 51 were found by the ACK-based method alone, and 94 were found by both the DATA-based and the ACK-based method ($(DATA \cap ACK) - SYN$). This indicates that a connection-based method can discover a relevant amount of IP interfaces that traditional probe methods cannot identify. However, it is better to use multiple probing methods to maximize the number of hops discovered.

The IP addresses have been converted into their AS numbers to investigate this result better. Hence, the discovery capability of camotrace was assessed in terms of both AS numbers and AS links. The UpSet plots of these results are shown in Figures 8.7b and 8.7c, respectively. As can we see, these results are coherent with those of Figure 8.7a. In fact, the SYN-based method found more ASes and more AS links than the connection-based methods. Precisely, the SYN-based methods discovered 21 unique ASes, while the DATA- and the ACK-based methods found 2 ASes each.

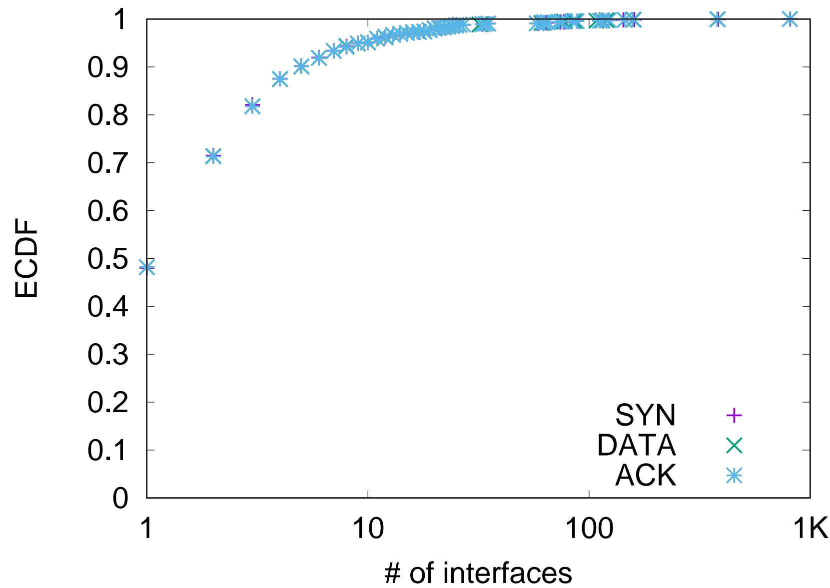


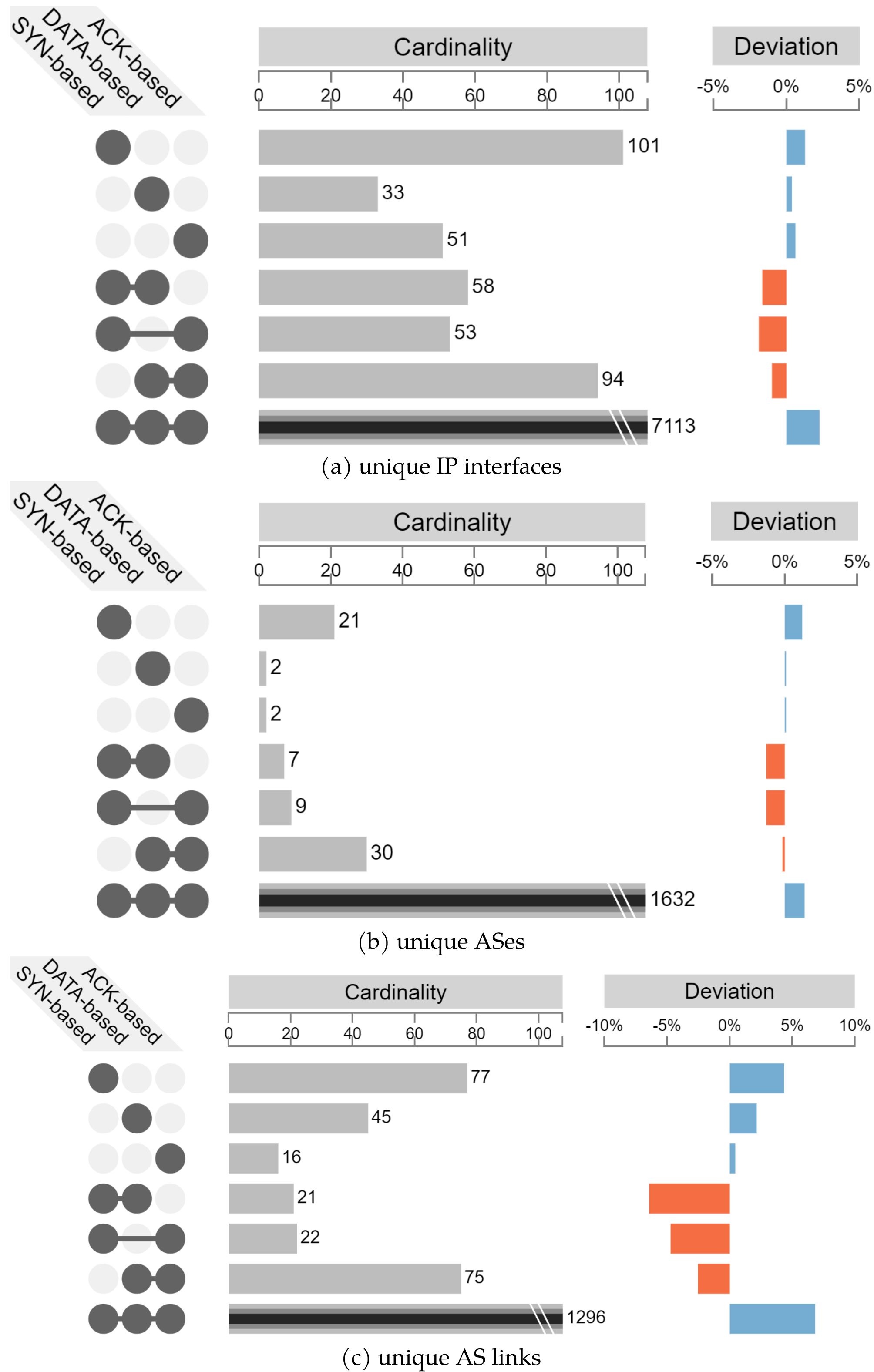
Figure 8.6: The eCDF of the number of interfaces per AS using the data segment-based camotrace, the ACK-based camotrace, and the TCP traceroute probing methods

Instead, 77 AS links were uniquely discovered by the SYN-based method, while the DATA- and the ACK-based methods have found only 45 and 16 links. However, if we considered the intersection between the two connection-based methods ($(DATA \cap ACK) - SYN$), it is possible to note that a relevant amount of additional information was obtained. In fact, the $(DATA \cap ACK) - SYN$ set contains 30 unique ASes and 75 AS links.

Finally, the right side of the UpSet plot shows the deviation of both the sets and their intersections. It is possible to notice that exclusive sets have positive variations, while intersections between couples of methods have negative ones. This indicates that the sets are more separated than expected. In other words, each method appears to be slightly more effective than the other two on a subset of targets.

Then, the discovery capabilities of camotrace were evaluated in terms of discovered paths. First, we analyzed the number of paths for which differences emerged, considering the DATA- and the SYN-based methods. Given methods M and N, we say that method M detected an *additional hop* if ICMP messages from the i-th hop were received only for probes belonging to method M. We additionally defined I_{DATA} and I_{SYN} as the set of intermediate hops discovered in a path for the DATA-based version of camotrace and the TCP traceroute, respectively. Then the following four cases were identified:

- **Case 1**: set of paths for which only the DATA-based method has identified at least an additional hop($I_{DATA} \supset I_{SYN}$).



(a) unique IP interfaces

(b) unique ASes

(c) unique AS links

Figure 8.7: The upset plots of the number of unique IP interfaces, unique AS numbers, unique AS links found by the SYN-, the DATA-, and the ACK-based methods.

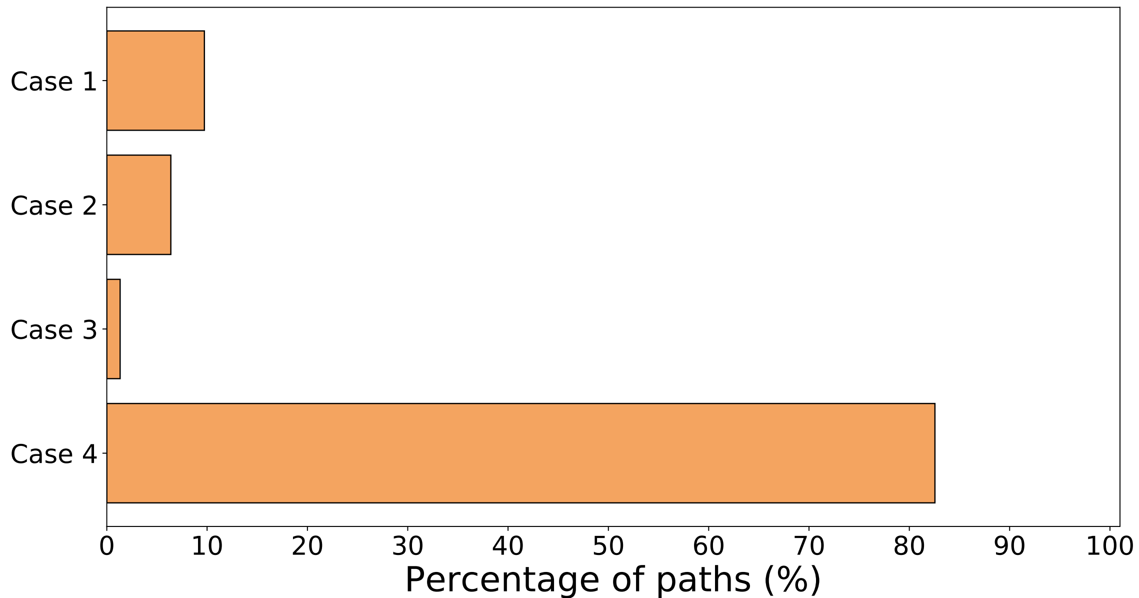
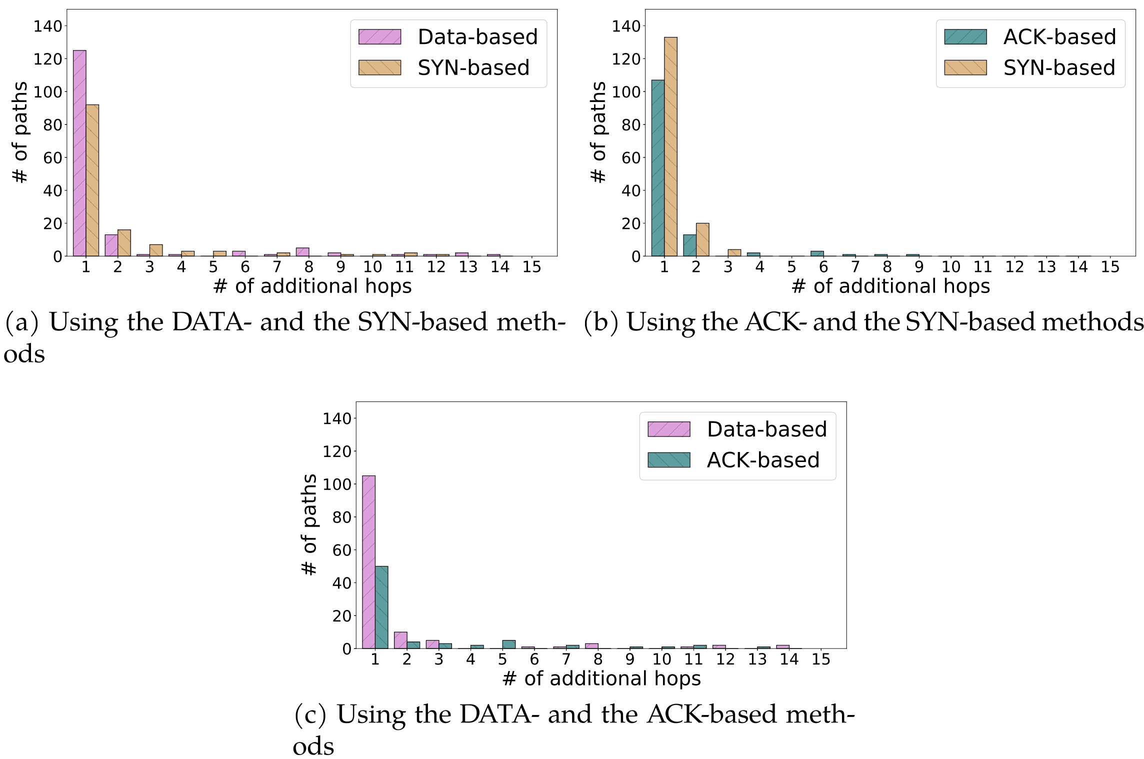


Figure 8.8: The percentage of paths for which differences in terms of number of hops can be appreciated when (1) the DATA-based methods found at least one hop that cannot be discovered using the SYN-based method, (2) the SYN-based methods found at least one hop that cannot be discovered using the DATA-based method, (3) both methods found at least one hop that cannot be discovered using the other method, or (4) the two methods found the same hops.

- **Case 2**: set of paths for which only the SYN-based method has identified at least an additional hop ($I_{DATA} \subset I_{SYN}$).
- **Case 3**: set of paths for which both the DATA- and ACK-based methods have identified at least an additional hop ($I_{DATA} \neq I_{SYN}$ and $I_{DATA}, I_{SYN} \subset (I_{DATA} \cup I_{SYN})$).
- **Case 4**: set of paths for which no additional hops can be identified ($I_{DATA} = I_{SYN}$).

The results have been plotted in Figure 8.8. First, we can see that about 83% of the paths fall into case 4. This means that, for most of the targets, the DATA- and the SYN-based methods can correctly detect the same number of hops, while only in approximately 17% of the paths the chosen algorithm is able to discover additional hops. Case 1 accounts for about 10% of the paths, whereas case 2 accounts for about 6%. This means that the DATA-based methods can provide more information than the SYN-based method for a higher amount of targets. Finally, case 3 includes only $\sim$1% amount of paths.

Then, Figure 8.9 compares the number of additional hops discovered, considering paths towards the same destination obtained using different probing methods.



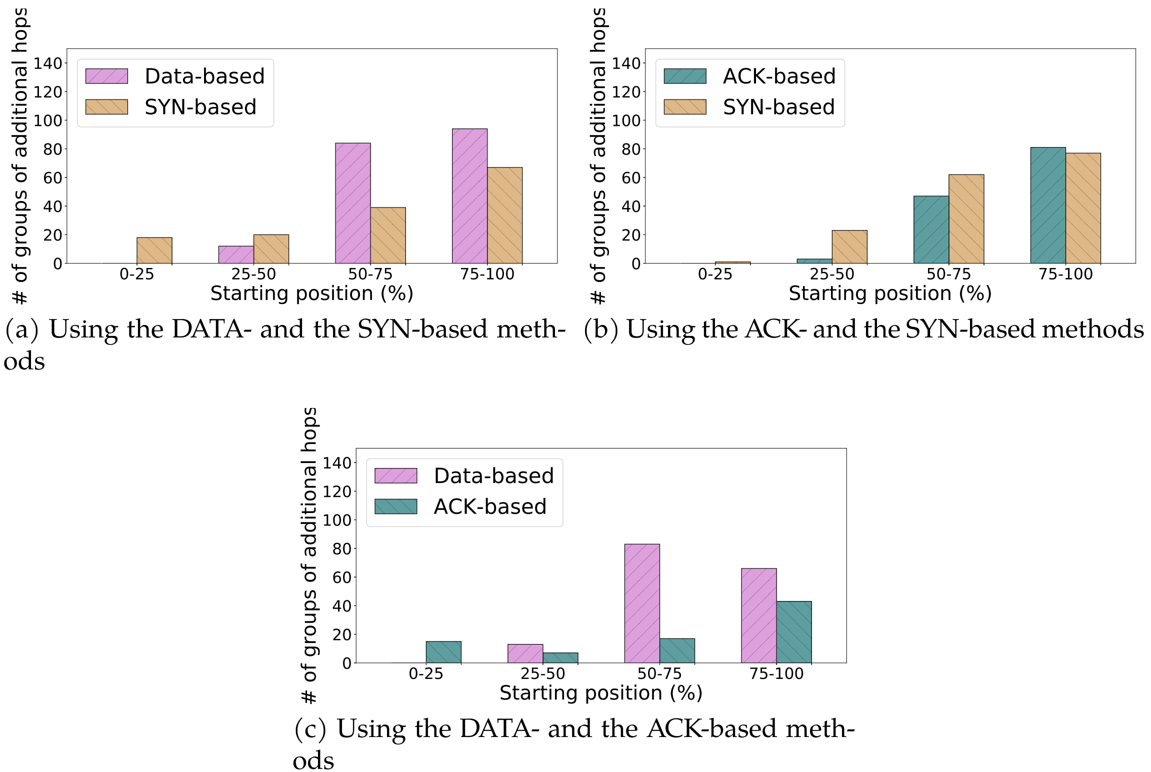
(a) Using the DATA- and the SYN-based methods

(b) Using the ACK- and the SYN-based methods

(c) Using the DATA- and the ACK-based methods

Figure 8.9: The number of additional hops discovered.

As can be seen, only a few additional hops can be identified for most of the targets. In other words, it is unlikely to find significant differences within a single path. Instead, Figure 8.10 shows where the additional hops are located. The positions are expressed as a percentage so that paths of different lengths can be compared. Note that the two connection-based methods found most of their additional hops in the last 50% of the path. This behavior could be attributed to the presence of a firewall placed in the target AS and configured to block traceroute traffic. Instead, in Figures 8.10a and 8.10c can be seen that both the ACK- and the SYN-based methods, when compared with the DATA-based method, identified some additional hops in the first 25% of the path. This means that the DATA-based methods failed in discovering some hops located in the proximity of the source machine. This phenomenon was further investigated employing Wireshark(The Wireshark Home Page, 2021), detecting two anomalous behavior to which the DATA-based version of camotrace is subjected. The first situation involved a set of servers that send a TCP window update message immediately after the connection setup. Once a window update message is received, the sender resets the TTL to its default value. This means that the GET message is sent directly to the target, and no ICMP packet is elicited. Once a GET request is received, these servers either send a new TCP window update message or close the connection. This means that the following GET requests will also be



(a) Using the DATA- and the SYN-based methods

(b) Using the ACK- and the SYN-based methods

(c) Using the DATA- and the ACK-based methods

Figure 8.10: The normalized position of groups of additional hops

sent directly to the target. At the end of the discovery procedure, the data segment-based version of camotrace can only send GET requests with TTL equal to 64, and consequently, no hop can be discovered. Instead, the second anomalous situation involved servers that did not reply to GET requests despite accepting the TCP connection. Let us try to understand in more detail the problem. At the beginning of the discovery procedure, camotrace sends the first message using a TTL of 1 and receives an ICMP message from the first intermediate router. Then, the TTL is set to its default value (see Algorithm 4, Line10). This means that the TCP retransmission of the GET request is sent to the target. However, it does not send any reply. This forces the underlying TCP layer to waste time on retransmissions while queuing all the following requests. Only one ICMP message was received from the first intermediate routers. After a while, sometimes, the server sends a TCP reset, and the connection is established for a second time. However, since the server maintains its non-responsive behavior, only an additional intermediate node can be discovered. Note that after $N_ATTEMPT_MAX$ GET requests, camotrace starts to send probe the x+1 intermediate node. At some point, after sending probes for TTL values from 1 to MAX_DEPTH, camotrace terminates, discovering very few nodes.

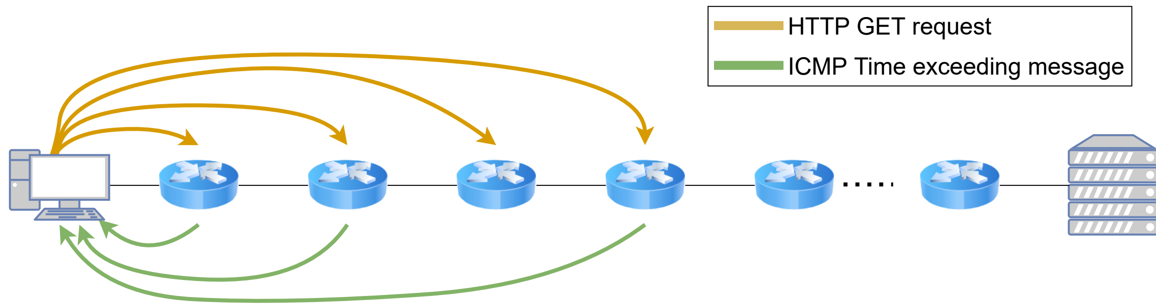


Figure 8.11: An example of d_M^{k*} computation.

Evaluating the characteristics of the discovered paths

Finally, the characteristics of the paths between the source and the destination were evaluated.

First of all, let us define a *complete path* as a path in which all the intermediate hops have been discovered. First of all, for each method, the complete paths were calculated considering only one probe attempt. For example, using the SYN-, DATA-, and ACK-based methods, 1024, 878, and 933 complete paths were found, respectively. Instead, from the union of all methods $(DATA \cup ACK \cup SYN)$, 1068 complete paths were identified. This means that most of the complete paths can be found using only the TCP traceroute. However, the combination of multiple probing mechanisms can still provide additional information. The increase in the number of complete paths identified was further investigated. The detection of these additional paths can be attributed to different probing mechanisms, or it may be due to multiple probing attempts. For this purpose, the full paths for the TCP-based methods were calculated using a N_ATTEMPT_MAX equal to 3, discovering 1046 complete paths. Note that, although the greater number of attempts has led to an increase in the number of paths discovered, the use of three different probing methods continues to have slightly better performance.

Finally, the length of each path was assessed. Let d_M^{k*} be the distance expressed in terms of encountered intermediate routers between the source node and the last responding hop, where M is the method used to probe the intermediate nodes and k is the maximum number of not responding hops allowed. For example, the path of Figure 8.11 has a d_M^{0*} of 2 since the third node did not respond and k is equal to 0 was used, d_M^{1*} is equal to 4 since the third, and the fifth router did not respond, and k is equal to 1, etc.

Then, considering values of k belonging to the $[0, 5]$ interval, we computed

$$\delta_M^{k*} = \frac{\sum_{\forall\ t_k} \left[d_M^{k*} - min\left(d_{SYN}^{k*}, d_{DATA}^{k*}, d_{ACK}^{k*}\right)\right]}{|t_k|} \tag{8.1}$$

where t_k is the subset of targets for which differences in the d^{k*} values arise. In

Method	k					
	0	*1*	*2*	*3*	*4*	*5*
SYN-based	2.10	2.29	1.61	1.50	1.41	1.39
DATA-based	2.31	2.34	2.01	2.13	2.19	2.22
ACK-based	1.84	1.93	1.90	1.91	1.88	1.81

Table 8.1: δ^{k*} values obtained with the three different probing methods

other words, δ_M^{k*} is the additional distance traveled by method M with respect to the shortest one. The values of δ_M^{k*} are summarized in Table 8.1. When low k values are used, the SYN- and the DATA-based methods have similar performances, while the ACK-based method shows poorer performance. Instead, when higher values of k are considered, the SYN-based method obtains the poorer performance while the DATA-based method remains the most successful method. Overall, this behavior suggests using the connectionless traceroute mechanism to explore most of the network (because of its simplicity and effectiveness), switching to DATA/ACK-based probing in the presence of non-responsive segments of the path close to the destination.

Chapter 9

Conclusions

The MEC architecture is a network paradigm based on shifting storage and computing capabilities from centralized remote clouds to the edge of the network. Basically, the MEC architecture introduces lower latencies, higher throughput, improved privacy, and reduced network congestion compared to the classic cloud architecture.

In this thesis, a methodology aimed at collecting network performance metrics in a MEC network was presented. Precisely, the collection methodology makes use of several software components, which cooperate to collect and store metrics following MECPerf-active, MECPerf-passive, self-active, and self-passive approaches. Each of these different measurement methods enables the collection of network metrics from a different perspective. Starting from this general idea, a tool called MECPerf was developed. Then, MECPerf was used during an extensive set of edge-based experiments performed using MECPerf-active, MECPerf-passive, and self-passive measurement methods. Moreover, the experiments involved TNs connected both via Wi-Fi and LTE connection and considered different network and server workloads. The results of MECPerf-active experiments displayed that slightly higher bandwidth and significantly lower latency metrics can be achieved by considering an edge server. Consequently, applications that require high bandwidth or strict latency requirements should run on edge servers. Furthermore, bandwidth metrics collected considering TNs connected using a Wi-Fi connection were revealed to be significantly affected by the presence of cross-traffic. The same did not occur for experiments based on LTE connections. Finally, MECPerf- and self-passive experiments had shown that higher bandwidth metrics can be achieved by using a cloud server if the server was under a high workload. Furthermore, latency metrics have also been shown to be affected by the number of users. This indicates that the choice of server placement cannot be based only on network metrics but should also take into account the system workload since cloud servers are generally equipped with more computing power than edge servers. After the experimental phase, a last series of measures aimed at evaluating the computational load of MECPerf was per-

formed. More details about this topic can be found in Appendix A. Finally, it was presented a library, called MECPerfLibrary, which provides an API for accessing the results collected during the experimental phase. Precisely, the library is capable of processing the raw metrics and generating bandwidth and latency traces based on experimental results. These traces can be used by researchers to emulate a MEC network under specific network conditions and do not require in-depth knowledge of the structure of the database in which the results are stored.

Then, the impact of the communication latency upon the energy consumption of an LTE TN operating in a MEC environment was investigated. This evaluation can be particularly beneficial to all those applications that are concerned with battery-powered devices as they need to optimize their overall power consumption as much as possible. First, the FSM model of the LTE interface was provided and an analytical energy consumption model for connectionless applications had been provided. Then, the analytical model had been integrated with real-world measurement results to deal with the complexities introduces by connection-oriented protocols. The results revealed that in a connectionless scenario using an edge server is generally the best choice and that the benefit introduced by edge servers are generally in the order of 5%. However, there exist also some operating conditions where cloud servers proved to be the best choice. Instead, in a connection-oriented scenario, the best choice in terms of energy consumption always involved the edge server, which can introduce benefits in the order of 30-40%. These differences between connectionless and connection-oriented applications can be explained as follows. The TCP throughput is highly influenced by the RTT between the two end-points. As a consequence, when the client interacts with a remote cloud server using a connection-oriented application, the throughput grows more slowly because of the higher RTT. In the end, the amount of time needed to send and receive the data using a cloud server is higher, and the interface is forced to remain continuously turned on for a longer interval, increasing the overall energy consumption. Instead, the same did not emerge for UDP-based applications, as they lack rate-limiting mechanisms. Consequently, the time required to send (or receive) some data is independent of the location of the server, which had a marginal impact on the energy needed to transmit the data.

Finally, a study aimed at evaluating the paths that separate a TN and an application server was presented. Fundamentally, the main idea is to exploit pre-existing TCP connections to mask probing mechanisms within HTTP traffic, to eventually elude filtering mechanisms. This allowed us to deepen our knowledge of the network paths that separate TNs from cloud servers. Furthermore, this type of knowledge can be used by orchestrators to improve the complexity of placement strategies. With this goal was developed camotrace, a tracerouting mechanism based on the use of TCP connections. A set of experiments were then performed using the target Web

servers, and then the results obtained with the two versions of camotrace had been compared with those obtained using the TCP traceroute. The results showed that differences emerged only for some destinations. In fact, for approximately 10% of the paths, the version of camotrace based on the usage of HTTP messages was able to find more information than TCP traceroute, while the opposite happened only for about 6% of the targets. Furthermore, the two versions of camotrace were able to discover more IP interfaces, more ASs, and more AS links than those found by the TCP traceroute. These results demonstrate that connection-based probing methods are capable of gathering information that could not be obtained through the traditional TCP traceroute. However, it loses some other information. Hence, to get a complete picture of the network, probing methods based on established connections should be used in conjunction with classic connectionless traceroute approaches. Finally, we found that most of the differences emerged in the last 50% of the path where it is reasonable to assume that most of the classification and filtering systems are located. This could confirm the presence of traceroute suppression mechanisms in the proximity of the destination host.

Appendix A

Evaluating the computational load of MECPerf

Finally, a last set of experiments aimed at evaluating the computational load of the MECPerf collection system were performed. We executed the MECPerf components on three machines belonging to the network of the University of Pisa. Precisely, the first machine hosted an MC, the second hosted an MO, and finally the third hosted an MRS and an MA. The technical characteristics of the three machines used during the experiment are reported in Table A.1. The setup employed during the experiments is pictured in Figure A.1.

Each experiment consists of a set of TCP MECPerf-active bandwidth measures in the uplink direction. Each of these measures is initiated by the MC immediately after the end of the previous one. Then, after the beginning of the first measure, a CPU performance monitoring session was started on the MO. Note that a MECPerf-active measurement can be initiated only by the MC and it assesses the performance of both the access-MEC (i.e., the network segment that connects the MC and the

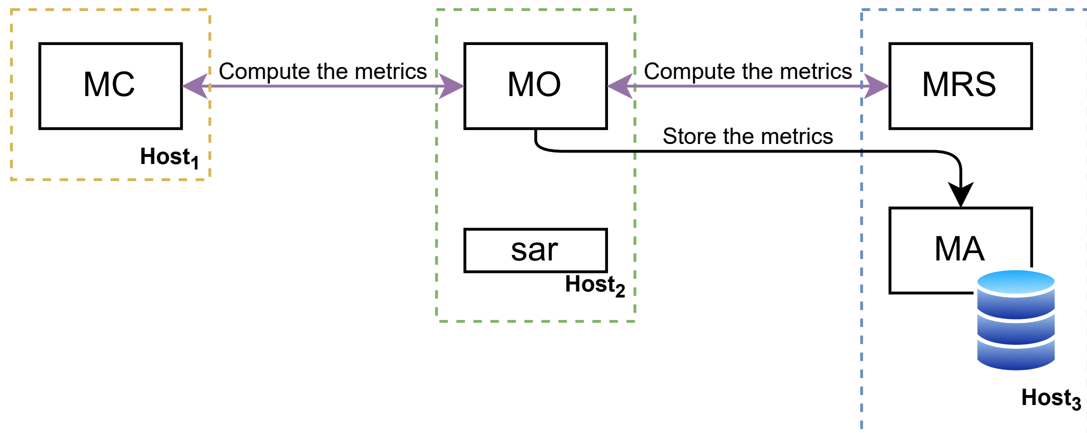


Figure A.1: The setup used to assess the computational load of MECPerf.

Table A.1: The technical characteristics of the devices used to assess the computational load of MECPerf.

	Host 1	**Host 2**	**Host 3**
OS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.6 LTS	Ubuntu 18.04.3 LTS
CPU	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
Number of cores	2	4	2
RAM	4GiB	8GiB	4GiB

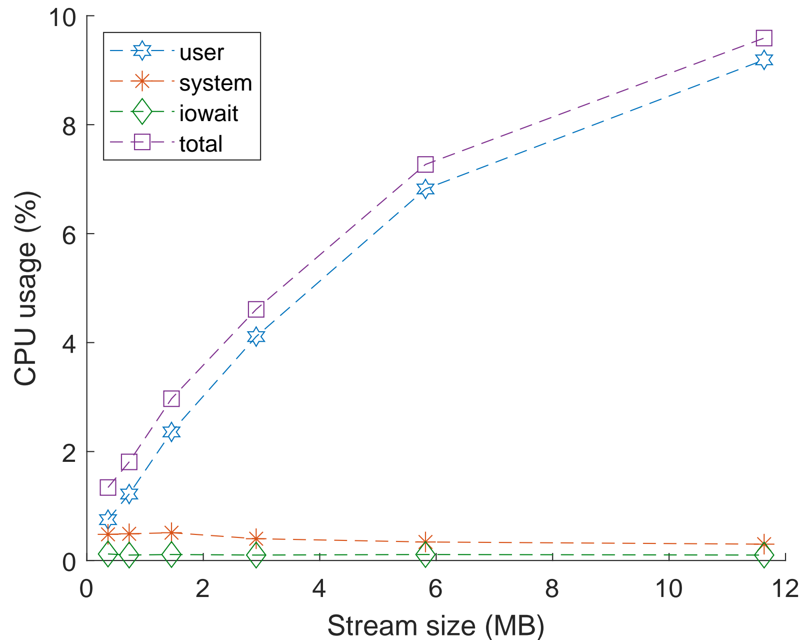


Figure A.2: The CPU usage rate measured on the MO during a set of experiments aimed at evaluating the computational load of MECPerf. Precisely, each value represents the average rate monitored for 1000 seconds during a set of TCP uplink measures.

MO) and the MEC-cloud network segment (i.e., the network segment that connects the MO and the MRS). This means that the MO is always involved in a measure or in the upload of the results of a measure. Hence, we decided to monitor the performance of the MO as it is the node under the greatest computational load. The CPU usage rate of the MO was monitored using sar(sar(1) — Linux manual page, 2020), a tool contained in the sysstat package (sysstat - System performance tools for the Linux operating system, 2022). Each monitoring session lasted 1000 seconds. The average CPU usage rates are shown in Figure A.2. As can be noted only a marginal amount of CPU is used by the system, I/O, and other non-user processes, while most of the CPU usage can be ascribed to the MO. In addition, the amount of CPU required increases as the amount of data used to perform the

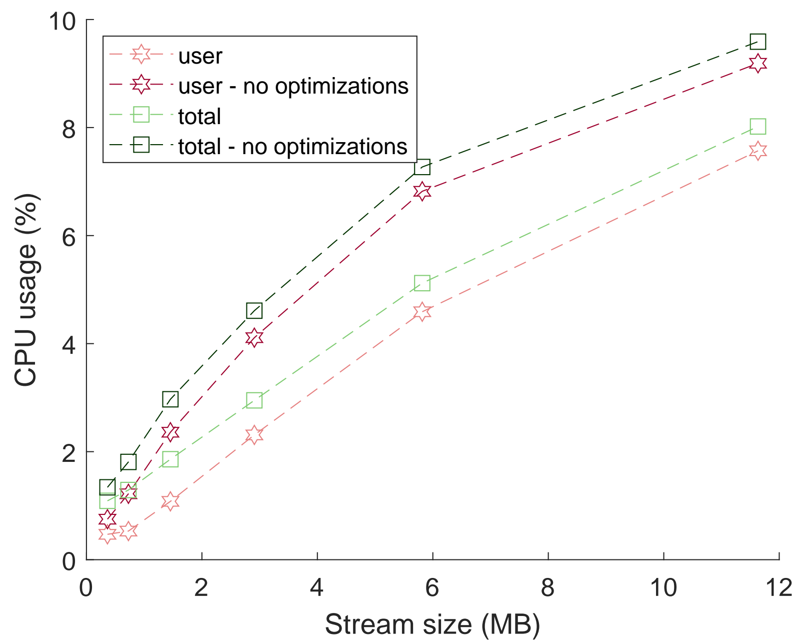


Figure A.3: The CPU usage rate with and without the optimizations applied to the measurement library.

measurements increases. If the MO would be hosted on constrained devices this may become a problem. To overcome this problem we implemented an optimized version of the measurement library. This latest version was still developed in Java and makes use of the SocketChannel and the ByteBuffer classes. Then, we ran a new set of experiments to evaluate the CPU usage of the MECPerf collection system. The results, shown in Figure A.3, demonstrate that the new version of the library succeed in reducing the amount of CPU used by the MECPerf collection system.

Appendix B

List of Acronyms

1-D CNN 1-Dimensional Convolutional Neural Networks

ABR Adaptive Bitrate

AI Artificial Intelligence

AR Augmented Reality

AS Autonomous System

AP Access Point

API Application Programming Interface

ASR Automatic Speech Recognition

BS Base Station

CNN Convolutional Neural Networks

CR Continuous Reception

DASH Dynamic Adaptive Streaming over HTTP protocol

DNN Deep Neural Networks

DRX Discontinuous Reception

eCDF empirical Cumulative Distribution Function

ETSI European Telecommunications Standards Institute

FSM Finite State Machine

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IoT Internet of Things

LTE Long Term Evolution

MA MECPerf Aggregator

MBS Macro Base Station

MC MECPerf Client

MEC Multi-access Edge Computing

ML Machine Learning

MO MECPerf Observer

MR Mixed Reality

MRS MECPerf Remote Server

MTC Machine Type Communication

MTU Maximum Transmission Unit

NGI Next Generation Internet

NTM NetworkTraceManager

OS Operating System

QoE Quality of Experience

RFC Request for Comments

RNN Recurrent Neural Networks

RSRP Reference Signal Received Power

RTT Round Trip Time

SBS Small Base Station

SDN Software Defined Networking

SLA Service Level Agreement

TCP Transmission Control Protocol

TLD Top-Level Domain

TN Terminal Node

TTL Time To Live

V2E Vehicle to Everything

VM Virtual Machine

VR Virtual Reality

Appendix C

Publications

Finally, the paper done during the Doctorate will be listed below. For each work, the candidate's contributions will be explicitly listed. The roles taxonomy adopted is summarized in Table C.1 and it can be consulted at the following link [1].

Journal papers

1. **C. Caiazza**, V. Luconi, A. Vecchio, "TCP-based traceroute: An evaluation of different probing methods", *Internet Technology Letters*, pages: 6, 2020, Wiley, DOI: https://doi.org/10.1002/itl2.134. **Candidate's contributions**: Conceptualization, Investigation, Methodology, Software, Validation, Writing – Original Draft.

2. **C. Caiazza**, C. Cicconetti, V. Luconi, A. Vecchio, "Measurement-driven design and runtime optimization in edge computing: Methodology and tools", *Computer Networks*, pages: 108–140, 2021, Elsevier, DOI: https://doi.org/10.1016/j.comnet.2021.108140. **Candidate's contributions**: Software, Formal analysis, Data curation, Writing – Original Draft, Visualization.

Peer reviewed conference papers

1. **C. Caiazza**, E. Gregori , V. Luconi, F. Mione, A. Vecchio, "Application-Level Traceroute: Adopting Mimetic Mechanisms to Increase Discovery Capabilities", *International Conference on Wired/Wireless Internet Communications (WWIC)*, pages: 66–77 , 2019, DOI: https://doi.org/10.1007/978-3-030-30523-9_6. **Candidate's contributions**: Conceptualization, Investigation, Methodology, Software, Validation, Writing – Original Draft.

2. **C. Caiazza**, L. Bernardi, M. Bevilacqua, A. Cabras, C. Cicconetti, V. Luconi, G. Sciurti, E. Senore, E. Vallati, A. Vecchio, "MECPerf: An Application-Level

[1] https://casrai.org/credit/

Tool for Estimating the Network Performance in Edge Computing Environments", *Computers, Software, and Applications Conference* (*COMPSAC*), pages: 1163–1168, 2020, IEEE, DOI: https://doi.org/10.1109/COMPSAC48688.2020.00-99. **Candidate's contributions**: Investigation, Methodology, Software, Writing – Original Draft

Other

1. **C. Caiazza**, S. Giordano, V. Luconi, A. Vecchio, "Edge Computing vs Centralized Cloud: Impact of Communication Latency on the Energy Consumption of LTE Terminal Nodes", ArXiv preprints, to be submitted to a journal, https://arxiv.org/abs/2111.10076. **Candidate's contributions**: Conceptualization, Investigation, Methodology, Software, Validation, Writing – Original Draft.

Conceptualization	Ideas; formulation or evolution of overarching research goals and aims.
Data curation	Management activities to annotate (produce metadata), scrub data and maintain research data (including software code, where it is necessary for interpreting the data itself) for initial use and later re-use.
Formal analysis	Application of statistical, mathematical, computational, or other formal techniques to analyze or synthesize study data.
Investigation	Conducting a research and investigation process, specifically performing the experiments, or data/evidence collection.
Methodology	Development or design of methodology; creation of models.
Software	Programming, software development; designing computer programs; implementation of the computer code and supporting algorithms; testing of existing code components.
Validation	Verification, whether as a part of the activity or separate, of the overall replication/reproducibility of results/experiments and other research outputs.
Visualization	Preparation, creation and/or presentation of the published work, specifically visualization/data presentation.
Writing – original draft	Preparation, creation and/or presentation of the published work, specifically writing the initial draft (including substantive translation).

Table C.1: CRediT – Contributor Roles Taxonomy.

Bibliography

5G-ACIA (2019). 5G Non-Public Networks for Industrial Scenarios. Technical Report March.

5GAA (2017). Toward fully connected vehicles : Edge computing White Paper. Technical report.

About Fed4Fire+ (Last accessed May 2022). About Fed4Fire+. `https://www.fed4fire.eu/the-project/`.

Apostolis, A. (2020). Results of the MECinFire experiment in Fed4FIRE+ EU project.

ARK (Accessed Oct 2019). The Cooperative Association for Internet Data Analysis Archipelago Measurement Infrastructure (CAIDA Ark). `http://www.caida.org/projects/ark/`.

Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., Friedman, T., Latapy, M., Magnien, C., and Teixeira, R. (2006). Avoiding Traceroute Anomalies with Paris Traceroute. In *Proc. ACM SIGCOMM IMC '06*, pages 153–158. ACM SIGCOMM IMC '06.

Augustin, B., Friedman, T., and Teixeira, R. (2007). Multipath tracing with Paris traceroute. In *Proc. IEEE/IFIP E2EMON '07*, pages 1–8. IEEE/IFIP E2EMON '07.

AWS for the Edge (Last accessed May 2022). Aws for the edge. `https://aws.amazon.com/it/edge/`.

Bao, W., Wong, V. W. S., and Leung, V. C. M. (2010). A Model for Steady State Throughput of TCP CUBIC. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–6.

Bellavista, P., Berrocal, J., Corradi, A., Das, S. K., Foschini, L., and Zanni, A. (2019). A survey on fog computing for the Internet of Things. *Pervasive and Mobile Computing*, 52:71–99.

Beverly, R. (2016). Yarrp'ing the internet: Randomized high-speed active topology discovery. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 413–420, New York, NY, USA. Association for Computing Machinery.

Brand, P., Falk, J., Ah Sue, J., Brendel, J., Hasholzner, R., and Teich, J. (2020). Adaptive Predictive Power Management for Mobile LTE Devices. *IEEE Transactions on Mobile Computing*, pages 1–1.

Braud, T., Bijarbooneh, F. H., Chatzopoulos, D., and Hui, P. (2017a). Future Networking Challenges: The Case of Mobile Augmented Reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1796–1807.

Braud, T., Bijarbooneh, F. H., Chatzopoulos, D., and Hui, P. (2017b). Future Networking Challenges: The Case of Mobile Augmented Reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1796–1807. IEEE.

Caiazza, C., Cicconetti, C., Luconi, V., and Vecchio, A. (2021). Measurement-driven design and runtime optimization in edge computing: Methodology and tools. *Computer Networks*, 194:108140.

Campbell, M. (2019). Smart Edge : The the Center of Data Gravity Out of the Cloud. *Computer*, 52(December):99–102.

Candela, M., Gregori, E., Luconi, V., and Vecchio, A. (2019). Using RIPE Atlas for Geolocating IP Infrastructure. *IEEE Access*, 7:48816–48829.

Caprolu, M., Di Pietro, R., Lombardi, F., and Raponi, S. (2019). Edge computing perspectives: Architectures, technologies, and open security issues. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 116–123.

Cardwell, N., Savage, S., and Anderson, T. (1998). Modeling the performance of short TCP connections. *Techical Report*.

Chang, H., Jamin, S., and Willinger, W. (2001). Inferring AS-level Internet Topology from Router-Level Path Traces. In *Proc. SPIE ITCom '01*, pages 196–207. SPIE ITCom '01.

Chen, X., Ding, N., Jindal, A., Hu, Y. C., Gupta, M., and Vannithamby, R. (2015). Smartphone Energy Drain in the Wild: Analysis and Implications. *SIGMETRICS Perform. Eval. Rev.*, 43(1):151–164.

Cheng, S., Xu, Z., Li, X., Wu, X., Fan, Q., Wang, X., and Leung, V. C. M. (2020). Task offloading for automatic speech recognition in edge-cloud computing based mobile networks. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6.

Cheswick, B., Burch, H., and Branigan, S. (1999). Mapping the Internet. *IEEE Computer*, 32(4):97–98, 102.

claffy, k., Hyun, Y., Keys, K., Fomenkov, M., and Krioukov, D. (2009). Internet Mapping: From Art to Science. In *Proc. DHS CATCH '09*, pages 205–211. DHS CATCH '09.

Cloudflare - Security and innovation at the network edge (Last accessed May 2022). Cloudflare - Security and innovation at the network edge. `https://www.cloudflare.com/edge/`.

Consortium GARR Home Page (Accessed Aug 2021). Consortium garr home page. `https://www.garr.it/`.

Das, A., Patterson, S., and Wittie, M. (2019). EdgeBench: Benchmarking edge computing platforms. *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, pages 175–180.

De Vita, F., Nocera, G., Bruneo, D., Tomaselli, V., Giacalone, D., and Das, S. K. (2021). Porting deep neural networks on the edge via dynamic k-means compression: A case study of plant disease detection. *Pervasive and Mobile Computing*, 75:101437.

Detal, G., Hesmans, B., Bonaventure, O., Vanaubel, Y., and Donnet, B. (2013). Revealing Middlebox Interference with Tracebox. In *Proc. ACM SIGCOMM IMC '13*, pages 1–8. ACM SIGCOMM IMC '13.

Donnet, B. (2013). Internet Topology Discovery. In *Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience*, pages 44–81. Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience, Springer Berlin Heidelberg.

Dovrolis, C., Ramanathan, P., and Moore, D. (2004). Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Trans. Netw.*, 12(6):963–977.

Emara, M., Filippou, M., and Sabella, D. (2018). MEC-Assisted End-to-End Latency Evaluations for C-V2X Communications. In *Proc. EuCNC '18*, pages 157–161.

Ericsson Mobility Report (published on November 2021). Ericsson mobility report. `https://www.ericsson.com/4ad7e9/assets/local/reports-papers/mobility-report/documents/2021/ericsson-mobility-report-november-2021.pdf`.

ETSI (2018). Multi-access Edge Computing (MEC); Phase 2: Use Cases and Requirements; ETSI GS MEC 002 V2.1.1. Technical report.

Faggiani, A., Gregori, E., Lenzini, L., Luconi, V., and Vecchio, A. (2014). Smartphone-based crowdsourcing for network monitoring: Opportunities, challenges, and a case study. *IEEE Commun. Mag.*, 52(1):106–113.

Faggiani, A., Gregori, E., Lenzini, L., Mainardi, S., and Vecchio, A. (2012). On the feasibility of measuring the Internet through smartphone-based crowdsourcing. In *Proc. WiOpt '12*, pages 318–323. WiOpt '12.

Fed4Fire+ (Project started on January 2017). Fed4Fire+. `https://www.fed4fire.eu/`.

FED4FIRE+ - MECPerf (Last accessed May 2022). FED4FIRE+ - MECPerf. `https://www.fed4fire.eu/demo-stories/oc6/mecperf/#presentations`.

Fernández-Cerero, D., Fernández-Montes, A., Javier Ortega, F., Jakóbik, A., and Widlak, A. (2020). Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption. *Simulation Modelling Practice and Theory*, 101:101966. Modeling and Simulation of Fog Computing.

Fiandrino, C., Blanco Pizarro, A., Jiménez Mateo, P., Andrés Ramiro, C., Ludant, N., and Widmer, J. (2019). openLEON: An end-to-end emulation platform from the edge data center to the mobile user. *Computer Communications*, 148:17–26.

Filippou, M. C., Sabella, D., Emara, M., Prabhakaran, S., Shi, Y., Bian, B., and Rao, A. (2020). Multi-Access Edge Computing: A Comparative Analysis of 5G System Deployments and Service Consumption Locality Variants. *IEEE Communications Standards Magazine*, 4(2):32–39.

flask (Accessed on: Aug 2021). Flask's documentation. `https://flask.palletsprojects.com/en/2.0.x/`.

Giust, F., Sciancalepore, V., Sabella, D., Filippou, M. C., Mangiante, S., Featherstone, W., and Munaretto, D. (2018). Multi-access edge computing: The driver behind the wheel of 5g-connected cars. *IEEE Communications Standards Magazine*, 2(3):66–73.

Gomez-Miguelez, I., Garcia-Saavedra, A., Sutton, P. D., Serrano, P., Cano, C., and Leith, D. J. (2016). Srslte: An open-source platform for lte evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, WiNTECH '16, page 25–32, New York, NY, USA. Association for Computing Machinery.

Gregori, E., Improta, A., Lenzini, L., Luconi, V., Redini, N., and Vecchio, A. (2016). Smartphone-based crowdsourcing for estimating the bottleneck capacity in wireless networks. *Journal of Network and Computer Applications*, 64:62 – 75.

Gregori, E., Lenzini, L., Luconi, V., and Vecchio, A. (2013). Sensing the Internet through crowdsourcing. In *Proc. PerMoby '13*, pages 248–254. PerMoby '13.

Gregori, E., Luconi, V., and Vecchio, A. (2018). Studying forwarding differences in european mobile broadband with a net neutrality perspective. In *European Wireless 2018; 24th European Wireless Conference*, pages 1–7.

Guidelines on the Implementation by National Regulators of European Net Neutrality Rules (2016). (2016). Guidelines on the implementation by national regulators of european net neutrality rules (2016). `http://berec.europa.eu/eng/document_register/subject_matter/berec/download/0/6160-berec-guidelines-on-the-implementationb_0.pdf`.

Guo, H., Liu, J., and Zhang, J. (2018). Computation Offloading for Multi-Access Mobile Edge Computing in Ultra-Dense Networks. *IEEE Communications Magazine*, 56(8):14–19.

Gupta, H., Vahid Dastjerdi, A., Ghosh, S. K., and Buyya, R. (2017). ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296.

Hao, P. and Wang, X. (2019). Integrating PHY Security Into NDN-IoT Networks By Exploiting MEC: Authentication Efficiency, Robustness, and Accuracy Enhancement. *IEEE Transactions on Signal and Information Processing over Networks*, 5(4):792–806.

Hao, Y., Jiang, Y., Chen, T., Cao, D., and Chen, M. (2019). itaskoffloading: Intelligent task offloading for a cloud-edge collaborative system. *IEEE Network*, 33(5):82–88.

Harutyunyan, D., Shahriar, N., Boutaba, R., and Riggio, R. (2019). Latency-aware service function chain placement in 5g mobile networks. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 133–141.

Hong, C. H. and Varghese, B. (2019). Resource management in fog/Edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys*, 52(5).

Hu, W., Gao, Y., Ha, K., Wang, J., Amos, B., Chen, Z., Pillai, P., and Satyanarayanan, M. (2016a). Quantifying the Impact of Edge Computing on Mobile Applications. In *Proc. ACM SIGOPS APSys '16*, pages 5:1–5:8.

Hu, W., Gao, Y., Ha, K., Wang, J., Amos, B., Chen, Z., Pillai, P., and Satyanarayanan, M. (2016b). Quantifying the Impact of Edge Computing on Mobile Applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA. Association for Computing Machinery.

Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. (2015). Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16.

Huang, J., Qian, F., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O. (2012). A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 225–238, New York, NY, USA. Association for Computing Machinery.

Huang, Y., Rabinovich, M., and Al-Dalky, R. (2020). Flashroute: Efficient traceroute on a massive scale. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 443–455, New York, NY, USA. Association for Computing Machinery.

Industrial Internet Consortium (2019). The Edge Computing Advantage. Technical report.

Jang, M., Lee, H., Schwan, K., and Bhardwaj, K. (2016). SOUL: An Edge-Cloud System for Mobile Applications in a Sensor-Rich World. In *Proc. IEEE/ACM SEC '16*, pages 155–167.

Jiang, C., Fan, T., Gao, H., Shi, W., Liu, L., Cérin, C., and Wan, J. (2020). Energy aware edge computing: A survey. *Computer Communications*, 151:556–580.

Kekki, S., Featherstone, W., Fang, Y., Kuure, P., Li, A., Ranjan, A., Purkayastha, D., Jiangping, F., Frydman, D., Verin, G., Wen, K.-W., Kim, K., Arora, R., Odgers, A., Contreras, L. M., and Scarpina, S. (2018). ETSI White Paper: MEC in 5G networks. Technical Report 28, ETSI.

Keys, K. (2010). Internet-scale ip alias resolution techniques. *ACM SIGCOMM Comput. Commun. Rev.*, 40(1):50–55.

Keys, K., Hyun, Y., Luckie, M., and Claffy, K. (2013). Internet-scale ipv4 alias resolution with midar. *IEEE/ACM Transactions on Networking*, 21(2):383–399.

Kolosov, O., Yadgar, G., Maheshwari, S., and Soljanin, E. (2020). Benchmarking in the dark: On the absence of comprehensive edge datasets. *HotEdge 2020 - 3rd USENIX Workshop on Hot Topics in Edge Computing*.

Krupitzer, C., Wagenhals, T., Züfle, M., Lesch, V., Schäfer, D., Mozaffarin, A., Edinger, J., Becker, C., and Kounev, S. (2020). A survey on predictive maintenance for industry 4.0.

Lera, I., Guerrero, C., and Juiz, C. (2019). YAFS: A Simulator for IoT Scenarios in Fog Computing. *IEEE Access*, 7:91745–91758.

Lex, A., Gehlenborg, N., Strobelt, H., Vuillemot, R., and Pfister, H. (2014). UpSet: Visualization of Intersecting Sets. *IEEE Trans. Vis. Comput. Graph.*, 20(12):1983–1992.

Li, W., Deng, W., She, R., Zhang, N., Wang, Y., and Ma, W. (2021a). Edge Computing Offloading Strategy Based on Particle Swarm Algorithm for Power Internet of Things. In *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 145–150.

Li, X., Zhao, L., Yu, K., Aloqaily, M., and Jararweh, Y. (2021b). A cooperative resource allocation model for iot applications in mobile edge computing. *Computer Communications*, 173:183–191.

Lin, H., Zeadally, S., Chen, Z., Labiod, H., and Wang, L. (2020a). A survey on computation offloading modeling for edge computing. *Journal of Network and Computer Applications*, page 102781.

Lin, Z., Jain, A., Wang, C., Fanti, G., and Sekar, V. (2020b). Using gans for sharing networked time series data: Challenges, initial promise, and open questions. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 464–483, New York, NY, USA. Association for Computing Machinery.

Liu, J., Xiao, S., Li, Y., Song, H., Jin, D., and Su, L. (2019). Netwatch: End-to-end network performance measurement as a service for cloud. *IEEE Transactions on Cloud Computing*, 7(2):553–567.

Liu, Y., Peng, M., Shou, G., Chen, Y., and Chen, S. (2020a). Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things. *IEEE Internet of Things Journal*, 7(8):6722–6747.

Liu, Z., Wu, Z., Gan, C., Zhu, L., and Han, S. (2020b). Datamix: Efficient privacy-preserving edge-cloud inference. In Vedaldi, A., Bischof, H., Brox, T., and Frahm, J.-M., editors, *Computer Vision – ECCV 2020*, pages 578–595, Cham. Springer International Publishing.

Luckie, M. (2010). Scamper: A scalable and extensible packet prober for active measurement of the internet. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 239–245, New York, NY, USA. Association for Computing Machinery.

Luckie, M., Hyun, Y., and Huffaker, B. (2008a). Traceroute probe method and forward ip path inference. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, page 311–324, New York, NY, USA. Association for Computing Machinery.

Luckie, M., Hyun, Y., and Huffaker, B. (2008b). Traceroute Probe Method and Forward IP Path Inference. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, pages 311–324.

Madhyastha, H. V., Isdal, T., Piatek, M., Dixon, C., Anderson, T., Krishnamurthy, A., and Venkataramani, A. (2006). iPlane: An Information Plane for Distributed Services. In *Proc. USENIX OSDI '06*, pages 367–380. USENIX OSDI '06.

Marchetta, P. and Pescapé, A. (2013). Drago: Detecting, quantifying and locating hidden routers in traceroute ip paths. In *2013 Proceedings IEEE INFOCOM*, pages 3237–3242.

Mathis, M., Semke, J., Mahdavi, J., and Ott, T. (1997). The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82.

MATLAB Engine API for C++ (Accessed on 2021). MATLAB Engine API for C++ - Documentation. `https://www.mathworks.com/help/matlab/calling-matlab-engine-from-cpp-programs.html`.

Maxmind (Accessed Aug 2021). MaxMind GeoLite2 Databases. `https://dev.maxmind.com/geoip/geoip2/geolite2/`.

Mazouzi, H., Boussetta, K., and Achir, N. (2019). Maximizing mobiles energy saving through tasks optimal offloading placement in two-tier cloud: A theoretical and an experimental study. *Computer Communications*, 144:132–148.

McChesney, J., Wang, N., Tanwer, A., de Lara, E., and Varghese, B. (2019). Defog: Fog computing benchmarks. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 47–58, New York, NY, USA. Association for Computing Machinery.

MECPerf experimentation results (Dataset published on April 26, 2020). MECPerf experimentation results. `https://zenodo.org/record/4647753#.YGNLDnUzZH4`.

Mehmood, Y., Zhang, L., and Förster, A. (2019). Power Consumption Modeling of Discontinuous Reception for Cellular Machine Type Communications. *Sensors*, 19(3).

Mlab (Accessed Oct 2019). Measurement Lab (M-Lab). `https://www.measurementlab.net/`.

Monsoon Power Monitor (Accessed Jan 2022). Monsoon Power Monitor. `https://www.msoon.com/high-voltage-power-monitor`.

Moors, T. (2004). Streamlining traceroute by estimating path lengths. In *2004 IEEE International Workshop on IP Operations and Management*, pages 123–128.

Morandi, I., Bronzino, F., Teixeira, R., and Sundaresan, S. (2019). Service traceroute: Tracing paths of application flows. In Choffnes, D. and Barcellos, M., editors, *Passive and Active Measurement*, pages 116–128, Cham. Springer International Publishing.

Napolitano, A., Cecchetti, G., Giannone, F., Ruscelli, A., Civerchia, F., Kondepu, K., Valcarenghi, L., and Castoldi, P. (2019). Implementation of a MEC-based vulnerable road user warning system. In *Proc. AEIT AUTOMOTIVE '19*.

Nath, S. and Wu, J. (2020). Deep reinforcement learning for dynamic computation offloading and resource allocation in cache-assisted mobile edge computing systems. *Intelligent and Converged Networks*, 1(2):181–198.

Nauman, A., Qadri, Y. A., Amjad, M., Zikria, Y. B., Afzal, M. K., and Kim, S. W. (2020). Multimedia internet of things: A comprehensive survey. *IEEE Access*, 8:8202–8250.

netem (Accessed on: March 2020). NetEm - Network Emulator. `http://man7.org/linux/man-pages/man8/tc-netem.8.html`.

netfilter project home page (Accessed Aug 2021). netfilter project home page. `https://www.netfilter.org/`.

netlink(7) — Linux manual page (Accessed Aug 2021). netlink(7) — linux manual page. `https://man7.org/linux/man-pages/man7/netlink.7.html`.

Nguyen, Q.-H., Morold, M., David, K., and Dressler, F. (2020). Car-to-pedestrian communication with mec-support for adaptive safety of vulnerable road users. *Comput. Commun.*, 150:83 – 93.

Nguyen, T. D. and Huh, E. N. (2018). ECSim++: An INET-based simulation tool for modeling and control in edge cloud computing. *Proceedings - 2018 IEEE International Conference on Edge Computing, EDGE 2018 - Part of the 2018 IEEE World Congress on Services*, pages 80–86.

Nielsen, H., Mogul, J., Masinter, L. M., Fielding, R. T., Gettys, J., Leach, P. J., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.

Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. (1998). Modeling TCP Throughput: A Simple Model and Its Empirical Validation. *SIGCOMM Comput. Commun. Rev.*, 28(4):303–314.

Palo Alto Networks (Accessed Aug 2021). Palo Alto Networks. `https://www.paloaltonetworks.com/`.

Pan, J. and McElhannon, J. (2018). Future Edge Cloud and Edge Computing for Internet of Things Applications. *IEEE Internet of Things Journal*, 5(1):439–449.

Pei, Y., Peng, Z., Wang, Z., Wang, H., and Fernandez-Veiga, M. (2020). Energy-Efficient Mobile Edge Computing: Three-Tier Computing under Heterogeneous Networks. *Wirel. Commun. Mob. Comput.*, 2020.

Peuster, M., Karl, H., and van Rossem, S. (2016). Medicine: Rapid prototyping of production-ready network services in multi-pop environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 148–153.

Porambage, P., Okwuibe, J., Liyanage, M., Ylianttila, M., and Taleb, T. (2018). Survey on multi-access edge computing for internet of things realization. *IEEE Communications Surveys Tutorials*, 20(4):2961–2991.

Postel, J. (1981). Internet Control Message Protocol – DARPA Internet Program Protocol Specification. RFC 792.

Prasad, R., Dovrolis, C., Murray, M., and Claffy, K. (2003). Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6):27–35.

Qadri, Y. A., Nauman, A., Zikria, Y. B., Vasilakos, A. V., and Kim, S. W. (2020). The future of healthcare internet of things: A survey of emerging technologies. *IEEE Communications Surveys Tutorials*, 22(2):1121–1167.

Qayyum, T., Malik, A. W., Khattak, M. A., Khalid, O., and Khan, S. U. (2018). FogNetSim++: A Toolkit for Modeling and Simulation of Distributed Fog Environment. *IEEE Access*, 6:63570–63583.

Quadri, C., Mancuso, V., Marsan, M. A., and Rossi, G. P. (2022). Edge-based platoon control. *Computer Communications*, 181:17–31.

Rahman, G. M. S., Dang, T., and Ahmed, M. (2020). Deep reinforcement learning based computation offloading and resource allocation for low-latency fog radio access networks. *Intelligent and Converged Networks*, 1(3):243–257.

Rashed, A. (2019). Empirical measurements of function placements and executions in a mixed cloud-edge cluster.

Rashed, A. and Rausch, T. (2020). Execution Traces of an MNIST Workflow on a Serverless Edge Testbed.

Rausch, T., Rashed, A., and Dustdar, S. (2021). Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259 – 271.

Reznik, A., Sulisti, A., Artemenko, A., Fang, Y., Frydman, D., Giust, F., Lv, H., Sheikh, S., Yu, Y., and Zheng, Z. (2018). Mec in an enterprise setting: A solution outline. *ETSI, Sophia Antipolis, France, White Paper*, 2(30):1–20.

RIPE Atlas (Accessed Oct 2019). RIPE Atlas Built-in Measurements. `https://atlas.ripe.net/docs/built-in/`.

sar(1) — Linux manual page (Last Modified Aug 2020). sar(1) — Linux manual page. `https://www.man7.org/linux/man-pages/man1/sar.1.html`.

Sarkar, S., Chatterjee, S., and Misra, S. (2018). Assessment of the Suitability of Fog Computing in the Context of Internet of Things. *IEEE Transactions on Cloud Computing*, 6(01):46–59.

Shavitt, Y. and Shir, E. (2005). DIMES: Let the Internet Measure Itself. *ACM SIGCOMM Comput. Commun. Rev.*, 35(5):71–74.

Sherwood, R. and Spring, N. (2006). Touring the Internet in a TCP Sidecar. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 339–344.

Sodagar, I. (2011). The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67.

Sonmez, C., Ozgovde, A., and Ersoy, C. (2018). Edgecloudsim: An environment for performance evaluation of edge computing systems. *Trans. Emerg. Telecommun. Technol.*, 29(11).

Srinivasa, R., Naidu, N. K. S., Maheshwari, S., Bharathi, C., and Hemanth Kumar, A. R. (2019). Minimizing Latency for 5G Multimedia and V2X Applications using Mobile Edge Computing. In *2019 2nd International Conference on Intelligent Communication and Computational Techniques (ICCT)*, pages 213–217.

sysstat - System performance tools for the Linux operating system (Last commit Apr 18 2022). sysstat - System performance tools for the Linux operating system. `https://github.com/sysstat/sysstat`.

Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., and Sabella, D. (2017). On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys and Tutorials*, 19(3):1657–1681.

tc (Traffic Control) (Accessed on: March 2020). tc (Traffic Control). http://man7.org/linux/man-pages/man8/tc.8.html.

Team Cymru (Accessed Aug 2021). Team cymru: IP to ASN mapping. http://www.team-cymru.com/IP-ASN-mapping.html.

The map of the GARR network (Accessed Aug 2021). The map of the GARR network. https://gins.garr.it/xWeathermap/mapgen.php?slice=garrx_top.

The NITOS facility (Last accessed May 2022). The NITOS facility. https://nitlab.inf.uth.gr/NITlab/nitos.

The source code of camotrace (Accessed Aug 2021). The source code of camotrace. https://bitbucket.org/ChiaraCaiazza/camouflagetraceroute/src/master/.

The source code of the energy evaluator (Last updated on 13 May 2021). The source code of the energy evaluator. https://github.com/ChiaraCaiazza/EnergyEvaluator.

The source code of the MECPerf Collection system (Accessed Aug 2021). The source code of the MECPerf Collection system. https://github.com/MECPerf/MECPerf.

The source code of the MECPerf library (Accessed Aug 2021). The source code of the MECPerf library. https://github.com/MECPerf.

The Wireshark Home Page (Accessed Aug 2021). The Wireshark Home Page. https://www.wireshark.org/.

Tim: Le tecnologie abilitanti per l'IoT (Accessed: March 2021). Le tecnologie abilitanti per l'IoT. https://www.gruppotim.it/tit/it/notiziariotecnico/edizioni-2016/n-3-2016/capitolo-4.html.

Toczé, K., Lindqvist, J., and Nadjm-Tehrani, S. (2020). Characterization and modeling of an edge computing mixed reality workload. *Journal of Cloud Computing*, 9(1):1–24.

Toczé, K., Schmitt, N., Kargén, U., Aral, A., and Brandic, I. (2020). Edge workload traces from Aeneas, Julius, and MR- Leo.

Tong, L., Li, Y., and Gao, W. (2016). A hierarchical edge cloud architecture for mobile computing. In *Proc. IEEE INFOCOM '16*, pages 1–9.

Tseng, C., Wang, H., Kuo, F., Ting, K., Chen, H., and Chen, G. (2016). Delay and Power Consumption in LTE/LTE-A DRX Mechanism With Mixed Short and Long Cycles. *IEEE Transactions on Vehicular Technology*, 65(3):1721–1734.

tshark (Accessed 2021). tshark - manual page. `https://www.wireshark.org/docs/man-pages/tshark.html`.

Vermeulen, K., Strowes, S. D., Fourmaux, O., and Friedman, T. (2018). Multilevel MDA-Lite Paris Traceroute. In *Proc. ACM SIGCOMM IMC '18*, pages 29–42. ACM SIGCOMM IMC '18.

Vilela, P. H., Rodrigues, J. J., Solic, P., Saleem, K., and Furtado, V. (2019). Performance evaluation of a Fog-assisted IoT solution for e-Health applications. *Future Gener. Comput. Syst.*, 97:379–386.

Wang, J., Amos, B., Das, A., Pillai, P., Sadeh, N., and Satyanarayanan, M. (2017). A Scalable and Privacy-Aware IoT Service for Live Video Analytics. In *Proceedings of the 8th ACM on Multimedia Systems Conference - MMSys'17*, pages 38–49, New York, New York, USA. ACM Press.

Wang, J., Feng, Z., Chen, Z., George, S., Bala, M., Pillai, P., Yang, S., and Satyanarayanan, M. (2018). Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 159–173.

Xu, X., Li, D., Dai, Z., Li, S., and Chen, X. (2019a). A heuristic offloading method for deep learning edge services in 5g networks. *IEEE Access*, 7:67734–67744.

Xu, Z., Liu, X., Jiang, G., and Tang, B. (2019b). A time-efficient data offloading method with privacy preservation for intelligent sensors in edge computing. *EURASIP Journal on Wireless Communications and Networking*, 2019.

Yan, X., Yang, L., and Wong, B. (2020). Domino: Using network measurements to reduce state machine replication latency in wans. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 351–363, New York, NY, USA. Association for Computing Machinery.

Yang, S., Gong, Z., Ye, K., Wei, Y., Huang, Z., and Huang, Z. (2020). Edgernn: A compact speech recognition network with spatio-temporal features for edge computing. *IEEE Access*, 8:81468–81478.

Yeganeh, B., Durairajan, R., Rejaie, R., and Willinger, W. (2020). A first comparative characterization of multi-cloud connectivity in today's internet. In *International Conference on Passive and Active Network Measurement*, pages 193–210. Springer.

Zahed, M. I. A., Ahmad, I., Habibi, D., and Phung, Q. V. (2020). Green and Secure Computation Offloading for Cache-Enabled IoT Networks. *IEEE Access*, 8:63840–63855.

Zhang, J., Hu, X., Ning, Z., Ngai, E. C. ., Zhou, L., Wei, J., Cheng, J., and Hu, B. (2018). Energy-Latency Tradeoff for Energy-Aware Offloading in Mobile Edge Computing Networks. *IEEE Internet of Things Journal*, 5(4):2633–2645.

Zhang, X., Chen, H., Zhao, Y., Ma, Z., Xu, Y., Huang, H., Yin, H., and Wu, D. O. (2019). Improving Cloud Gaming Experience through Mobile Edge Computing. *IEEE Wireless Communications*, 26(4):178–183.

Zhao, X., Peng, J., Li, Y., and Li, H. (2020). A privacy-preserving computation offloading method based on privacy entropy in multi-access edge computation. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 1016–1021.

Zheng, X. and Cai, Z. (2020). Privacy-preserved data sharing towards multiple parties in industrial iots. *IEEE Journal on Selected Areas in Communications*, 38(5):968–979.

Zhou, L., Xu, H., Tian, H., Gao, Y., Du, L., and Chen, L. (2008). Performance Analysis of Power Saving Mechanism with Adjustable DRX Cycles in 3GPP LTE. In *2008 IEEE 68th Vehicular Technology Conference*, pages 1–5.