Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (32.th cycle)

On the Orchestration of Dynamic Services over Distributed IT Infrastructures

Gabriele Castellano
* * * * * *

Supervisor
Prof. Fulvio Risso

Politecnico di Torino
July 2020

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

. . . . . .

Gabriele Castellano

Turin, July 2020

Summary

The arise of Network Function Virtualization and Software Defined Networking enabled the programmability of the network infrastructure, hence allowing the dynamic deployment of complex services on top of the network. Additionally, Edge Computing extends the Cloud paradigm towards the edge of the network, leading to a highly distributed infrastructure and introducing new players in the service provisioning ecosystem. This transformation leads to enhanced possibilities for service providers, which, thanks to the flexibility and new capabilities offered by the infrastructure below, may realize and deliver a plethora of new applications, such as virtual reality, remote critical tasks, and more. However, managing such a distributed infrastructure and enabling interoperability between the multiple actors involved introduces a series of challenges. A particularly challenging problem is resource management. Since resources (such as computing, networking, and storage) should be partitioned in slices that are allocated for each service, a key component called Orchestrator is often employed to decide on the deployment and the management of each service. However, the optimality of the taken decisions may not match the actual necessity of the services, as each of them may benefit from different allocation strategies and may want to optimize on different parameters and service-specific metrics. Such metrics are often unknown to the orchestrator, which operates at the infrastructure level and based on a one-size-fits-all paradigm. Moreover, mandating the existence of centralized coordination components may not be suitable in a scenario where services are executed on scattered compute nodes, e.g., at the edge of the network, which features arbitrary and dynamic topologies. Finally, since resources are scarce and geographically distributed in different areas, service provisioning may involve multiple providers that should inter-operate coordinating the deployment of applications on top of their clusters. Given these considerations, this thesis investigates new service-centric orchestration paradigms, which cover different aspects of the above problem. A novel Service-Defined Orchestration approach is proposed, which distributes the orchestration task and delegates it to the service providers competent for each application. The problem of service management, utilization, and dissemination in harsh environments is also investigated, by designing highly distributed architectures and algorithms with the aim of transparently enable a suitable service layer on heavily scattered IT infrastructures.

Acknowledgements

First, I would like to thank Tierra Telematics, which primarily supported the work presented in this thesis. In particular, I express my gratitude to Riccardo Loti and Nicola Smaldone, for their help and willingness. Additionally, I would like to thank TIM, and in particular, Antonio Manzalini, for his support and the many fruitful discussions.

A special thanks goes to Flavio Esposito, which was my co-advisor during an important period of my Ph.D. spent abroad at Saint Louis University (MO), USA. With his ideas, opinions, and motivation he significantly contributed to a core piece of my doctorate program.

Finally, I would like to sincerely express my gratitude to my advisor Fulvio Risso, for his important support in all the activities where I was involved during my Ph.D. period. His patience, dedication, and genuine motivation, as well as all the opportunities he provided me, have made my Ph.D. experience great.

"Considerate la vostra semenza:
Fatti non foste a viver come bruti,
Ma per seguir virtute e canoscienza."

— Dante Alighieri,
La Divina Commedia,
Inferno, XXVI, 118-120.

Contents

List of Tables XI

List of Figures XII

1 Introduction 1

2 Capability-based Orchestration across multi-technological infrastructures 5
2.1 Introduction 5
2.2 Multi-domain orchestration 7
2.2.1 Exported domain information 8
2.2.2 Overarching orchestrator 9
2.3 OpenStack Domain Orchestrator 11
2.3.1 Domain overview 11
2.3.2 Discovering and exporting domain information 12
2.3.3 Deploying service graphs 12
2.3.4 Limitations 13
2.4 SDN Domain Orchestrator 14
2.4.1 Exploiting SDN applications as NFs in service graphs 14
2.4.2 Discovering and exporting domain information 16
2.4.3 Deploying service graphs 16
2.5 Experimental Results 17
2.6 Related Work 19
2.7 Conclusion 20

3 Toward a Disaggregated Edge Model: Business Consideration and Optimization Opportunities 21
3.1 Introduction 21
3.2 State-of-the-Art in Standardization Bodies 23
3.3 Toward a new Model for MEC 25
3.3.1 Definitions 25
3.3.2 Interaction models 26

3.3.3 Interface standardization . . . 28
3.3.4 Business models . . . 29
3.3.5 Disaggregated MEC . . . 30
3.4 Use Cases . . . 32
3.4.1 Everywhere in the city . . . 32
3.4.2 Enterprise connectivity services . . . 33
3.5 Interactions Optimization . . . 33
3.5.1 Interaction between PaaS and SaaS . . . 33
3.5.2 Interaction among PaaS providers . . . 35
3.5.3 Interaction between PaaS and IaaS . . . 35
3.5.4 Interaction among IaaS providers . . . 36
3.6 Conclusion . . . 37

4 Service-Defined Orchestration of Heterogeneous Applications in Cloud/Edge Platforms 39
4.1 Introduction . . . 39
4.2 Related Work . . . 41
4.3 Overall Architecture . . . 42
4.3.1 Service Plane . . . 42
4.3.2 Orchestration Plane . . . 43
4.3.3 Infrastructure Plane . . . 44
4.4 Service-Defined Orchestrator . . . 45
4.4.1 Orchestrator Behavioral Model (OBM) . . . 45
4.4.2 SDO Architecture . . . 47
4.4.3 A practical use case: the video streaming application . . . 49
4.5 Experimental Results . . . 50
4.6 Conclusion . . . 53

5 A Distributed Orchestration Algorithm for Edge Computing Resources with Guarantees 55
5.1 Introduction . . . 56
5.2 Related Work . . . 57
5.3 Problem Definition and Modeling . . . 58
5.4 Distributed Resource AssiGnment and OrchestratioN (DRAGON) . . . 60
5.4.1 DRAGON Overview . . . 60
5.4.2 Orchestration Phase . . . 62
5.4.3 Agreement Phase . . . 66
5.4.4 Recommendations on the score function . . . 67
5.5 Convergence and Performance Guarantees . . . 68
5.6 Experimental Results . . . 70
5.7 Conclusion . . . 74

6 Configuring Services in Highly Modular Environments: a Model-Based Solution 77
6.1 Introduction . . . 78
6.2 Related Works . . . 79
6.3 Use Cases . . . 81
6.4 Architecture . . . 83
6.4.1 A data model for SDNApps . . . 83
6.4.2 Communication infrastructure . . . 85
6.4.3 To-Yang Agent . . . 87
6.5 Mapping Algorithms . . . 93
6.5.1 Structures and Function Notations . . . 93
6.5.2 Algorithms Description . . . 95
6.6 Experimental Results . . . 99
6.6.1 ToY Agent Evaluation . . . 99
6.6.2 Use Cases Evaluation . . . 103
6.7 Discussion . . . 105
6.8 Conclusion . . . 106

7 Service management on disrupted infrastructures: Æther 107
7.1 Introduction . . . 108
7.2 Related Work . . . 109
7.3 Æther Architecture . . . 112
7.3.1 The DTN Daemon . . . 113
7.3.2 Auspex Routing . . . 114
7.3.3 Service Discovery . . . 116
7.3.4 Virtual Services . . . 123
7.4 Æther Framework . . . 124
7.4.1 Protocol Gateways . . . 124
7.4.2 Framework APIs . . . 127
7.5 Experimental Results . . . 128
7.5.1 Prototype - on Physical Devices . . . 128
7.5.2 MQTT Gateway - on Virtualized Environment . . . 130
7.5.3 Service Discovery - on Virtualized Environment . . . 132
7.5.4 Auspex Routing - on Emulated Environment . . . 134
7.6 Conclusion . . . 139

8 Conclusions 141

A Author publications 145

Bibliography 147

List of Tables

6.1 Deriving the YANG path from the YANG data-model. 85
6.2 Excerpt of PMT, referred to the data model of Figure 6.3 and to the object tree shown in Figure 6.5. 89
6.3 Formal notation for data structures used in the algorithms description. 94
6.4 Delay introduced by the ToY Agent for three different use cases compared with direct ah-hoc access (average times taken over thousand samples are shown). 105
7.1 Mandatory properties for predefined categories. 119
7.2 Service Discovery Messages. 121
7.3 Throughput on the Bluetooth CLA. 129
7.4 RTT (ms) over the BT-CLA compared with TCP/IP over WiFi ad-hoc. 129
7.5 Server-side computation required to process the content of the Net-G5 dashboard to convey it into the DTN. 130
7.6 Average number of messages generated in one hour, divided for destination area. 136
7.7 Time needed to deliver 95% of the bundles. 137
7.8 Latency constraint imposed on generated messages, based on the destination area. The table shows the percentage of generated bundles that are affected by latency constraints and the ranges within which the constraint values are uniformly selected. 138

List of Figures

2.1 Service graph deployment in a multi-domain infrastructure. 6
2.2 Placement and splitting of a service graph: sub-graphs are interconnected through traffic steering technologies exported by each domain. 9
2.3 Inter-domain traffic steering based on SAPs parameters. 10
2.4 Interactions of the OS-DO with the components of an OpenStack domain. 12
2.5 Service graph deployment in an SDN domain. 15
2.6 Deployment of a service graph in two different scenarios. In case (b), the SDN-DO provides the capability of running a NAT (as SDN application), thus the Overarching Orchestrator opts for better optimization. Performances are compared at the bottom. 18
2.7 Time required to deploy the service graph, highlighting all the main operations and components (horizontal axis not in scale — CPE domain not shown). 19
3.1 Layered model for telco 5G actors, inspired by ETSI White Paper [133]. 22
3.2 Overall disaggregated edge architecture with business interactions. . 26
3.3 Disaggregated MEC architecture. 31
4.1 Overall distributed and service-defined orchestration architecture. . 43
4.2 Interactions between an SDO and *(i)* the infrastructure controller and *(ii)* components of the managed application. 44
4.3 Overall architecture of the Service-Defined Orchestrator. 48
4.4 Frame rate over time for a video streaming application in a resource constrained situation. A Service-Defined Orchestrator may effectively mitigate the QoE deterioration being able to explore alternative solutions. 51
4.5 Evaluation of a CDN cache provisioning application comparing different placement strategies: (a) miss rate over time varying the geographical users distribution; (b) distribution of measured miss rate varying the number of concurrent applications. 52

4.6 Evaluation of a mobile gaming application for different deployment strategies: (a) QoE over time perceived by a user moving in different areas; (b) QoE distribution varying the number of concurrent applications. 53
5.1 Example of false winners after an election routine: SDO #2 prevents #1 to allocate needed resources on node 1, although #2 cannot be deployed, since it lost elections on node 2. 65
5.2 Convergence evaluation of DRAGON for different system policies. . 71
5.3 DRAGON convergence evaluation on large scale simulation varying the number of concurrent applications and available hosting nodes, where resource allocation requests are randomly performed over time. 72
5.4 Performance evaluation of DRAGON comparing (a) different system utilities and (b) DRAGON solutions against *(i)* three one-size fits-all common approaches and *(ii)* a reference solution obtained running a centralized solver. 73
6.1 Use case example: an Intrusion Prevention System service (*ServiceApp*) requiring the access to the run-time state of multiple SDN applications (*SDNApps*). 82
6.2 Overall software architecture. ServiceApps (on top) relies on a northbound interface (composed by the Pub/Sub message bus and on the Rest-based channel) to perform `get`, `set` and `subscribe` operations on the run-time state of SDNApps. These are accessible through this northbound thanks to the YANG-based mapping performed by ToY Agents. 83
6.3 On the left, a YANG data model for the NAT application. On the right, an example of YANG path (the one of the *"private interface"* container highlighted in the model) and an instance node showing a possible run-time value. 84
6.4 Example of messages sent on the message bus (in the left) and on the REST-based channel (in the right), defined according to the data model of Figure 6.3. 86
6.5 From run-time variables (in the left) to object paths (in the right). . 88
6.6 Architecture of the ToY Agent. 91
6.7 Steps required to include the ToY Agent in an existing application. 92
6.8 (a) Partial times needed to *(i)* map YANG paths into Object paths and *(ii)* fetch an object reference starting from its path, for different depth levels. (bc) `get_node` (b) and `set_node` (c) total times for different depth levels; values are compared with direct ad-hoc access. Upper and lower quartils are plotted along with errorbars. 100

6.9 (a) Time overhead introduced by the ToY Agent for read and write operations, at different depth levels. (b) Additional lines of code needed to enable read and write access to SDNApp variables for different depth levels. (c) ToY Agent CPU consumption and notification delay for different polling periods (upper and lower quartiles are plotted along with error bars). 101
6.10 Use case workflows for (a) IPS (b) Service Migration and (c) Service Orchestration scenarios. The figures detail the list of operations that ServiceApps execute on SDNApps. 103
6.11 Overall percentage overhead introduced by the ToY Agent for three different use cases. 105
7.1 Opportunistic connections in a disrupted environment. 109
7.2 Overview of the Æther disrupted communication scenario. Applications on board of scattered devices can interact through the disrupted network environment using high-level service-oriented facilities. . . . 112
7.3 Æther node architecture. 113
7.4 Performance of main DTN routing algorithms compared. 115
7.5 Service Description Model (SDM). 118
7.6 Default categories pre-defined in Æther service description. Other custom categories can be defined. 119
7.7 The Service Discovery module. 122
7.8 Workflow of a Virtual Service Request. The VS Handler first schedules the best available option according to the request parameters, binding a VS Worker on the selected remote service. Upon receiving an event that makes the currently bound service inadequate on the delay requirements, the Handler repeats scheduling and binding, making the worker switch to a different remote service (communication channel changes from *(a)* to *(b)*). 123
7.9 MQTT gateway in (a) upstream and (b) downstream communication. 125
7.10 (a) Distribution over time of delivered bundles, for different connection time intervals. (b) Bundle delivery ratio varying the probability of inter-device connections, comparing different connection time intervals (1, 2, 4 and 6 seconds). All values refer only to bundles already expired at the end of the run. 130
7.11 (c) Average number of bundles stored on each device during the simulation for different probabilities of inter-device connections, with a bundle lifetime of 5 minutes. (d) Average storage overhead varying the probability of inter-device connections and comparing different bundle lifetimes (1, 5 and 10 minutes). 132

7.12 Time needed for a discovery request to get a response. (a) Normalized probability mass for the case with 30 nodes, 20% of connection probability and 50% of directories. (bcd) Comparisons in different network setup: (b) 15 nodes and 50% of directories varying connection probability; (b) 20% of connection probability and 50% of directories varying number of nodes; (b) 25 nodes and 20% of connection probability varying percentage of directories. 133
7.13 (a) Virtual Service binding delay comparing different number of nodes. (b) Throughput overhead of the virtual service in a connected topology. 134
7.14 Routing experiments comparing Epidemic, MaxProp Vanilla, and Auspex over MaxProp, on two setups with a different number of nodes. Plots show delivery time (a-d, f) and delivery ratio (e, g), distinguishing for destination area, and the overhead of Epidemic (h) on both the setups (scale is logarithmic). 135

Chapter 1

Introduction

The rise of virtualization technologies has drastically changed the way services are deployed and delivered in new generation computing and networking infrastructures. In IT, Cloud Computing decoupled the role of infrastructure management from the provisioning of final services to end customers, leading to the rise of a variety of infrastructure providers (e.g., Amazon Web Services, Microsoft Azure) and diverse platform frameworks (e.g., Google Cloud Platform) that enabled the proliferation of a variety of applications and service providers (e.g., Netflix).

With new trends in IT applications, such as IoT, social networks, industrial control loops, and more, data is increasingly produced at the edge of the network: "things" in the network are estimated to double in a few years [8], and they mostly produce data that is intended to be consumed at the edge; every single minute, users upload more than 500 hours of content on the YouTube servers, and tens of thousands of new pictures on Instagram [170, 78]. This change in the data production/consumption flow makes cloud not efficient anymore for its processing, because of the physical limitation of the Internet. For this reasons, as more and more new applications require particular infrastructure capabilities and constraints, such as low latency and high bandwidth (e.g., virtual reality, autonomous driving), the virtualization paradigm has extended its domain from the IT area to the one of telecommunication operators, which, given their position, are the best candidates to manage and provide highly distributed computing facilities at the very edge of the network (Edge Computing). Telcos are pursuing a digital transformation that brings flexibility and programmability on their network infrastructures, through the introduction of novel paradigms such as Network Function Virtualization, that is, networking middleboxes are implemented in software and executed within isolated VMs or containers, and Software-Defined Networking, which decouples the control logic from the physical network devices.

The availability of cloud-like facilities within the boundaries of the telcos' networks enables a large variety of scenarios. Users can benefit from fast offloading for computation-intensive tasks such as augmented/assisted reality or face/speech

recognition [174]; vehicle-to-everything (V2X) interactions can be extended into the mobile network, by running roadside applications within the edge servers and helping in enabling autonomous driving [105]; telcos may even benefit from the presence of computation power in their networks deploying smarter optimization services, e.g., for radio-backhaul coordination, analytic tasks, and more [104].

In such flexible and dynamic environment, automated configuration, management, and coordination become crucial. Such tasks are commonly referred to as "Orchestration" [128], whose overall aim is to enable flexible and dynamic deployment of complex services. The decisions that the Orchestrator takes mainly involve how to optimally slice the infrastructure continuum of resources, which may encompass computing, networking, storage, and even more heterogeneous IoT devices.

Challenges introduced by the new Edge paradigm are manifold. Since resources are scarce and geographically distributed in different areas, multiple providers, even playing different roles, should inter-operate, coordinating the deployment of services on top of their clusters. In doing so, it is important to exploit any capability available on the infrastructure, a task that is often not trivial due to the heterogeneity of such an unconventional environment. Coordination could not be demanded to centralized components, as services are executed on scattered clusters and multiple providers are involved. Additionally, different services may benefit from different orchestration strategies and predefined one-size-fits-all approaches may not match the actual necessity of the application, whose metrics are only known to the service provider of competence. In this work, we investigate novel paradigms for service management and orchestration that could overcome these challenges.

In particular, Chapter 2 analyzes the problems faced by a new generation Edge provider while deploying and orchestrating services on top of a multi-technological infrastructure. In this regard, we propose a capability-based solution, which aims to spot and exploit any facility offered by the cluster of resources below. This ensures flexibility and better optimization possibilities in service deployment.

In Chapter 3 we highlight the disadvantages and limitations of a centralized and monolithic orchestration model in Edge Computing. Taking as reference the Multi-access Edge Computing (MEC) architecture [86], we identify the new actors involved in service provisioning at the edge of the network and their mutual interactions. Contextually, we propose a possible open and disaggregated model for the business interactions between these new edge actors and some preliminary considerations on their algorithmic optimization.

Chapter 4 overcomes the limitations of one-size-fits-all orchestration approaches by proposing a novel Service-Defined Orchestration (SDO) paradigm, where the orchestration task is distributed among the service providers. Using a formalized declarative language, they may define custom strategies to manage their own services, optimizing on metrics that are service-specific and therefore unknown to a traditional orchestrator that operates at the infrastructure layer. To coordinate the coexistence of such a variety of service providers orchestrating resources

on the same shared infrastructure, we design DRAGON, a Distributed Resource AssiGnment and OrchestratioN optimization algorithm that is then described in Chapter 5. Through a fully distributed decision process, DRAGON leads a pool of Service-Defined Orchestrators to reach a dynamic agreement on how resources should be (temporarily) partitioned among them, providing guarantees on both convergence time and performance.

The solutions adopted in Edge computing lead to a highly modular and distributed infrastructure that is populated with arbitrary new facilities, generally exploited by service providers to compose the final application. In Chapter 6 we design a configuration layer that enables interoperability between the existing actors, by mean of a model-based approach that facilitates service composition and enables monitoring and tuning of arbitrary components.

Chapter 7 ultimately overviews the applicability of new generation service facilities on highly scattered infrastructures, with particular focus to the Industrial Internet of Things, where services operate without relying on fixed networks and exploit opportunistic connections between fleets of vehicles. To enable IoT services to transparently operate on such environments we propose Æther, a service-oriented communication system that provides service management features and service-aware routing optimization on disrupted networks, with advantages over state-of-the-art algorithms.

Part of the work described in this dissertation has been previously published in other scientific manuscripts authored by the candidate. A list of the relevant publications is available in Appendix A.

Chapter 2

Capability-based Orchestration across multi-technological infrastructures

This chapter presents an open-source orchestration framework that deploys end-to-end services across multi-technological domains, with a focus on OpenStack-managed clusters and Software-Defined Networks controlled either by ONOS or OpenDaylight. The proposed approach improves the existing work in two directions. First, it better exploits capabilities available in the infrastructure below, by taking benefit from the different available technologies. This allows, for instance, the usage of SDN domains not only to implement traffic steering but also to execute selected network functions (e.g., NAT). Second, this work can deploy a complex service by partitioning the original service graph into multiple subgraphs, each one instantiated in a different domain, dynamically connected by means of traffic steering rules and parameters negotiated at run-time.

2.1 Introduction

New generation infrastructure providers often manage a multitude of heterogeneous technological domains. In such scenarios, end-to-end service deployment of Network Functions (NFs) usually involves two levels of orchestrators [148, 42] (Figure 2.1), handling the deployment process in a hierarchical fashion [58, 167]. An *Overarching Orchestrator* sits on top of a multi-technology infrastructure and

Part of the work presented in this chapter has been first published in [26] and [41]. This work is also partially described in the Ph.D. dissertation of Roberto Bonafiglia [24], who collaborated in its study.

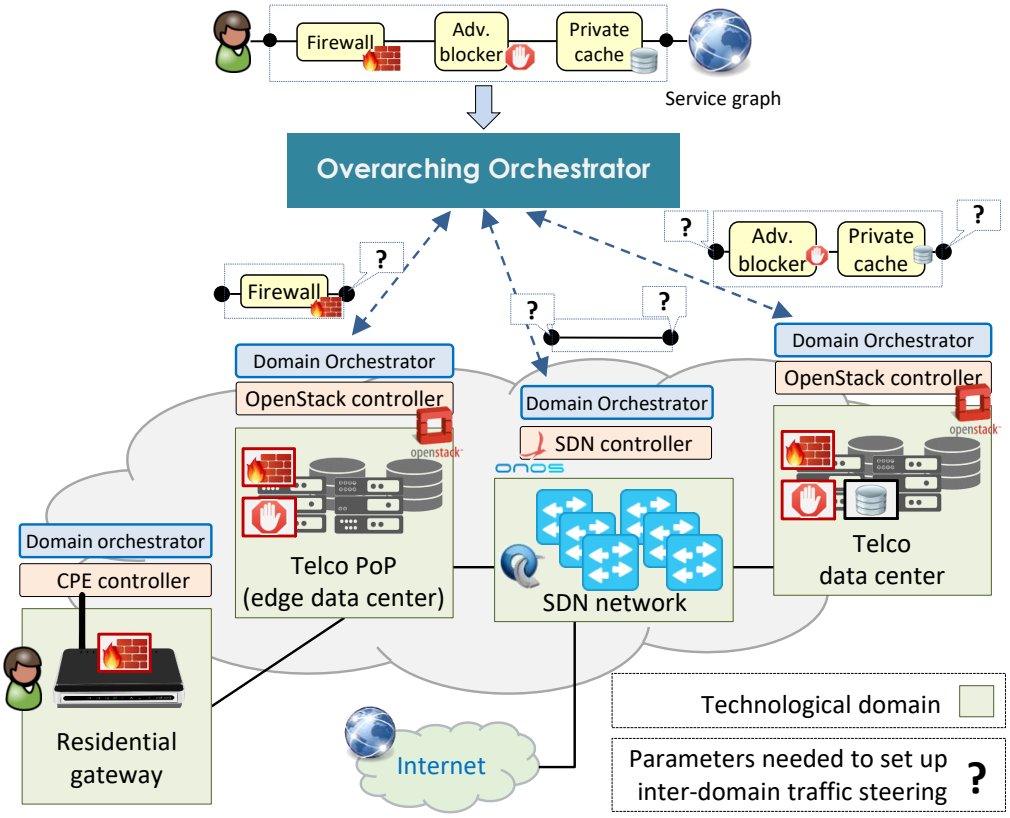


Figure 2.1: Service graph deployment in a multi-domain infrastructure.

receives service deployment requests in the form of service graphs, which define the NFs to be deployed and their interconnections. This component is responsible for *(i)* selecting the domains to involve in the service deployment (e.g., where NFs have to be executed), *(ii)* setup the network parameters for the proper traffic steering between different infrastructure domains, and *(iii)* creating the service *subgraphs* to be instantiated in the selected domains.

At a lower layer, a set of *Domain Orchestrators* manage each a specific technological domain and interact with the given infrastructure controller (e.g., the OpenStack [142] cloud toolkit in data centers, the ONOS [16] or OpenDaylight (ODL) [108] controller in SDN networks) to actually instantiate the service subgraphs in the underlying infrastructure. In addition, Domain Orchestrators export a summary of the computing and networking characteristics of their domains, used by the Overarching Orchestrator to execute its own tasks. This architecture simplifies the integration of existing infrastructure controllers in the orchestration framework, as any possible missing feature is implemented in the Domain Orchestrators themselves, while the infrastructure controllers are kept unchanged.

Existing orchestration frameworks (e.g., [31, 100]) present the following limitations. First, they exploit SDN domains only to create network paths, neglecting the

fact that modern SDN controllers can actually host many applications (e.g., firewall, NAT) that program the underlying network devices according to their own logic. Second, little importance is given to the problem of automatically configuring the inter-domain traffic steering to interconnect portions of the service graph deployed on different domains. For instance, this would require to properly characterize subgraphs endpoints (called Service Access Point, or SAPs) with the proper network parameters, thus replacing the question marks in the subgraphs shown in Figure 2.1 with the proper information such as VLAN IDs, GRE keys and more, based on the capabilities of the underlying infrastructure.

To overcomes these limitations, we propose a novel capability-based orchestration approach that *(i)* can transparently instantiate NFs wherever a suitable implementation is available (e.g., either on cloud computing or SDN domains), and *(ii)* enables the Overarching Orchestrator to enrich the service subgraphs with the information needed for Domain Orchestrators to automatically set up the inter-domain traffic steering.

In our approach, the Overarching Orchestrator executes its own tasks based on domain capabilities, which are exported from a functional rather than a technological point of view. This enables the deployment of NFs not only using traditional VMs or containers on data centers or edge clusters, but also in domains where such functions are implemented through more heterogeneous technologies (e.g., an SDN enabled backbone or a Customer Premise Equipment).

We first present the overall approach and the architecture of the Overarching Orchestrator, then detail the architecture and implementation of two Domain Orchestrators that deploy service graphs in vanilla SDN-based and OpenStack-based domains, thus in Software-Defined Networks and in Cloud/Edge clusters. Notably, other domains can be integrated into our orchestration framework as well (e.g., Customer Premise Equipment), provided that the proper Domain Orchestrator is implemented and run on top of the companion infrastructure controller.

The remainder of this chapter is structured as follows. Section 2.2 describes the information exported by Domain Orchestrators and shows how this is used by the Overarching Orchestrator. Sections 2.3 and 2.4 detail respectively the OpenStack and the SDN Domain Orchestrators. In Section 2.5 we provide and discuss the experimental results, while Section 2.6 positions our contribution with respect to the existing work. Finally, Section 2.7 concludes the paper.

2.2 Multi-domain orchestration

The Overarching Orchestrator receives from each Domain Orchestrator below the characteristics of the portion of infrastructure under its responsibility, such as its capabilities and availability in terms of computing and networking. When a service deployment request is received, the Overarching Orchestrator *(i)* selects

the domains to involve in the service deployment, *(ii)* for each of these, creates the proper service subgraph based on its capabilities, and *(iii)* pushes the resulting subgraphs to the selected Domain Orchestrators (Figure 2.1). In this section, we first present *what* and *how* is exported by Domain Orchestrators to describe the infrastructure below. Then, we describe the operations carried out by the Overarching Orchestrator to deploy a service graph.

2.2.1 Exported domain information

Each Domain Orchestrator exports a summary of the available computing and networking capabilities according to a specific data model (available at [1]) that has been derived from the YANG [23] templates defined by OpenConfig [121].

From the point of view of computing, each domain exports the capability to execute specific NFs. Examples of NFs include firewall and NAT, possibly with some attributes such as the number of ports, support for IPv4 or IPv6, and more. Additional metadata provide information on the current status of the NF within the domain, such as any delay needed to fetch it from a repository, whether local resources are currently enough to allow its execution, expected boot time, and more. The description of each NF is in turn associated with its own YANG data model and is used by the Overarching Orchestrator to evaluate scheduling possibilities. An NF can be a software bundle available in the ONOS controller in case of an SDN domain, a specific VM image on the VM repository in a data center domain, or even a hardware module in a CPE. Information on the actual implementation of the NFs is not advertised from the Domain Orchestrators. Thus the Overarching Orchestrator is able to schedule portions of graphs on any domain advertising the proper capabilities, regardless of the nature of the involved domains.

From the point of view of networking, the domain is described as a "big-switch", with a set of *boundary interfaces*, whose attributes are used by the Overarching Orchestrator to properly set up inter-domain traffic steering and provide the needed instruction to the two domains that terminate the connection. First, the Domain Orchestrator advertises whether the selected boundary interface is directly connected with another domain (and, if so, who), with an access network (that represents the entry point for the traffic into the operator infrastructure), or to a WAN. Second, it advertises a set of *inter-domain traffic steering technologies*, which indicate the ability of the domain to classify incoming traffic based on specific patterns (e.g., VLAN ID, GRE tunnel key), and modify outgoing traffic in the same way (e.g., send packets as encapsulated in a specific tunnel, or tagged with a given VLAN ID, and more). Each inter-domain traffic steering technology is then associated with the set of *labels* (e.g., VLAN ID, GRE key) that are still available and can then be exploited to identify new types of traffic. Finally, other parameters associated with interfaces are inherited from the OpenConfig model, e.g., their Ethernet/IP configuration.

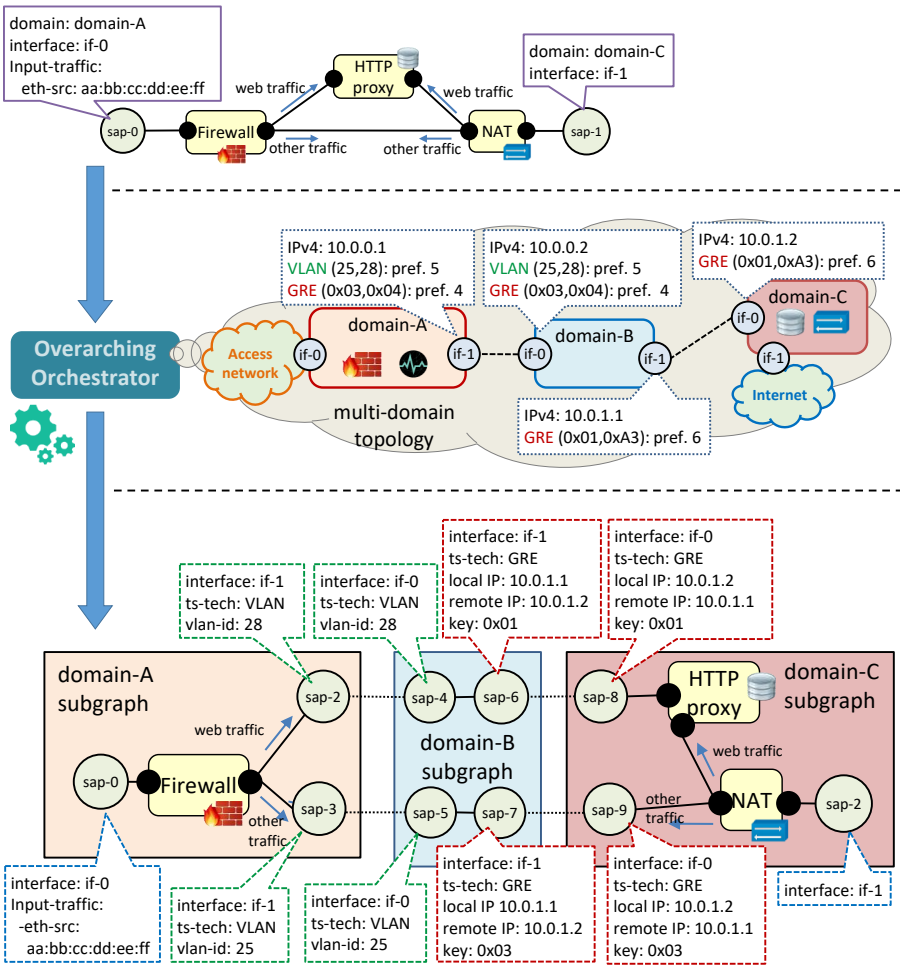


Figure 2.2: Placement and splitting of a service graph: sub-graphs are interconnected through traffic steering technologies exported by each domain.

2.2.2 Overarching orchestrator

The overarching orchestrator deploys service graphs that consist of service access points (*SAPs*), NFs and (logical) links, as shown at the top of Figure 2.2. A SAP represents an entry/exit point of traffic into/from the service graph; it may be associated with a specific traffic classifier (i.e., a selector of packets that have to enter in the service graph) and with a specific domain boundary interface that corresponds, e.g., to the entry point of those packets in the multi-domain infrastructure. Links can be enriched with constraints on the traffic that has to transit on that specific connection.

While scheduling each NF needed by the service graph, the Overarching Orchestrator takes into account any domain that advertised a compatible capability, regardless of the nature of such domain. It can be a data center implementing the

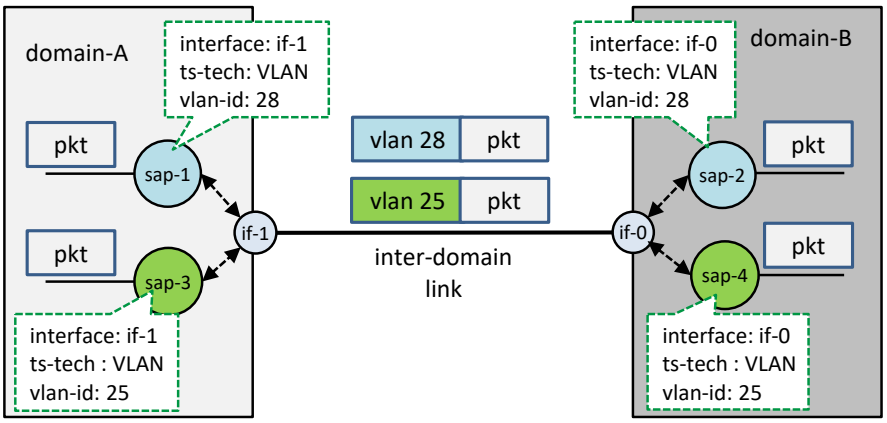


Figure 2.3: Inter-domain traffic steering based on SAPs parameters.

NF as a VM, or an SDN domain implementing it through an application executed in the controller. In this phase decisions are taken only in accordance with known information on the status of functional capabilities exported by the domains below (e.g., which domains notified to be in condition to run a given NF within the required constraints). Deployment details that concern the actual technology used to run the NFs, such as precise decisions on resource management, are then handled by the particular Domain Orchestrator (Sections 2.3, 2.4), e.g. optimizing task scheduling in a data center domain. This allows a certain level of scalability and flexibility in the Overarching Orchestrator it jointly managing not a-priori known domain types. Note that some SAPs may already be associated with specific domain interfaces (e.g., customer access network), and then are constrained to be scheduled on that specific domain. As shown in Figure 2.2, once the domains involved in the deployement of NFs have been selected, the Overarching Orchestrator creates one subgraph per each of them. The subgraph includes the NFs and SAPs assigned to that domain and, possibly, new SAPs not present in the "original" service graph. These are originated whenever a link between two NFs (or SAPs) is split across different domains. Notably, some domains (e.g., `domain-B` in Figure 2.2) may only be used to create network paths between NFs/SAPs instantiated somewhere else; in such case, a service subgraph is generated for these domains as well, which just include links between (new) SAPs.

In order to recreate the connection described in the service graph, each of the two SAP generated upon the split of a link is enriched with the proper networking parameters, thus allowing Domain Orchestrators to set up the proper communication tunnel. To this purpose, as shown in Figure 2.2, new SAPs are associated with specific domain boundary interfaces, and a specific inter-domain traffic steering technology and label (e.g., GRE tunnel based on the key `0x01`), chosing a combination that is available in both the interfaces to be connected.

As shown in Figure 2.3, this information is then used by Domain Orchestrators,

which configure the infrastructure below so that packets sent through a specific SAP are properly manipulated (e.g., encapsulated in a specific GRE tunnel) before being sent towards the next domain through a specific interface. Similarly, this information enables to recognize received traffic as entering from a given SAP (e.g., `domain-B` in Figure 2.2 receives, from interface `if-0`, traffic for `sap4` and `sap5`). Notably, packets should be tagged/encapsulated just before being sent out of the domain, while the tag/encapsulation should be removed just after the packet is classified in the next domain.

2.3 OpenStack Domain Orchestrator

This section describes details on our implementation of the OpenStack Domain Orchestrator (OS-DO), which that enablesinstantiating service (sub)graphs on cloud-based clusters managed by a vanilla OpenStack controller. Our implementation have been tested on Mikata release. Source code is available at [3].

2.3.1 Domain overview

As shown in Figure 2.4, an OpenStack domain includes the OpenStack infrastructure controller that manages: *(i)* high-volume servers, called *compute nodes*, hosting VMs (or containers); *(ii)* a network node hosting the basic networking services and representing the entry/exit point for traffic in/from the data center network; *(iii)* an SDN controller; *(iv)* other helper services such as the VM images repository. Each compute node includes two OpenFlow-enabled virtual switch instances (we used Open vSwitch [126]): `br-int`, connected to all the VM ports running in that specific compute node, and `br-ex`, connected to the physical ports of the server. Servers are connected through a physical network not necessarily managed by the OpenStack controller.

Most interactions of the OS-DO are with the Nova and Neutron components. Nova takes care of handling the lifecycle of VMs (e.g., start/stop) in the compute nodes, while Neutron, is responsible for managing the virtual switches (e.g., create ports, instantiate flow rules), also through the SDN controller. Particularly, Neutron programs the forwarding tables of the virtual switches to create *virtual networks* between VMs ports, which implement a broadcast LAN and may include basic services (e.g., DHCP and routing) deployed on the network node. Management of the boundary interfaces and their interaction with such virtual networks is conducted through the SDN Controller.

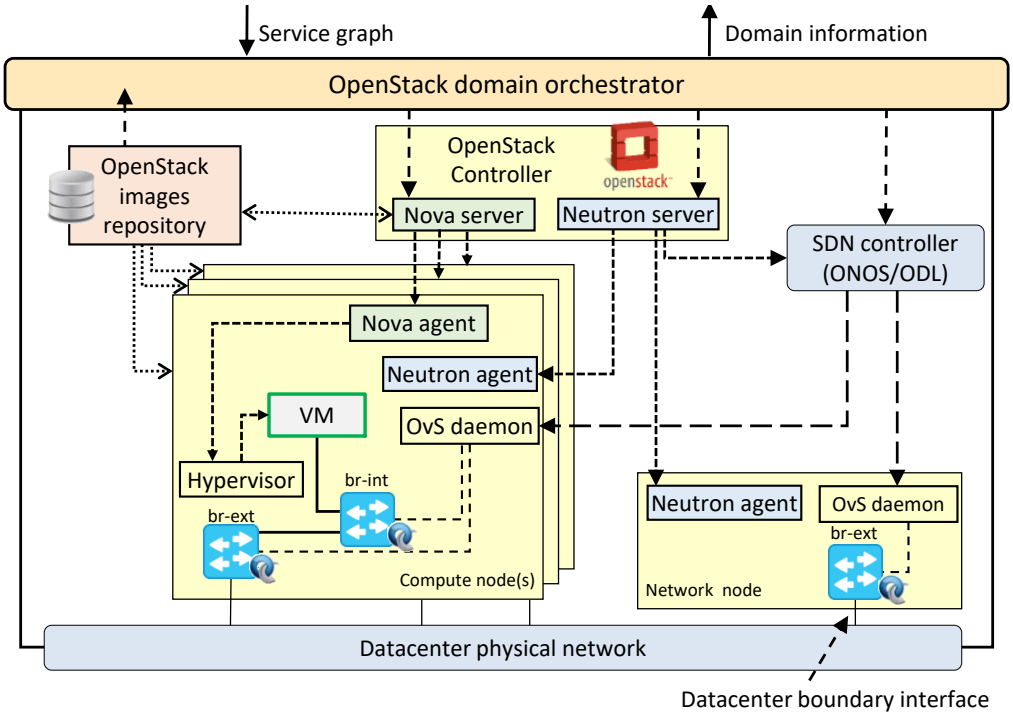


Figure 2.4: Interactions of the OS-DO with the components of an OpenStack domain.

2.3.2 Discovering and exporting domain information

One of the tasks of the OS-DO is to advertise a summary of the computing and networking characteristics of the underlying domain, which includes the supported NFs (taken from the OpenStack images repository) and information about the boundary interfaces. Boundary interfaces are virtual/physical interfaces of the network node(s), which are responsible for connecting the OpenStack domain to the rest of the world and hence handling incoming and outgoing traffic (Figure 2.4).

The list of boundary interfaces and the associated parameters (e.g., neighbor domains, available inter-domain traffic steering technologies/labels, etc.) is loaded at the system bootstrapping, and exported both immediately and whenever a change is detected (e.g., an inter-domain traffic steering technology cannot be used anymore, an interface has been shut down).

2.3.3 Deploying service graphs

When receiving a service (sub)graph to be deployed, the OS-DO first checks that the service satisfies specific constraints (that depend on the limitations described in Section 2.3.4), and that the required NFs and inter-domain traffic steering parameters are available.

If the graph is valid, the OS-DO interacts with Neutron to define the NFs ports

and create one virtual network for each link of the service graph; each virtual network will then be attached to the two NFs ports that are supposed to be connected by the graph link. In case of links between a NF port and a SAP, the virtual network is first connected only to the port of the NF, since Neutron is not able to attach domain boundary interfaces to such a network. At this point, the OS-DO interacts with Nova in order to start the NFs; Neutron automatically creates the NF ports declared before, connects them to the `br-int` switch in the compute node(s) where NFs are deployed, and finally instantiates the flow rules needed to implement the required virtual networks.

Then, the OS-DO creates all the links connecting a NF with a SAP and the inter-domain traffic steering by interacting with the SDN controller (ODL in our case). This is necessary because Neutron offers limited possibilities to control the inbound/outbound traffic of the data center (e.g., only through IP addresses), which are not suited to set up the inter-domain traffic steering. To create the link between a NF port and a SAP (which is associated with a domain boundary interface in the network node, and with inter-domain traffic steering information), the OS-DO fetches from ODL the `br-int` switch connected to the NF. Then, it creates a GRE tunnel between this virtual switch and the network node (that is where boundary interfaces are connected), and sets up the flow rules that actually create the connection. At this point, the OS-DO inserts in the network node also the flow rules needed to properly tag/encapsulate outgoing traffic and to classify incoming packets, as required by the inter-domain traffic steering parameters associated with the SAPs.

2.3.4 Limitations

Vanilla OpenStack does not support either complex graphs that require to split the traffic between different NFs (e.g., the web traffic exiting from a firewall has to go to the HTTP proxy, while the rest goes directly to the Internet, as shown in the graph of Figure 2.2), nor asymmetric graphs (e.g., traffic exiting from the firewall goes to the HTTP proxy, but not vice versa). In fact, since Neutron only connects VM ports to virtual LANs, it does not provide the possibility to create end-to-end flow rules. Thus, in our implementation we implemented links by mean of such virtual LANs, resulting in the impossibility to finely split network traffic and to have asymmetric connections. Overcoming this limitation requires a set of deep modifications to OpenStack, as authors show in [101], resulting in the impossibility to rely on vanilla controllers.

An other problem is that, by default, OpenStack assumes that any vNIC is associated to an IP / MAC address, hence it checks that the traffic exiting from that port has a source MAC address that is compliant with the MAC address of thet vNIC. However, this assumption does not hold in case a transparent network function is deployed (e.g., a transparent firewall), which sends out traffic generated

by other components and therefore featuring source MAC address not compliant with the vNIC. Since it is impossible to know whether a VNF operates in transparent mode, when creating any NF port in Neutron the OS-DO also configures the VM so that such checks are disabled on these ports.

2.4 SDN Domain Orchestrator

This section details our SDN Domain Orchestrator (SDN-DO) [5] that sits on top of a network domain consisting of OpenFlow switches under the responsibility of an SDN controller. It allows an Overarching Orchestrator to instantiate service graphs that include both NFs and links, thus mimicking a typical compute domain. The rationale behind this idea is that widespread SDN controllers can do much more than just steer the traffic between two boundary interfaces. In fact, acting as middle-layer between the control and the data plane, they expose a northbound interface that specific software bundles (a.k.a. SDN applications) can exploit to deploy advanced flow rules (e.g., OpenFlow) in the devices below. In principle, the dynamic deployment of these flow rules may allow to even implement simple network functions (NFs) such as NAT, stateless firewalls, and more.

As shown in Figure 2.5, the SDN-DO executes its tasks (e.g., retrieves the list of NFs to be exported, implements the received service subgraph) by interacting with the SDN controller through a specific driver, which exploits the vanilla REST API exported by the controller itself. At the time of writing, a complete driver for ONOS (Falcon, Goldeneye, Hummingbird and Ibis releases) has been developed. A partial driver for OpenDaylight (Hydrogen, Helium and Lithium releases) is also available, but it lacks the support for managing NFs.

2.4.1 Exploiting SDN applications as NFs in service graphs

The proposed SDN-DO exploits the possibility offered by widespread SDN controllers to dynamically deploy and run applications in the form of software bundles.

However, the following main differences exist between NFs implemented as SDN applications, and NFs implemented as VMs. First, SDN applications usually just process the first packet(s) of a flow, then install specific rules in the underlying switches so that the next packets are directly processed by the switches themselves, while VMs reside on the data path and hence receive (and process) all packets explicitly. Second, while VMs features virtual ports that can be connected among each other through, e.g., a vSwitch, software bundles do not have ports, then it is not trivial to guarantee that, for instance, flow rules instantiated by a NF B only operate on traffic already processed by flow rules installed by a NF A, in case B follows A in the service graph. Then, the current version of the SDN-DO prototype

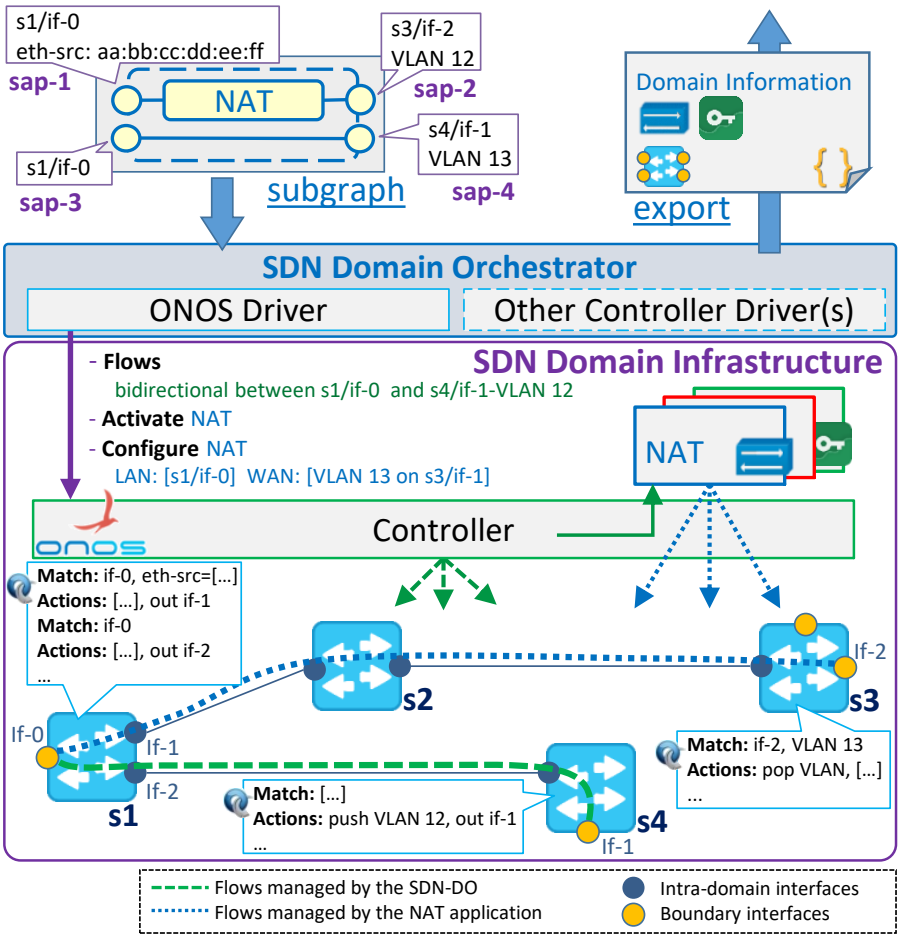


Figure 2.5: Service graph deployment in an SDN domain.

only supports graphs where two SAPs are connected by up to one NF[1].

Unfortunately, not all the bundles available in the SDN controller may be used as NFs. For instance, some of them may not implement any NF (e.g., the bundle that discovers the network topology) while others, although implementing NFs, may not accept the configuration of specific parameters such as the subset of traffic on which they have to operate, thus preventing the SDN-DO to properly set up graph links. In other words, SDN applications must be extended to be compatible with our architecture; our implementation of the SDN-DO (details in Section 2.4.2) looks at specific additional information in the *Project Object Model (POM)* of each ONOS bundle to detect suitable applications.

[1]As described later in this section, we solved a similar problem when chaining flow rules instantiated by NFs and those instantiated by the SDN-DO, e.g., rules needed to set up the inter-domain traffic steering.

Finally, SDN controllers usually do not support multiple instances of the same bundle running at the same time, preventing the SDN-DO to deploy the same NFs as part of different graphs; then multi-tenancy, if desired, has to be managed by the application itself and specified, with the proper syntax, in the POM file as well.

2.4.2 Discovering and exporting domain information

One of the tasks of the SDN-DO is to export information about the domain boundary interfaces and the list of NFs available in the domain. The former information is managed in the same way as the OS-DO. For what concerns the list of NFs, we implemented a new ONOS bundle (named *"app-monitor"* [2]) in charge of collecting the needed information and make them available to the SDN-DO. In particular, *app-monitor* interacts with a specific ONOS API to intercept the following events: *(i) bundle installed* - a new application is available, which may be used to implement a NF; *(ii) bundle removed* - the application is no longer available and cannot be used anymore to implement NFs; *(iii) bundle activated* - the application is running; however, given that ONOS does not support multiple instances of the same application, that application is no longer available for future services (hence, the SDN-DO must not longer advertise that capability), unless it explicitly supports multi-tenancy; *(iv) bundle deactivated* - the application is no longer used, hence it is available again. The app-monitor bundle is in charge to inspect the POM of monitored applications to understand if they provide the capability of running a NF. In such case, the POM should come with a reference to the YANG-based model describing the NF implemented by the application. The app-monitor fetch this information and make it available to the SDN-DO, enriched with any additional status parameter. Each time the status of a NF bundle changes, the SDN-DO updates the exported information and notifies the Overarching Orchestrator accordingly.

2.4.3 Deploying service graphs

When receiving a service (sub)graph, the SDN-DO first validates the request by checking the availability of the requested NFs in ONOS and the validity of the parameters associated with the SAPs (e.g., VLAN IDs, GRE keys). If the graph is valid, the SDN-DO begins the deployment by interacting with the network controller in order to start the proper SDN applications, still not specifying any particular configuration. Graphs links and inter-domain traffic steering information are managed in different ways depending on the fact that the link is between two SAPs, or between a SAP and the port of a NF.

In the former case (e.g., the connection between `SAP-3` and `SAP-4` in Figure 2.5), the SDN-DO, through the SDN controller, directly instantiates the flow rules to setup the connection between the endpoints. Furthermore, it instantiates

the flow rules needed to implement the inter-domain traffic steering, i.e., to properly tag/encapsulate the packets before sending them out of the domain, and to classify incoming packets and recognize the SAP they "belong" to.

Instead, if a link connects a SAP to a NF (e.g., the connection between `SAP-2` and the NAT in Figure 2.5), the SDN-DO configures the application (using the ONOS *Network Configuration Service*) with information about the traffic on which it has to operate, which is derived by the parameters associated with the SAP itself. For instance, the above NAT is configured to operate on specific traffic coming from interfaces `s1/if-0` and `s3/if-2` (i.e., with a specific source MAC address in the former case, with VLAN_ID 12 in the latter), to tag traffic transmitted on `s3/if-2` with the VLAN_ID 12, and to untag traffic tagged with VLAN_12 and received from such an interface. Hence, in this case the inter-domain traffic steering is handled directly by the ONOS application, which has to be aware of these parameters. This is quite complex compared to what happens in the OS-DO, where VMs completely ignore how they are connected with the rest of the graph.

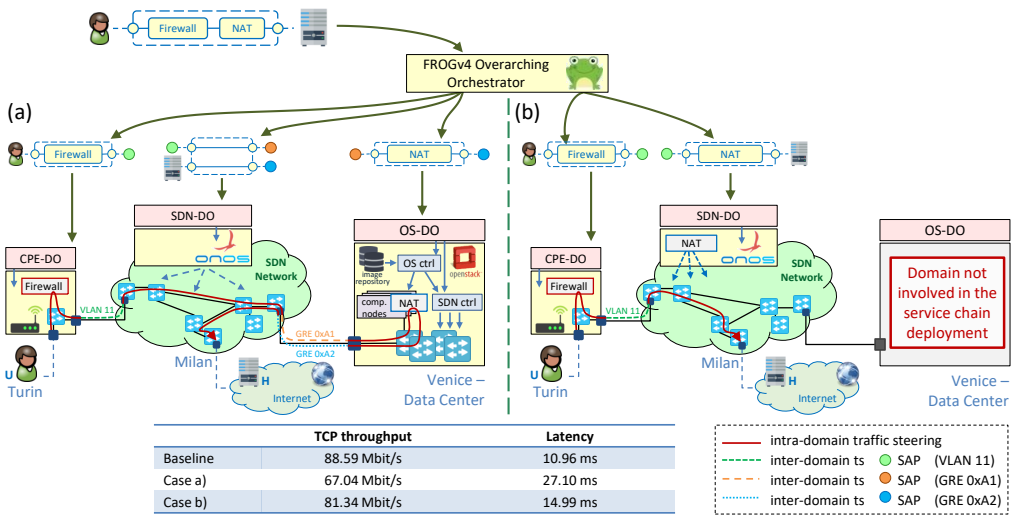
2.5 Experimental Results

We run the proposed orchestration framework over the JOLNET [84], an Italian geographical testbed consisting of an SDN domain that connects various compute clusters and access networks. In our tests, we deployed a service graph consisting of a custom connection between a user and a public server (Figure 2.6. Apart from the SDN experimental backbone, the setup encompasses two other domains: *(i)* a data center, running OpenStack (in Venice), and *(ii)* a CPE[2] that represents the network entry point for the user (in Turin).

Tests have been repeated in two configurations. Initially, (Figure 2.6(a)), the SDN domain has no available NFs, hence it is exploited only for traffic steering. In this case, the Overarching Orchestrator deploys the NAT as a VM on OpenStack, while it manages to optimize the deployment of the firewall by exploiting the capabilities of the CPE. Inter-domain connections are implemented by setting up the proper VLAN and GRE tunnels. In a second time (Figure 2.6(b)), the SDN-DO exports the capability of running the NAT as a bundle on top of the ONOS controller [3], hence the Overarching Orchestrator optimizes the deployment accordingly. As shown in the figure, in this case, the traffic exchanged between the user and the server only traverses the SDN domain, and the data center is not involved at all in the service deployment.

[2]As Domain Orchestrator to integrate the CPE in our framework, we used a previous work available at [25].

[3]The SDN NAT application used for this test is available at https://github.com/netgroup-polito/onos-applications

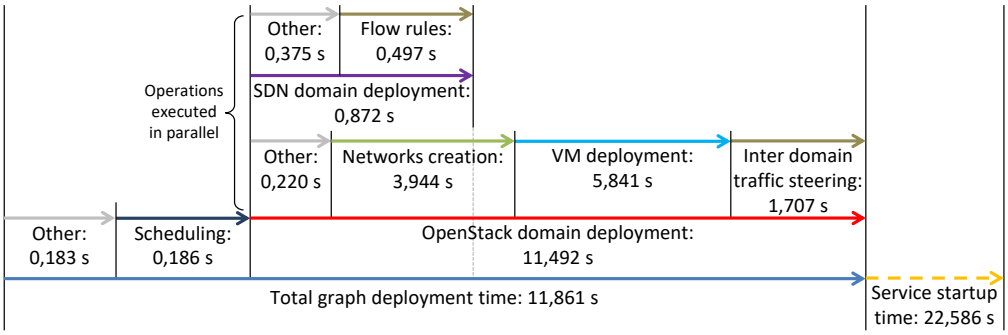


	TCP throughput	Latency
Baseline	88.59 Mbit/s	10.96 ms
Case a)	67.04 Mbit/s	27.10 ms
Case b)	81.34 Mbit/s	14.99 ms

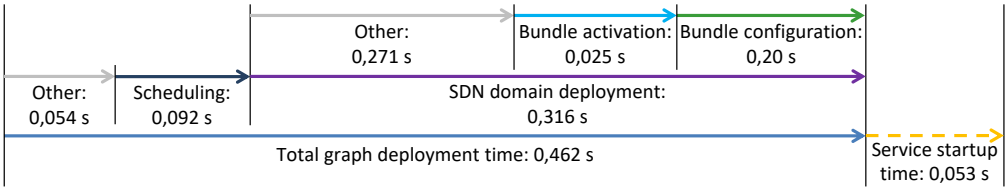
Figure 2.6: Deployment of a service graph in two different scenarios. In case (b), the SDN-DO provides the capability of running a NAT (as SDN application), thus the Overarching Orchestrator opts for better optimization. Performances are compared at the bottom.

For both the deployments, we measured the end-to-end latency introduced by the service (using the `ping` tool) and the throughput (using the `iperf3` tool to generate TCP traffic). Results are at the bottom of Figure 2.6, where also values for a direct connection without NFs are shown (baseline). As expected, the throughput is higher when the NAT is deployed in the SDN domain (Figure 2.6). In fact, in this case, the traffic is kept local to the SDN network, and the NAT is implemented as a set of OpenFlow rules installed on the switches (only the first packet of a flow is processed by the application in the controller).

Additionally, we measured the time needed by the orchestration framework to deploy and start the service. Results are shown in Figure 2.7, which breaks the total deployment time in the several steps of the process. As expected, the major contribution to the service deployment time is given by the VM startup, while the activation of the SDN bundle is almost immediate. As shown in the picture, we also measured the time between the end of the service deployment from the point of view of the overarching orchestrator, and the time when the two hosts were able to communicate. In case of NAT implemented in a VM (on OpenStack), this time is higher because the service starts only after the bootstrapping of the guest operating system. Instead, in case of ONOS bundle, the service starts immediately.



(a) Case of NAT deployed as VM in datacenter.



(b) Case of NAT deployead as ONOS bundle (OpenStack is not involved).

Figure 2.7: Time required to deploy the service graph, highlighting all the main operations and components (horizontal axis not in scale — CPE domain not shown).

2.6 Related Work

The deployment of service chains in heterogeneous technological domains is considered by ESCAPE [148] and FROG [42], two multi-layer orchestration architectures proposed in the context of the FP7 UNIFY project [46]. Similarly, Cloud4NFV [146] deploys network services on different OpenStack and OpenDaylight based environments interconnected through a WAN, while the recent 5GEx project [18] proposes an architecture to deploy services across multiple administrative domains.

Additionally, the problem of embedding VNF graphs in multi-domain infrastructures is studied by many works in literature [149]. For instance, works as [163, 147] proposes to model physical domains based on: *(i)* available amount of resources (e.g., CPU, memory and storage); *(ii)* inter and intra-domain link capacity.

None of these works focus on how to fully exploit the capabilities provided by the heterogeneous domains below: while NFs are executed on VMs and Containers on server clusters (e.g., edge data center), SDN controllers are only used to set up the traffic steering between them, while any resource available on the CPEs at the very edge of the network is not even considered.

Furthermore, existing works do not investigate the information needed to effectively set up the inter-domain traffic steering. Proposals like StEERING [173] and FlowFall [115] can be considered orthogonal to our work, since they define traffic steering architectures that could be exploited within a single infrastructure

domain. A similar consideration also goes for the work carried on by the Service Function Chaining Working Group (SFC) [70] in IETF, which defines a Network Service Header identifying the sequence of NFs that a packet should traverse, provided that the data plane components understand can understand it. Moreover, SFC mainly focuses on the architecture of data plane components.

2.7 Conclusion

In this chapter, we presented a multi-domain orchestration framework based on two hierarchical layers. The proposed approach enables a telco/edge provider to automatically manage its infrastructure, jointly distributing services across multiple technological domains (e.g., SDN backbones, data centers, and even CPE). The architecture needs a Domain Orchestrator for each specific technological domain, which exports (*i*) the list of NFs that the domain is capable to run, and (*ii*) the information associated with the domain boundary interfaces. The Overarching Orchestrator uses such information to transparently instantiate NFs exploiting any capability provided by the infrastructure below. As a result, SDN domains are enabled to provide richer service composition that goes beyond traditional traffic steering, and even the resources of CPEs at the very edge can be exploited. Whenever the service graph is split across multiple domains, the Overarching Orchestrator enriches every sub-graphs with the information needed to set up the inter-domain traffic steering properly.

We also detailed the implementation of two different Domain Orchestrators: one instantiates services in OpenStack-based cloud environments, the other one interacts either with ONOS or OpenDaylight to deploy traffic steering in the SDN network and (in case of ONOS) to execute NFs in the form of software bundles. Other infrastructure domains can be integrated into our framework, assuming that the proper Domain Orchestrator is implemented.

By evaluating our framework on the JOLNET (an Italian geographical testbed), we shown that that the Overarching Orchestrator may introduce significant advantages in the service deployment when all the capabilities of the infrastructure below can be fully exploited.

Chapter 3

Toward a Disaggregated Edge Model: Business Consideration and Optimization Opportunities

Network and Service Providers are exploring different exploitation for the Multi-access Edge Computing (MEC), mainly motivated by the opportunities for saving costs and generating new revenues (e.g., through new business models). While a single provider that manages its own infrastructure may rely on an centralized orchestration approach (e.g., the one described in Chapter 2), this may not be suitable when multiple providers with different roles (e.g., services, cloud, edge, IoT, telco) and possibly different optimization objectives are involved. In this Chapter, we argue that a clear separation of roles for MEC will help accelerating the development of new business actors and models, possibly replacing the current competition-oriented practices in the telco domain with new forms of cooperation, which already appeared in the IT sector. In this direction, we propose a disaggregated MEC architecture by identifying interfaces between MEC actors and their roles, also performing a preliminary analysis of optimization opportunities and algorithmic solutions, which will be then deepened in the next chapters.

3.1 Introduction

The digital transformation triggered by upcoming telecom and ICT technologies will bring a significant impact on current ecosystems, potentially re-designing the roles of Network and Service providers. In fact, the massive softwarization

The work presented in this chapter has been partially published in [37].

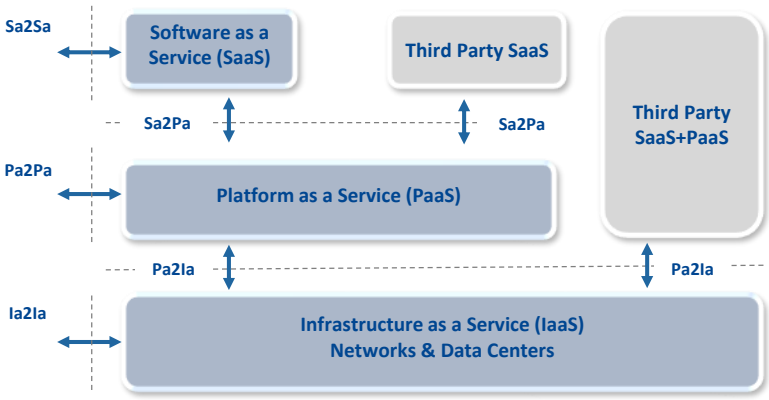


Figure 3.1: Layered model for telco 5G actors, inspired by ETSI White Paper [133].

provided by Network Functions Virtualization, the unprecedented agility guaranteed by Software-Defined Networking, advances in Artificial Intelligence (AI) and the deployment of powerful general-purpose servers at the edge of the (telco) infrastructure, lead to the creation of a *virtual continuum* of resources and services that span from the edge of the network to remote data centers, encompassing telco nodes, IT servers and potentially even terminals (e.g., smartphones), smart things and IoT devices (e.g., sensors).

Edge Computing represents an extension of the cloud paradigm towards the edge (i.e., between core and access nodes) of a telco infrastructure, aiming at providing better QoS/QoE, optimizing the use of bandwidth, and potentially enabling new services and business models. In this transformation, the Multi-Access Edge Computing (MEC) paradigm [158] plays an important role, being it widely accepted as the key technology to meet ultra-low latency requirements as well as to enable a rich computing environment for value-added services closer to end-users.

Without questioning the importance of MEC in future 5G infrastructures, evidence is rising that current well-established cloud computing models will extend to telco operators not only in terms of individual technologies (e.g., general-purpose servers instead of dedicated hardware appliances, deeply re-programmable networks, agile software micro-services), but also in terms of business models and involved actors [137].

In particular, following the layered model in use in cloud computing [135, 99], we speculate that the business value chain in a typical telco will evolve by originating specialized players such as infrastructure providers (a.k.a., Infrastructure as a Service – IaaS), platform providers (a.k.a., Platform as a Service – PaaS), and software providers (a.k.a., Software as a Service – SaaS). However, novel IaaS, PaaS and SaaS operators may be able to offer resources that (*i*) go beyond the traditional trio of computing/network/storage such as IoT devices, (*ii*) are more heterogeneous and (*iii*) may be present in larger numbers, making a clear evolutionary difference

with the existing cloud computing actors. In this respect, each actor will have a clear technical and business role, being in charge of delivering (and selling) richer services to actors that sits higher in the technological stack, which can be seen as a sort of *vertical* interaction toward richer abstractions. In addition, we speculate that, due to the peculiar characteristics of the telco market, additional *horizontal* interactions and novel business opportunities will arise between peers (Figure 3.1).

In this respect, in this chapter we argue that the current monolithic MEC architecture may not be appropriate for future 5G services because of *(i)* its unclear separation of the many required functions between the above mentioned roles (IaaS, PaaS, SaaS), *(ii)* the lack of well-defined software interfaces and *(iii)* the difficulty of enabling new business and service models and new forms of cooperation between network and service providers and third parties. Consequently, without any change to the current MEC building blocks, we propose a novel disaggregated MEC architecture in which clear interfaces between the different actors are foreseen, and show some possible business and optimization opportunities.

This chapter is structured as follows. Section 3.2 summarizes the recent MEC activities in different standardization bodies, while Section 3.3 presents a definition of the IaaS, PaaS and SaaS concepts applied to the 5G scenario, the characteristics of their main interfaces and a disaggregated MEC business model. Section 3.4 discusses possible innovative use cases enabled by the novel disaggregated architecture, while Section 3.5 analyzes the challenges in optimizing interactions between MEC actors and introduces possible algorithmic approaches. Finally, we conclude in Section 3.6.

3.2 State-of-the-Art in Standardization Bodies

Several standardization bodies and fora are addressing MEC and, in general, Edge Computing. Examples include ETSI MEC [86], GSMA [68], TIP [159] — WGs on Edge Computing, OpenFog [122], EdgeX Foundry [53], Open Edge Computing [120], ONF – CORD [143], MobiledgeX [111], Akraino [6].

Let's consider some examples of carried out activities. ETSI Multi-access Edge Computing (MEC) Industry Specification Group (ISG) has released various ETSI Group Specifications for the MEC architecture [86], and Application Programming Interfaces (APIs) to support edge computing interoperability. MEC API Specifications provide a generic set of design principles and patterns. Compliance with these principles ensures consistency across APIs. The work was inspired by TM Forum and Open Mobile Alliance (OMA) work, as well as approaches currently used in developer communities. At this stage, there is no proposition for operational APIs at ETSI MEC ISG.

GSMA [68] has recently addressed MEC techno-economic issues. Major messages were both highlighting the key role of MEC in the Digital Transformation

(Cloudification of telcos) and providing business guidelines for its exploitation. Specifically, it was also recognized by Operators that a MEC functional and system architecture based on a standard decoupling of MEC IaaS vs MEC PaaS is likely to open concrete business opportunities for Telcos, boosting new cooperation models in the direction of creating and developing open ecosystems for various verticals (e.g., IoT, V2X, Industry 4.0, VR/AR, etc). These activities are continuing in 2019 as GSMA set-up a new Work-Item named "MEC API: business opportunities". Main objectives of the group are: *(i)* to review the MEC APIs under definition and standardization (e.g., in ETSI, but not only) in order to identify gaps, business opportunities and exploitation models for the GSMA Stakeholders; *(ii)* to provide recommendations on the use of MEC APIs for relevant business cases, including potential proposals for standardization of further APIs.

Telecom Infra Project (TIP) — Edge Computing WG [159] focuses on enabling service composition and provisioning at the network edge, leveraging open architectures to build a service-aware and ease of use platform. Notably, the WG researches on new revenue generation models for actors in edge computing, with also a focus on providing implementations that could influence standards.

OpenFog Consortium [122] is a forum aiming at the definition of a reference architecture for Fog Computing, which is defined as a further extension of the Cloud–Edge Computing paradigm towards the end-Users' CPE terminals, devices, smart things. Examples of examples addressed by OpenFog include: Smart Building, High-Scale Package Drone Delivery, and more. ETSI MEC and OpenFog are collaborating on the development of APIs for both MEC and Fog Computing: this is meaning that the standardization of a functional architecture means more and more the standardization of APIs to access enabled services.

MobileEdgeX [111] is pursuing the vision of enabling edge open ecosystems. The overall ecosystem is based on an edge-cloud, composed by devices and applications that move from smart-phones to glasses to robots to cars to drones, and the marketplace of edge developer services that power the applications and devices. Underneath there is a distributed edge control fabric that allows efficient placement in real-time and the edge resources and data.

Overall, we are witnessing a fragmentation of efforts in the standardization of MEC/Edge Computing functional and system architectures. On the other hand, all players joining these bodies and fora are recognizing that global interoperability is a must for enabling open services ecosystems and new business models, although most efforts are devoted to north-south interfaces, with little awareness about the necessity of east-west standardization. We argue that the standardization of both east-west and north-south interfaces (e.g., APIs) is crucial to promote innovation and accelerates the development of third-party applications, capable of enabling Network and Service Providers to further capitalize on their investments.

3.3 Toward a new Model for MEC

This section defines IaaS, PaaS, and SaaS for a MEC scenario, characterizing each actor with a precise role, pursued goals and possible constraints, as well as business and functional interactions with the other actors. A summarized view of these concepts is depicted in Figure 3.2.

3.3.1 Definitions

Infrastructure-as-a-Service (IaaS) provider

An IaaS provider virtualizes a set of physical devices, mainly servers and network equipment, offering elementary resources such as CPU, memory, storage, and bandwidth. As shown in Figure 3.2, resources[1] are provisioned as *virtual execution environments* such as Virtual Machines (VMs) or containers (e.g., Docker) of different sizes. Besides, an IaaS provider has to support *slicing*, i.e., the capability to offer different virtual views of the same physical infrastructure, with strong isolation properties. This represents the key feature to enable *multi-tenancy*, i.e., the capability to support multiple independent users (or *tenants*) at the same time, possibly associated with different slices. Networking resources in an IaaS provider include connectivity as well as commonly used functions such as bridges, routers, load balancers, firewalls, tunneling endpoints. Additionally, external connectivity (e.g., toward the Internet, or to a set of xDSL customers) must be explicitly advertised to enable proper connections to other domains; this represents a key difference with the current cloud-based IaaS model. We refer to all the elementary facilities provided by a IaaS (i.e., VMs, network functions, etc.) as *infrastructure components* (Figure 3.2).

Platform-as-a-Service (PaaS) providers

PaaS providers offer platform resources and services (also to third parties) for enabling the development of end-user applications and (end-to-end) services. PaaS resources and services include run-time environment, identity and access management, usage accounting, SLA/QoS management, security, Artificial Intelligence tools, etc. The resulting software platform requires the creation of an additional programming logic acting as a glue between the different components, usually made with a mixture of high-level programming languages (e.g., Java, Python, Go, etc.) and REST APIs. A PaaS provider maps high-level deployment requests into low-level IaaS resources, hence consuming infrastructure components (VMs, storage,

[1]With *resources* we consider both hardware objects (e.g., CPU, network bandwidth, IoT devices), as well as software services (e.g., a database server).

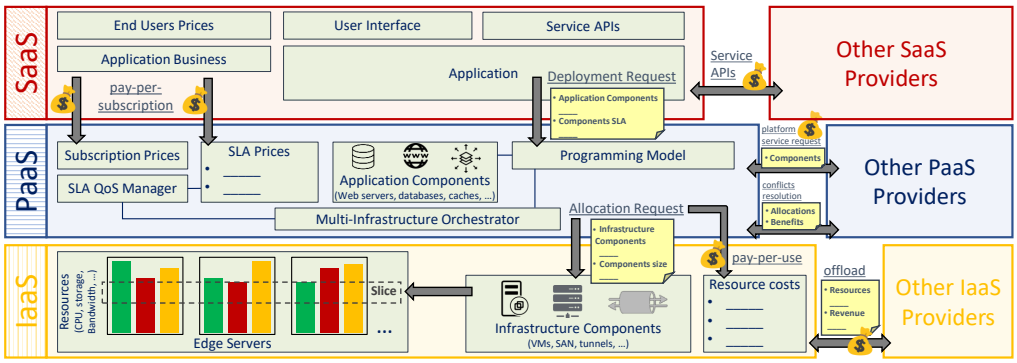


Figure 3.2: Overall disaggregated edge architecture with business interactions.

etc.); this is achieved through a resource orchestrator (Figure 3.2). In MEC, the design of such a central component needs to take into account *(i)* the heterogeneity of underlying resources, *(ii)* the existence of multiple infrastructure providers and *(iii)* the possible availability of a distributed allocation algorithm. Finally, PaaS should also provide slicing capabilities.

Software-as-a-Service (SaaS) Providers

A SaaS provider delivers turn-key solutions, i.e., ready-to-use applications plus a set of analytics to monitor the state of the service itself; tipically, no programming effort is required from its users. Possible examples are video streaming services, content delivery networks, augmented virtual reality, online games. A SaaS provider exploits PaaS facilities to compose, deploy and manage the needed topology of *application components* (Figure 3.2). Similarly to IaaS and PaaS, slicing capabilities are required as well.

3.3.2 Interaction models

An overall view of all the interactions we identified in MEC is depicted in Figure 3.2. At a first sight, interactions between IaaS, PaaS and SaaS look similar to the *vertical* interaction model in use in cloud computing: IaaS export resources to PaaS, which deliver end-to-end services used by SaaS providers to create their applications. In the case underlying domains export different interfaces, entities of the higher layer should create their own *adaptation layers*; for instance, a PaaS provider may offer *deperimetrized* services by establishing relationships with multiple IaaS providers covering different geographical areas.

However, we argue that future MEC players may greatly benefit from an additional *horizontal* interaction model, in which same-level actors can collaborate, enabling each provider to sell also resources owned by its (apparently) competing

siblings. This represents a radical departure for most actors in the telco market, which are more used to competition than collaboration.

Although we are aware that this change of paradigm can potentially create disruption in the current ecosystems, we argue that there are both economic and technical reasons why open collaboration may be a better (and forward-looking) option than pure competition even for current telco players. Note that horizontal interactions can be established not only at the connectivity level (e.g. what usually happens between traditional network providers to enable worldwide communication) but, more important, also at the service levels.

IaaS to IaaS

While cloud computing services, provided by OTT operators in centrally accessible data centers, can be largely considered location-independent, in Edge Computing, services are definitely tied to the location of the physical infrastructure. Interaction between multiple IaaS providers follows the well-established model of creating standard interfaces between different telco operators, which e.g. enables a phone call to cross the boundaries of a single telco. Although future cross-bordering issues may be hidden by creating end-to-end service platforms that establish business relations with multiple IaaS, we foresee several opportunities for an infrastructure provider to collaborate with its peers, enabling to offer a larger infrastructure that may include also (part of) the resources available in partnering IaaS domains. This (*i*) can simplify the operations of a PaaS provider, reducing the complexity required by the interaction with multiple IaaS, and (*ii*) can enable new cooperation strategies between IaaS operators. In fact, they can (*i*) offload services to their IaaS partners in case of resource overload (hence enabling new optimizations) and (*ii*) offer ubiquitous services even in geographic locations in which they have either no access or no economic interest to invest. In particular, this enables an IaaS provider to offer services available on "unconventional" IaaS actors such as enterprise factories or SOHO users, which represent new, tiny IaaS operators. Possible examples of such resources are a set of fog computing nodes in a production plant or an outside temperature sensor/webcam at home. Finally, this may enable further business opportunities for large IaaS providers who can administer the above domain on behalf of their owners, which may not have the technical skills and/or the will to properly operate their infrastructure.

PaaS to PaaS

Future PaaS providers may specialize offered facilities in specific application domains (e.g., high-performance computing, augmented virtual reality), hence possibly requiring their customers (e.g., SaaS providers) to interact with different PaaS actors to create a complex application. In this respect, in addition to the above

mentioned option (i.e., a SaaS establishing business relationships with multiple PaaS providers), we foresee the possibility for a PaaS provider to offer services that are in fact provided by another (partner) PaaS. Such as in the previous case, this opens up optimization opportunities for PaaS operators due to the possibility to specialize their offer in specific vertical markets, while buying other (platform) services from their business partners.

SaaS to SaaS

This interface introduces two main benefits. First, it may be useful to provide a minimum level of compatibility between (competing) applications, hence avoiding e.g. the nuisance of users being forced to login into different instant messaging applications at the same time; for instance, according to the Metcalfe's Law, this would increase the global value of the service. Second, it can be exploited by vertical SaaS providers to create bundles of applications and deliver them together to final users, also privileging a collaborative approach against a competitive one.

3.3.3 Interface standardization

For this systemic paradigm change to succeed, standardized interfaces (either de-jure or de-facto) are required. Standards increase the utility of the system by enlarging the number of potential users and offer new possibilities for cooperation (horizontal interactions). Given that the northbound of a layer can be exploited in either horizontal or vertical interactions, we should define three levels of interfaces.

A standard northbound IaaS interface is the initial mandatory step, facilitated by the reasonably clear understanding of current requirements. Existing northbound of open-source IaaS platforms such as OpenStack [142] or Kubernetes [75] can be taken as initial models and possibly extended to accommodate new characteristics such as the presence of heterogeneous computing nodes, diverse network connectivity between nodes (e.g., bandwidth, resiliency), non-negligible network latency due to the geographically distributed infrastructure, and more.

Standardization of higher layers may be more problematic because of the heterogeneity of the services delivered by PaaS and SaaS platforms. However, restricting the scope of the above platforms to the telco domain, we envision the possibility to standardize typical telecommunication services (e.g., Virtual Private LAN Service (VPLS) [91] or Cloud Radio Access Networks (C-RANs) [79]), hence enabling different PaaS/SaaS platforms operating on diverse infrastructure to cooperate for the end-to-end setup of the standardized services.

3.3.4 Business models

IaaS provider

Infrastructure resources are likely to be provided through a *pay-per-use* business model, i.e. revenue is proportional to the amount of reserved resources. Therefore, an IaaS provider aims at maximizing the amount of resources used by customers over time, e.g., by minimizing the number of physical servers.

Assuming a set of N_E edge servers and N_R different types of infrastructural resources, we can represent the capacity of an infrastructure provider n, in terms of resources available on its edge servers, through the matrix $\mathrm{P}^n \in \mathcal{R}_+^{N_E \times N_R}$, where vector $\boldsymbol{\rho}_i^n$ represents the total amount of resources featured by edge server i owned by infrastructure provider n. A cost vector $\boldsymbol{c}^n \in \mathcal{N}_{\mathcal{R}}$ represents the unitary prices that the IaaS sets for each resource type.

As shown in Figure 3.2, a northbound *resource request* will include the set of needed infrastructure components along with the desired resource values (e.g., memory and CPU for each VM, the bandwidth for a network tunnel, etc.). Moreover, the IaaS interface enables to monitor the state of allocated resources (i.e., VM utilization, latency) and to modify allocation at run-time. The total amount of resources required in a given moment by a set of N_P^n PaaS over IaaS n can be modeled with the matrix $R^n \in \mathcal{R}_+^{N_P^n \times N_E \times N_R}$, where each row $\boldsymbol{r}_m^n$ represents the amount of resources required by PaaS m over the IaaS n, while each element $\boldsymbol{r}_{mi}^n \in \mathbb{R}^{N_R}$ represents the amount of resources that PaaS m required on edge server i.

Since edge environments are characterized by resource scarcity and geographic constraints, the IaaS also features a peer-to-peer interface that can be used to create temporary coalitions. This enables sharing resources to satisfy a higher number of requests, thus improving the overall revenue (Section 3.5).

PaaS provider

A PaaS m consumes resources allocated a IaaS n in the form of infrastructure components. The amount of resources consumed by component i are modeled by vector $\boldsymbol{q}_{ni}^m \in \mathbb{R}^{N_R}$.

A platform provider aims at maximizing the number of customers while minimizing the amount of IaaS resources consumed (e.g., trying to fit as many containers as possible on the same VM). At the same time, it has to guarantee any SLA contract stipulated with SaaS providers (e.g., throughput, maximum delay between two application components). Given SLA contracts stipulated with its customers, a PaaS may calculate the minimum amount of physical resources (of each type N_R) that each application component needs when mapped over the infrastructure to work properly. For each application l, this is modeled by matrix $A^l \in N_A^{lm} \times N_R$, where each row $\boldsymbol{a}_i^l$ represents the amount of resources required by component i of application l (e.g., a point-to-point tunnel, or a web server) and N_A^{lm} is the number

of components of application l to be deployed by PaaS m.

The northbound PaaS interface (Figure 3.2) enables customers to ask for the deployment of application components, along with additional parameters such as logical topology, possible application-specific tunings, SLA constraints. Since platform services are likely to be provided through a *per-subscription* pricing model, customers are expected to pay a fixed fee plus the premium cost associated to additional services, such as the desired SLA. Additionally, the interface toward SaaS providers allows also *(i)* state monitoring (e.g., to estimate the current QoE) and *(ii)* run-time components and topology tuning.

Finally, an additional peer-to-peer interface is used by multiple PaaS to optimally resolve possible conflicts that may occur while allocating the scarce IaaS resources (details in Section 3.5.2).

SaaS provider

As, most likely, end-consumers will be charged with a subscription, the major goal of a software provider is to maximize the number of users; hence, it has to guarantee a competitive QoE (e.g., throughput, response time). On the other hand, a SaaS needs to minimize the costs of its consumed resources, e.g. requesting the appropriate SLAs and minimizing the number of PaaS providers.

A SaaS application l my be split on multiple PaaS. The portion of l that is deployed on PaaS m may be represented as a vector $\boldsymbol{\sigma}_m^l \in \mathbb{N}^{N_A^l}$, where each element models one of the N_A^l application components deployed relying on m.

Assuming that SLA constraints are stipulated at the application component level (i.e., SaaS specifies, granularly, one or more QoS requirement for each deployed component), a SLA contract vector $\boldsymbol{s}_i \in \mathbb{R}^{N_{QoS}^i}$ can be defined for each component i of an application, where its size N_{QoS}^i depends on the number of QoS parameters that i may feature (e.g., throughput of an end-to-end channel, response time and concurrency level of a web server), and each of its elements s_{ij} represents the numerical value of the constraint.

3.3.5 Disaggregated MEC

Current standardization for MEC involves the functional definition of the overall architecture and the consequent interfaces between the resulting building blocks. Particularly, the latter aims at guaranteeing the interoperability between different implementations and, potentially, different vendors, which is only one of the requirements in case of a scenario in which Infrastructure, Platform and Software services may be provided by different actors. In fact, in this case, the interface must support additional parameters such as authentication, accounting, billing; the current monolithic architecture, instead, assumes that all the functional blocks are under the control of the same organization.

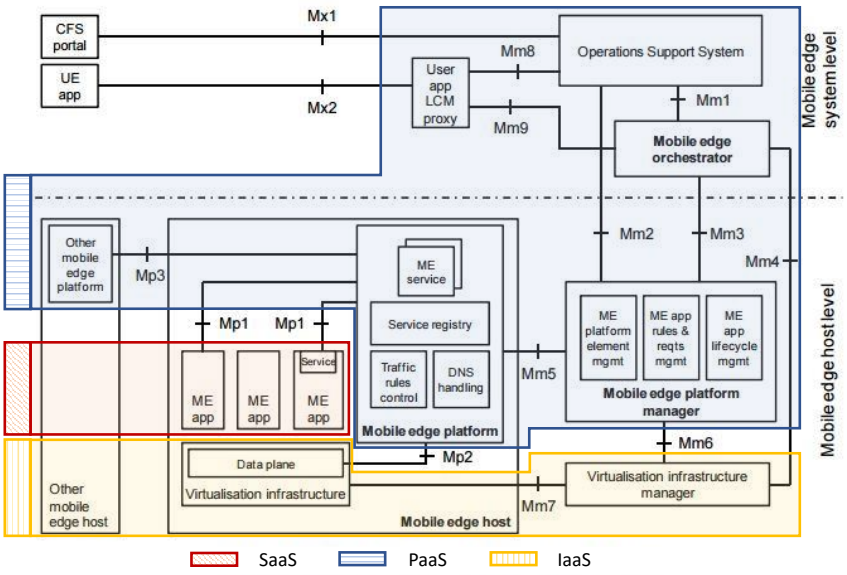


Figure 3.3: Disaggregated MEC architecture.

We propose the disaggregated MEC architecture depicted in Figure 3.3, which shows a possible splitting of functionalities between different IaaS, PaaS and SaaS providers. In addition, we highlight some interfaces (`Mp2`, `Mm6`) that must be extended to support cross-actor interactions, enabling the MEC platform to run on an infrastructure owned by a different entity, and even on different infrastructure domains. Furthermore, interface `Mp1` enables a third-party software provider to install its applications on top of the MEC platform.

The standardization of the above interfaces will have a profound impact on both technological and business sides. With respect to the former, the different pieces of the architecture can be sold (if we refer to technological providers) and/or operated (concerning network providers) by different actors, which can be different business units of the same company or different companies. On the business side, this would open up the market to multiple specialized actors, which can establish either competition or cooperation relationships. Finally, we expect this to bring a new breed of novelty to customers, which can experience new services and innovative offers thanks to the breakage of current monolithic network providers, replaced by more specialized and possibly competing actors.

This would transform the relatively slow world of network and service providers into a fast-growing and innovative area, similar to personal computers and smartphones. The splitting of concern between different (vertical) actors is considered one of the keys of the extraordinary evolution of the above markets over time, which was possible by unleashing the power of independent developers that were continuously enriching the original platform. Similarly, this vision aims at an innovative

evolution of the concepts of Service Delivery Platforms and Service Layers of former times. Specifically, IaaS-PaaS-SaaS decoupling together with standardization (either de-jure or de-facto) of the related interfaces will overcome static and vertical service silos that delay open innovation. That "openness" will enable multiple interactions of providers at different levels, will ease the role of apps developers, allow "deperimetrization" of services and create new business opportunities for all players of the new ecosystems.

3.4 Use Cases

As an example, we now present how two novel possible use cases can be mapped to the disaggregated MEC model, identifying actors and analyzing their interactions both from a business and technical point of view.

3.4.1 Everywhere in the city

In this first use-case, we provide each person the set of digital services more appropriate in each given location. For instance, in a supermarket we provide information about available offers; at work, we offer fast access to local services such as shared printers and servers or calendars of business partners; at home present a dashboard to control ambient-assisted living and smart appliances, heating, security alarms, surveillance cameras, and more. Other examples are the provisioning of infotainment services to both car drivers/passengers and pedestrians in smart roads and city hot spots (e.g., commercial areas, stadium, stations, etc.).

We speculate there might be one or more telco IaaS providers covering the overall city, while the stadium sets-up dedicated IT equipment to be used in case of live events, and the supermarket shares part of its IT infrastructure for serving local people. All the above actors may collaborate to provide a ubiquitous service through a logically partitioned infrastructure. A disaggregated MEC can enable end-to-end services, being them either provided through different infrastructures or platforms. The pervasive distribution of MEC IaaS and PaaS allows improving the users experience thanks to latency reductions and no service interruption on mobility. Standard interfaces create an open ecosystems boosting the opportunities for developers and providers. From a business perspective, a telco may wish to play the role of IaaS provider and/or PaaS provider; in the former case, it will need to engage business relationships with third parties playing the roles of MEC PaaS providers (e.g., municipality, supermarket, stadium) owning MEC platforms.

3.4.2 Enterprise connectivity services

This second use case envisions an enterprise willing to offer integrated connectivity services to its employees. The most common current options are to buy/install services such as VPN, and/or buy a "private mobile network" to enable users to communicate at convenient prices, possibly with internal (short) phone numbers. In the case of telco services provided by distinct IaaS-PaaS-SaaS actors, the enterprise may have several additional options. It can easily become a "virtual operator", providing its remote users with native network services, being them xDSL connections or direct data mobile connectivity (through enterprise-branded SIM cards), with the assurance that its remote users will be always securely connected to the corporate network without having to install/launch any VPN software. The "virtual operator" service can be either created by buying the elementary components from a PaaS provider and adding the additional logic to create the requested service, or by simply buying a turn-key software from a SaaS provider. The PaaS provider, from its side, will be in charge of establishing the proper business agreements with multiple IaaS providers (e.g., the IT infrastructure in the main corporate site, which represents the main infrastructure to connect to) to enable mobile users to take advantages of the above services whatever physical infrastructure they are in.

3.5 Interactions Optimization

Based on the business model proposed so far, this Section analyzes the interactions between different parties of the disaggregated MEC architecture, thinking to each actor as a rational individual which aims at maximizing its own profit according to the behavior of other providers. Additionally, we propose and discuss possible optimization strategies that providers can adopt for each different interaction.

3.5.1 Interaction between PaaS and SaaS

This interaction is mainly driven by the prices that the PaaS decides to set for its services. We identified two main prices: *(i)* the *subscription price* p_{sub}, i.e., the fixed fee a customer has to pay in order to use the platform facilities (typically paid periodically) and *(ii)* the prices $\boldsymbol{p}_A$ of any additional SLA that the SaaS stipulate for particular application components (Section 3.3.1).

PaaS can adjust its prices to motivate SaaS providers to use their virtual resources. In so doing, it should try to optimize a trade-off. In fact, by setting low unit prices, a higher number of customers (SaaS providers) will be encouraged to purchase its services, but the overall outcome may result in sub-optimality. On the other hand, high prices lead to higher revenues per unit of consumed resources, but customers may decide to switch to other PaaS featuring lower prices.

While deciding prices for its virtual resources, the PaaS is in a position of power towards customers. Since SaaS providers take their decisions after the prices have been set, a PaaS (leader) can predict the optimal SaaS (follower) decision given the current market situation and anticipate it by determining the optimal service price. A similar concept exists in economics, and it is known as *Stackelberg leadership competition* [166]. The model can be solved by *backward induction*. The PaaS estimates the best response function of customers given the current market situation (subscription and SLA prices for all PaaS). Then, the best response function is embedded in the PaaS profit function and maximized.

Profit for PaaS m can be modeled as

$$\Pi^m_{PaaS} = \sum_{l=1}^{N_S} y_{lm} \left[p^m_{sub} + \sum_{i=1}^{N^{ml}_A} \boldsymbol{p}^m_{Ai} \cdot \boldsymbol{s}^l_i - \sum_{i=1}^{N^{ml}_A} C_I(\boldsymbol{a}^l_i) \right], \tag{3.1}$$

where $\boldsymbol{a}^l_i$ (Section 3.3.1) is the amount of resources that PaaS needs to purchase from IaaS to deploy component σ^l_i satisfying SLA constraints $\boldsymbol{s}^l_i$, $C_I(\boldsymbol{a}^l_i)$ is its cost on the infrastructure, and being variable $y_{lm} = 1$ if SaaS l decides to use PaaS m, zero otherwise.

Since SaaS tries to provide a high QoS while minimizing the platform costs, PaaS may estimate the profit functions of its potential customers as

$$\Pi^l_{SaaS} = \sum_{m=1}^{N_P} y_{lm} \left[\sum_{i=1}^{N^{ml}_A} QoS^l_i(\boldsymbol{s}^l_i) - p^m_{sub} - \sum_{i=1}^{N^{ml}_A} \boldsymbol{p}^m_{Ai} \cdot \boldsymbol{s}^l_i \right], \tag{3.2}$$

where function QoS^l estimates the quality of service SaaS l obtains by deploying a particular component i with SLAs $\boldsymbol{s}_i$, while N^{ml}_A is the number of components that the l decides to deploy using PaaS m. Of course, the QoS may depend on the nature of the SaaS application; moreover, a PaaS may not know it exactly. However, a PaaS may employ some representative QoS functions for given clusters of SaaS, and proportionate them basing on known market analysis data (e.g., percentage of video streaming applications, etc.).

PaaS may use (3.2) to find the *best response function* of each SaaS, i.e., a function that, given as input the price vector of PaaS m, returns whether it is convenient for SaaS l to use m or not ($y_{lm}(\boldsymbol{p})$), and the values of any SLA constraint to stipulate with each PaaS ($\boldsymbol{s}^l(\boldsymbol{p})$).

At this point, PaaS m can rewrite its profit in function of its price vector $\boldsymbol{p}^m$ by substituting best responses $y_{lm}(\boldsymbol{p})$ and $\boldsymbol{s}^l(\boldsymbol{p})$ in (3.1). Finally, PaaS may determine the optimal prices $\boldsymbol{p}^m$ maximizing (3.1), by studying its gradient $\nabla\Pi^m_{PaaS}(\boldsymbol{p}^m)$.

As a result, every PaaS will adjust their prices towards SaaS iteratively, based on the current market situation.

3.5.2 Interaction among PaaS providers

A key role of the PaaS is to map high level (virtual) resource requests coming from the SaaS, i.e. application components, into low level (physical) resources owned by the IaaS. As in MEC IaaS resources are constrained, it may be the case that a PaaS, independently from business consideration, has to decide how to distribute the allocation of application components on available IaaS. However, multiple, potentially concurrent, PaaS providers try to allocate resources for different applications over the same set of IaaS providers, thus exacerbating the already hard problem of resource mapping introducing a distributed concurrency.

In [35], we proposed an asynchronous, fully distributed, resource assignment algorithm that coordinates a set of actors in deciding how infrastructure resources have to be temporarily assigned, basing on what they need to allocate to meet service constraints and preserving the overall infrastructure optimality. The algorithm, which is detailed in Chapter 5, can be used to solve the concurrent resource mapping problem of PaaS. In the following, we provide a high-level description.

Each PaaS m runs an *Orchestration Phase* where an optimal allocation is computed, i.e., the PaaS decides which resources it needs on every IaaS to deploy a given application. The allocation of PaaS m to deploy application l is represented with an allocation matrix $X_l^m \in \mathbb{R}^{N_I N_R}$, where each element x_{lnk}^m indicates the total amount of resource k that PaaS m is going to allocate on IaaS provider n to deploy application l. At this point, a *score vector* $\boldsymbol{v}^m \in \mathcal{R}_S^N$ is assigned to the computed allocation matrix, based on the benefit (e.g., QoS) it provides. In particular, a score for each involved IaaS is generated. PaaS then uses generated scores to participate to a distributed resource *election*, where last known information from other PaaS are used to decide how to assign resources of each IaaS by mean of a greedy approach. If the election result changes from the previous iteration, PaaS sends to its peers last known scores $\boldsymbol{v}^{m'}$ and allocations $X^{m'}$ for every known PaaS $m' \in N_P$, then waits for an analogous response coming from any number of them. During an Agreement Phase, all new data received from each peer are used in combination with the local values, to reach an agreement with it. The two phases are repeated iteratively until an overall agreement is reached.

The algorithm above has been proved to guarantee both an upper bound on convergence time and an optimal approximation bound with respect to the Pareto optimal resource assignment.

3.5.3 Interaction between PaaS and IaaS

The main difference with the PaaS-SaaS interface is that in MEC, a IaaS features scarce resources. Therefore, an infrastructure provider may not want to indefinitely increase the number of its customers, since at a certain point it will stop to gain profit from this. This means that a IaaS may set a high price for the remaining

scarce resources if the number of customers is still high enough. We represent with $\boldsymbol{c} \in \mathbb{R}^{N_R}$ the vector of the unitary prices that the IaaS sets for each resource type.

The profit for an IaaS n can be modeled as the revenue obtained from selling resources, minus the energy cost of active edge servers:

$$\begin{aligned} \Pi^n_{IaaS} &= \sum_{m=1}^{N_P} \sum_{i=1}^{N_E} \boldsymbol{c}^n \cdot \boldsymbol{r}^n_{mi} - E^S S_{\#}(\boldsymbol{r}) \\ \text{subject to } \sum_{m=1}^{N_P} r^n_{mik} &\leq \rho^n_{ik} \quad \forall i \in [1,..,N_E], \forall k \in [1,..,N_R], \end{aligned} \tag{3.3}$$

where E^S is the price in energy to keep a server active, while $S_{\#}$ gives the minimum number of servers needed to allocate all resources $\boldsymbol{r}$ (a calculation example for this value can be found in [131]).

IaaS providers indirectly compete with each other while setting resource prices. This model can be solved using the *subgradient algorithm* [19]. Being unaware of concurrency, a new IaaS starts setting its prices at a high value. Then, it predicts the profit that can be achieved by adjusting its price vector of a small quantity $d\boldsymbol{c}$ and modify it consequently, waiting for feedback from the market. This is repeated iteratively until no more profitable adjustments are found. On the other hand, each PaaS, as a consequence, decides which resources to purchase based on the optimization of its profit (3.1), which can be rewritten as

$$\Pi^m_{PaaS} = \sum_{l=1}^{N_S} y_{lm} \left[p^m_{sub} + \sum_{i=1}^{N^{ml}_A} \boldsymbol{p}^m_{Ai} \cdot \boldsymbol{s}^l_i - \sum_{i=1}^{N^{ml}_A} C_I(\boldsymbol{a}^l_i) \right], \tag{3.4}$$

substituting the linear costs $\boldsymbol{c}^n$ of each IaaS provider.

3.5.4 Interaction among IaaS providers

In MEC, allocating the limited edge resources to a large number of customers is a challenging problem for IaaS providers. Whenever an IaaS runs out of resources, it may be convenient to cooperate with its peers borrowing the missing portion of resources needed to satisfy some additional requests. IaaS providers may thus increase their profit by sharing the revenue coming from the additional requests satisfied this way.

In a recent work [172], authors propose a Game Theoretic approach for resource sharing between servers belonging to different providers. The problem is modeled as a coalition game between a set of players $\mathcal{N}$, where the goal is to maximize a characteristic function for each player $n \in \mathcal{N}$:

$$\max_{\mathcal{X}} \left(w_n u_n(\mathcal{X}) + \xi_n \sum_{j \in \mathcal{N}, j \neq n} u^n_j(\mathcal{X}^n) \right) \quad \forall n \in \mathcal{N}, \tag{3.5}$$

where $\mathcal{X}$ is the allocation decision vector, which indicates how much resources each player (IaaS) allocated for each customer request, while $u_n(\mathcal{X})$ indicates the profit received by player n with allocation $\mathcal{X}$ for requests coming from its own customers. Hence, $u_j^n(\mathcal{X})$ is the profit that IaaS n receives for sharing its resources with IaaS j. Finally, w_n and ξ_n are two weights that IaaS n may set to properly prioritize its own customers, and then share remaining resources with peers.

Since solving the problem by computing the Shapley value (fair measurement of the contribution of each player [144]) for the coalition game above requires solving the optimization problem $2^N - 1$ times, a heuristic than only need solving it $2N$ times is proposed. Each server first computes its optimal allocation without considering resource sharing and updates residual resources accordingly. Then, remaining resources are used to optimize the sum $\sum_{j \neq n} u_j^n(\mathcal{X}^n)$, i.e., the utility of sharing them with peers. It can be easily proved [172] that the solution obtained by this algorithm lies in the core of the coalition game, i.e. no player has the incentive to leave the coalition.

3.6 Conclusion

MEC is expected to play a key strategic role in the transition towards 5G. Network and service providers are exploring different strategies to introduce and exploit MEC concepts, mainly motivated by the potential opportunities for saving costs and generating new revenues.

Although several standardization bodies and fora are targeting MEC and, in general, Edge Computing, current solutions are still lacking end-to-end interoperability, which is a must for enabling open ecosystems. In this chapter, we argued that a clear architectural decoupling of IaaS, PaaS and SaaS for MEC, and then for 5G, represents an evolutionary step of the digital transformation capable of enabling new roles and business models. Furthermore, this may blur the borders between current OTT operators, which are mostly offering datacenter-only services, and network providers, which own the network infrastructure that connects datacenters to the customers. In this respect, network providers represent the most suitable actors to also offer edge-based resources such as IoT and domestic/enterprise-owned IT infrastructures at the edge of the network, hence positioning themselves in a stronger position when competing with current OTT actors.

This chapter defined a disaggregated MEC architecture identifying the main actors and their roles, also performing a preliminary analysis of optimization opportunities and algorithmic solutions, which will be deepened in the next chapters. Eventually, we propose to pursue joint efforts of providers and vendors involved in the MEC architecture. Such an open model is likely to boost the service provisioning ecosystem with new business opportunities and cooperation of network and service providers with third parties.

Chapter 4

Service-Defined Orchestration of Heterogeneous Applications in Cloud/Edge Platforms

Edge Computing is moving resources toward the network borders, thus enabling the deployment of a pool of new applications that benefit from the new distributed infrastructure. However, due to the heterogeneity of such applications, specific orchestration strategies need to be adopted for each deployment request. Each application can potentially require different optimization criteria and may prefer particular reactions upon the occurrence of the same event. In this chapter, we present a Service-Defined approach for orchestrating cloud/edge services in a distributed fashion, where each application can define its own orchestration strategy by means of declarative statements, which are parsed into a Service-Defined Orchestrator (SDO). The problem of coordinating the coexistence of a variety of SDOs on the same infrastructure while preserving the resource assignment optimality, will be later addressed in Chapter 5. We evaluate the advantages of our novel Service-Defined orchestration approach over some representative use cases of services running at the edge of the network, assessing the benefits compared to conventional orchestration approaches.

4.1 Introduction

With the expansion of Cloud Computing toward the edge of the network, the diversity of the involved applications and their management requirements has been

---

The work presented in this chapter has been partially published in [36]. The first prototype of the proposed solution has been implemented by Emanuele Fia during his master's thesis [59].

drastically exacerbated. Indeed, the largely heterogeneous set of (often distributed) applications running over Cloud/Edge platforms may have different and unpredictable deployment objectives; furthermore, reacting differently to network events, such as a traffic load increase, became a necessity. In the above circumstance, some applications may need to scale up or out, while others may migrate to a more convenient location. Others may even modify the service behavior without asking for additional resources. For instance, the optimization of a Content Distribution Network (CDN) service may require to monitor the average miss-rate on deployed caches, to identify occasional hot spots such as flash crowd during live events; this in turn requires optimizing the service by relocating *and possibly duplicating* some caches. Vice versa, a video streaming application may need to monitor the provided quality of service in terms of Frames Per Second (FPS), opting for the deployment of a more aggressive video transcoder whenever a particular high load deteriorates the current frame rate, instead of asking for more processing resources.

While large service providers may run their applications on ad-hoc platforms and therefore can define their best optimization strategies, most of the other providers have to rely on third-party Edge/Cloud platforms, which host heterogeneous applications and hence in need of adopting service-agnostic one-size fits-all orchestration strategies to handle the entire applications pool. Embedding of virtual services is thus accomplished by optimizing generic metrics such as energy saving, latency, load balancing [56]. Similarly, load increases are managed by taking into account conventional infrastructure metrics such as CPU and memory consumption through the traditional auto-scaling techniques (i.e., scaling up/down is performed whenever consumption goes above/below a certain threshold). In summary, existing orchestration approaches cannot *(i)* take their decisions based on service-specific parameters (i.e., cache miss-rate) and *(ii)* perform service-specific actions such as modifying the internal behavior of the service (e.g., switching the transcoder), as only infrastructure-related actions are possible. As a consequence, they often fail to optimize application-specific goals.

This chapter fills this knowledge gap by proposing a novel Platform-as-a-Service (*PaaS*) approach where the orchestration is fully distributed and provides the possibility to instantiate, prior to service deployment, small-scoped *Service-Defined Orchestrators* (SDOs), each dedicated to handling the life-cycle of a particular application. Such orchestrator may operate by modifying the current overall resource assignment (e.g., if the application needs more resources or may release some) as well as merely act on the application itself to adapt its operational state to the current infrastructure situation (e.g., switch the used video codec on a streaming service component). To avoid exacerbating applications development, which would also include such an orchestration module, we present a distributed architecture where SDOs are automatically synthesized starting from an high-level description of their orchestration strategies (*Orchestration Behavioral Model*), used by the service provider to specify metrics and objectives needed to build the proper orchestration

strategy for the given application by means of high-level declarative statements.

The contributions of this chapter are as follows. We present a novel distributed orchestration architecture and detail the design of its core component, namely the Service-Defined Orchestrator. We also formally define a Declarative Behavioral Model used to synthesize an SDO from an high-level description of its orchestration strategies. Finally, We assess the benefits of our approach analyzing three reference use cases: *(i)* QoS degradation for a video streaming application, *(ii)* cache placement for a CDN provider and *(iii)* edge migration for mobile gaming.

4.2 Related Work

The orchestration of infrastructure resources is primarily investigated in several recent works. Most of them focus on the VNF deployment problem proposing algorithms that rely on a centralized solver [52, 51, 150]. Among them, some propose a joint computation of different phases of the problem to seek better optimization. For instance, [51] proposes an algorithm in which scaling, placement, and mapping are optimized jointly, while in [150] authors solve the embedding problem combined with the service composition one. Other works as [96, 127] investigate instead the problem of joint orchestration among multiple infrastructure providers, proposing distributed optimization approaches. In [96], the authors propose a game-theoretic approach, while [127] illustrates a decentralized algorithm on top of an existing multi-domain architecture developed in the 5Gex project [18]. This last one addresses relationships across multi-administrative domain orchestrators, distinguishing between *Resource Orchestration*, service-agnostic and performed at the infrastructure level, and *Service Orchestration*, i.e., service-specific management of a single slice [69]. Within the project, a distributed architecture enabling multi-domain resource orchestration is proposed, as well as an analysis of their coordination [18, 48]. However, no focus is given on the service orchestrators and their interaction, which is only theorized in [69] and is mostly outside the scope of the project. Infrastructure level orchestration alone does not provide service specific optimization.

Indeed, recent work on edge computing [109, 13, 165] proposes ad-hoc optimization focusing single edge applications separately. For instance, [109] optimizes the placement of roadside units on new generation vehicular networks; [13] focuses on the service placement problem in mobile applications, where the dynamism of the user's location plays a key role; [165] proposes an optimal allocation for high-performance video streaming in 5G networks. While the above solutions enable optimization of isolated applications, to the best of our knowledge, it is still unclear how such a variety of service embedding algorithms can be integrated on top of an existing framework to coexist on a shared infrastructure. SONATA [92] introduced the concept of service-specific optimization in the ETSI NFV reference

architecture, extending it with Service-Specific Managers (SSMs) micro-services, so that service awareness can be dynamically introduced to the generic orchestrator. In this work, we even go further by enabling a fully distributed approach to remove such a centralized component. Indeed, mandating the existence of a centralized component (featured both in [77, 92]) may not be suitable in Edge Computing, where services are executed on scattered compute nodes and multiple providers are involved. As this work proposes the usage of a declarative model to characterize service specific orchestration strategies, we based it on existing solutions of declarative modeling [132]. In particular, we generalize the application specific approach of [113] by adopting some of the concepts from [76] on formalizing declarative workflows, with the aim of enabling the description of subsuming any deployment and run-time orchestration strategy.

4.3 Overall Architecture

This section introduces our distributed orchestration architecture (Figure 4.1). We identified three separated operational planes that have a correspondence with the XaaS layered model [135], which is well-established in IT and that we analyzed in Chapter 3 for the scenario of Edge Computing. An Infrastructure Plane (i.e., Infrastructure-as-a-Service) provides elementary resources (such as computing, networking, and storage) by virtualizing a set of edge or cloud servers (compute nodes) scattered across the network. A set of Service-Defined Orchestrators, each dedicated to the management of a given application, constitutes a distributed Platform-as-a-Service that we name Orchestration Plane. Finally, the whole set of edge/cloud end applications running on top of the distributed infrastructure constitutes the Service Plane (i.e., Software-as-a-Service).

4.3.1 Service Plane

To preserve the generality of our approach, we assume that applications may follow the micro-service paradigm [50], that is, services are composed of small components, each specialized on a given task. For instance, a video streaming application may feature a video source, a transcoder, and a web server, each one potentially deployed on a separate location according to the placement decisions enforced by the orchestrator.

Figure 4.1 shows a Service Plane where each application requires the deployment of multiple components. The proposed architecture assumes that each application component may feature multiple valid implementations when physically deployed on the infrastructure. Each implementation may feature different characteristics, for example, the execution environment could be over virtual machines, containers, or dedicated hardware. Each implementation may require different resources

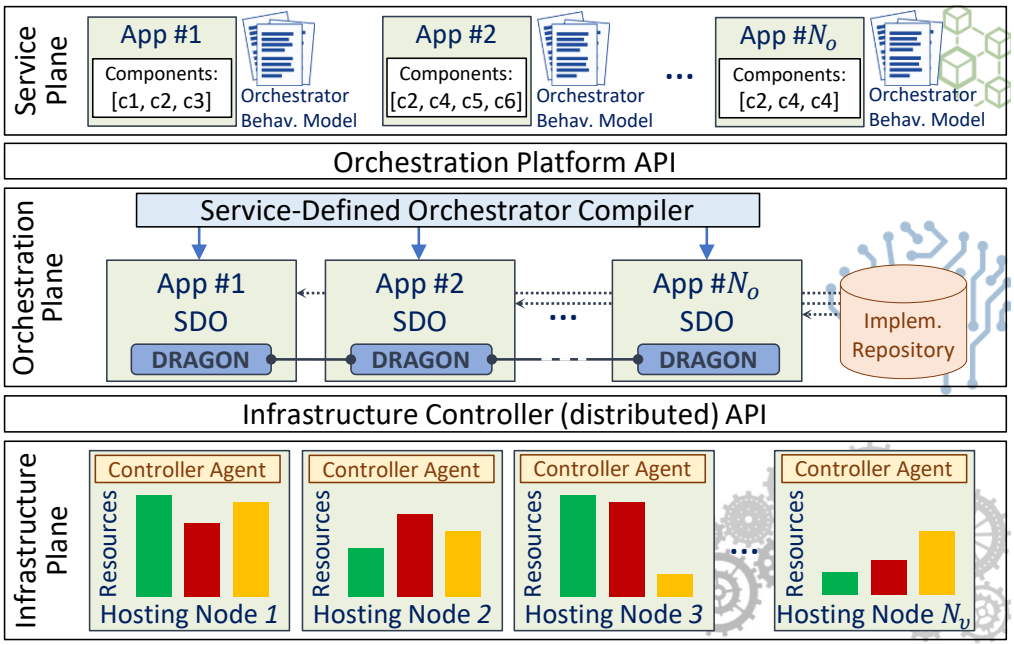


Figure 4.1: Overall distributed and service-defined orchestration architecture.

and provide different QoS levels. Based on the scenario, an application may benefit more from a particular implementation policy set. As shown in Figure 4.1, the Orchestration Plane features an *Implementation Repository* that stores valid implementations for well-known components, along with details about their configuration and resource requirements. Additionally, at deployment request time, an application may customize further each one of its components.

An application deployment request consists of *(i)* the list of components to be deployed, along with their virtual topology, *(ii)* any custom implementation needed for the deployment and *(iii)* a declarative description of the orchestration strategies to be used to properly deploy and manage the application.

4.3.2 Orchestration Plane

Our approach defines a highly modular and dynamic Orchestration Platform, whose building blocks are our Service-Defined Orchestrators (SDOs), each *(i)* dedicated to a single application and *(ii)* generated and executed on demand. Note that this approach makes the overall PaaS behavior defined by the application itself.

The orchestration platform accepts application deployment requests. These requests come with additional information that is used to drive the orchestration process (i.e., resource allocation, placement, and run-time management) in a way that is optimal for that specific application. Metadata that comes with each application deployment request are in the form of an *Orchestration Behavioral Model*,

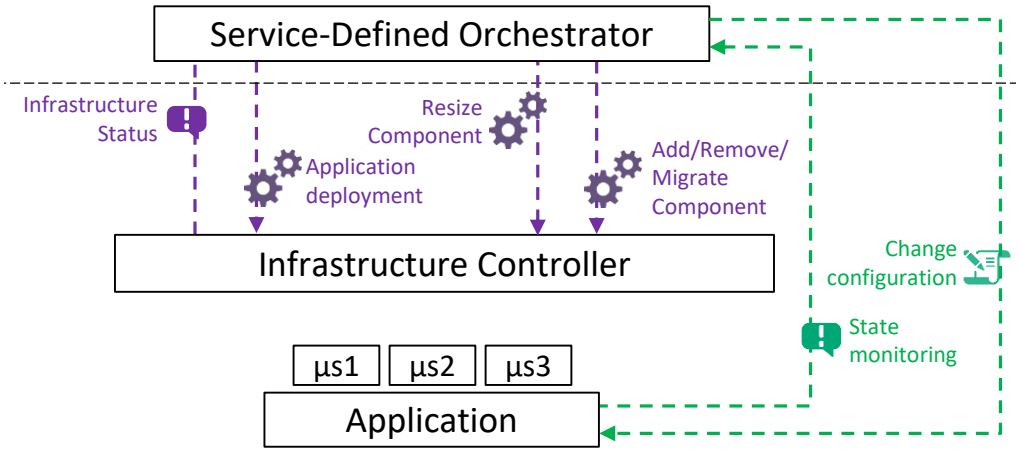


Figure 4.2: Interactions between an SDO and *(i)* the infrastructure controller and *(ii)* components of the managed application.

which features declarative statements used to actually generate the Service-Defined Orchestrator by means of an *SDO Compiler*. Details regarding the Orchestration Behavioral Model and how its declarative statements are composed to generate an SDO are discussed in Section 4.4.

Whenever a new SDO is generated, it is instantiated as an extension of the existing platform and employed to manage the orchestration of the corresponding application. Orchestration is performed with respect to both *deployment* and *run time*. At deployment time, the SDO interacts with the infrastructure to decide where each component should be physically deployed (placement) and the amount of resources to reserve (resource allocation). At run time, the SDO monitors the state of both application components and infrastructure, reacting to suboptimal placements and resource allocation. SDO actions include rescheduling components or resizing applications on demand.

4.3.3 Infrastructure Plane

After orchestration decisions have been taken, application components are physically deployed on a shared hosting infrastructure. The infrastructure is partitioned in multiple hosting nodes (Figure 4.1), each featuring different physical capacity in terms of resources of different types (e.g., CPUs, storage, network bandwidth, etc.). The current state of the hosting infrastructure and the state of each deployed application components are dynamically reported to the relevant SDOs by each hosting node, by means of a distributed message broker. Additionally, infrastructure nodes expose resource controller APIs through which SDOs can deploy and manage application components. Figure 4.2 highlights SDO interactions with both the infrastructure and application components.

4.4 Service-Defined Orchestrator

This section provides details on the Service-Defined Orchestrator (SDO), the on-demand generated piece of the platform that manages deployment and run time of a single application. We first provide a formal definition of the Orchestration Behavioral Model, used to describe a specific SDO behavior through declarative rules. Then, the architecture and synthesis of an SDO are detailed, and a practical example is discussed.

4.4.1 Orchestrator Behavioral Model (OBM)

By definition, an SDO cannot be a generic module, as it should necessarily be specialized for each particular application. Application needs in an edge/cloud environment are usually unknown a priori, so we need a mechanism that allows, on-demand, generation of any desired orchestration strategy. We propose an approach that derives a specialized SDO starting from a high-level declarative description. In this section, we formalize such a description through an *Orchestration Behavioral Model* (OBM). Our design generalizes the application-specific approach of [113] by adapting some of the concepts from [76] on the formalization of declarative workflows, with the aim of subsuming any deployment orchestration strategy and encompassing multiple run-time situations.

An OBM instance is provided with the application as deployment metadata, specified by the service provider. It should feature at least the following: *(i)* parameters which the SDO should be aware of, such as infrastructure and/or application state; *(ii)* the objective that should be optimized; *(iii)* events that may occur and actions to be performed in response. We formally define the OBM by providing the following abstractions.

Definition 1. *(state $\mathcal{S}$). We define as state $\mathcal{S} = \mathcal{S}_A \cup \mathcal{S}_I \cup \mathcal{S}_O$ the set of variables, parameters and, in general, configurations, that the SDO can have access to. We distinguish three separate sets composing it: Application State ($\mathcal{S}_A$), Infrastructure State ($\mathcal{S}_I$) and SDO State ($\mathcal{S}_O$).*

Each element $s \in \mathcal{S}$ represents a generic readable and, possibly, configurable parameter within a given area, and is associated with a few information (e.g., name, type, scope).

The *Application State* ($\mathcal{S}_A$) concerns the current deployment of each application component. It includes *(i)* the list of implementations currently chosen for each component (and where they have been physically deployed), *(ii)* their configuration and *(iii)* any *operational variable*, i.e., read-only data that the SDO may obtain by directly querying one or more components (e.g., the current miss-rate measured on a given deployed content cache).

The *Infrastructure state* ($\mathcal{S}_I$) is mainly the set of information the SDO obtains from the hosting nodes below, namely, their resource capacity and topology data. Since more than one SDO concurrently allocates resources over the same infrastructure, we add another piece of information to the Infrastructure State, i.e., how much resources the SDO is allowed to allocate at the moment. This is obtained by each SDO through our distributed agreement algorithm that will be described in Chapter 5.

Additionally, the OBM features an *SDO State* ($\mathcal{S}_O$), which is maintained internally to the SDO itself and can be used to store some run-time information, thus enabling the definition of stateful behaviors.

Definition 2. *(constraints $\mathcal{C}$). We define a set of constraints $\mathcal{C}$, where each element $\gamma \in \mathcal{C}$ is a mathematical statement (equation or inequation) between two functions $f_L, f_R \colon \mathcal{S}^{|\mathcal{S}|} \to \mathbb{R}$ defined on state variables $s \in \mathcal{S}$.*

Constraints represent additional requirements associated with a given application. They can state the maximum latency between two components, specific characteristics of the physical nodes where a given component has to be deployed, and more. If constraints are specified in the OBM, they are interpreted by the generated SDO as hard requirements that should always be satisfied, thus discarding any deployment solution that would violate them. Moreover, since variations that may occur at run-time in the *State* may possibly lead to a constraint violation, *Actions* to be performed (see below) upon such violation must be specified for each declared constraint.

Definition 3. *(events E). Given a state $\mathcal{S}$, a particular set of variations that may occur at run-time on its variables may be declared to be an event $\boldsymbol{e} \in E$, where E is the set of all the events declared in the OBM.*

A service parameter that exceeds a given threshold, the amount of a given resource that drops below the configuration requirement, the expiration of a timer defined at run-time on the SDO State, and more, can identify situations where the application is suffering and reconfiguration actions should be performed. An *Event* may be defined on a single state variable variation, or even when more than one of the variables change in a predefined way. Each *variation* is defined through: *(i)* the reference to the state variable in question; *(ii)* the kind of variation that must be observed, i.e. *equal* to, *higher* or *lower* a threshold, or it simply changes in *any* way; *(iii)* the value, if any, to which the changing variable should be compared, which can be a static value (e.g., a string or a number) or even a reference other variables on the state. Each event is labeled with a name, which is used to associate *Action*(s) that should be performed upon its occurrence.

Definition 4. *(actions $\mathcal{A}$). Given an event $\boldsymbol{e} \in E$ that may occur on a state $\mathcal{S}$, we define as action $\boldsymbol{a} \in \mathcal{A}$, a vector of functions $a_i \colon \mathcal{S}^{|\mathcal{S}|} \to \mathcal{S}$, each giving the new*

value to "write" on a particular state variable $s_i \in \mathcal{S}$. The size of the action vector $\boldsymbol{a}$ represents the number of write operations to be performed on the state.

Executing an action may consist of one or more of the following: *(i)* modify the configuration of a given application component; *(ii)* reschedule the deployment of the application, or scale/migrate just a particular component; *(iii)* update some local variables of the SDO State, e.g. to modify the SDO future behavior. Whenever an action is invoked, it takes as implicit parameters any variation registered by the triggering event.

Definition 5. *(objective o). Given a state $\mathcal{S}$ declared in an OBM, we define the objective of the associated SDO, and we denote it with $o\colon \mathcal{S}^{|\mathcal{S}|} \rightarrow \mathbb{R}$, the numerical function that the SDO should optimize during the application deployment.*

The objective function of an application should model one or more service QoS metrics (e.g., the frame rate in a video streaming application) through variables that correspond to placement and resource assignment decisions during the deployment. The objective optimization process is modeled through a default, implicit *Action* $\boldsymbol{a}_o$, which is automatically invoked at deployment time. Additionally, one can declare to invoke the same action in response to some particular events, in order to reschedule application components on resources from scratch when necessary.

In declaring their own orchestration strategies through the OBM, service providers are able to take into account the multiple situations a service may face while operating. The remote need for re-defining the strategies while the service is operating may constitute a limitation. However, in a real framework product, an SDO specified through declarative statements should provide an increased level of flexibility compared to developing and deploying ad-hoc software.

4.4.2 SDO Architecture

Figure 4.3 shows the architecture of the Service-Defined Orchestrator. The figure distinguishes between modules that are dynamically generated from the Orchestration Behavioral Model, and those that are fixed (hence application-independent). Each of the dynamically generated modules has a direct correspondence to a precise piece of the OBM. Their description is provided in the following.

The *State Module* maintains the run-time information about all the state variables described in the dedicated section of the OBM (Definition 1), distinguishing between SDO internal variables, information related to the infrastructure and each deployed application component. An *Event Listener* implements the detection of events declared in the OBM through Definition 3. Whenever there is a variation on one of the relevant state variables, the Event Listener checks for events that may have occurred. Additionally, this module also checks if a state variation causes a violation of one of the defined constraints. In any of these cases, the Event Listener

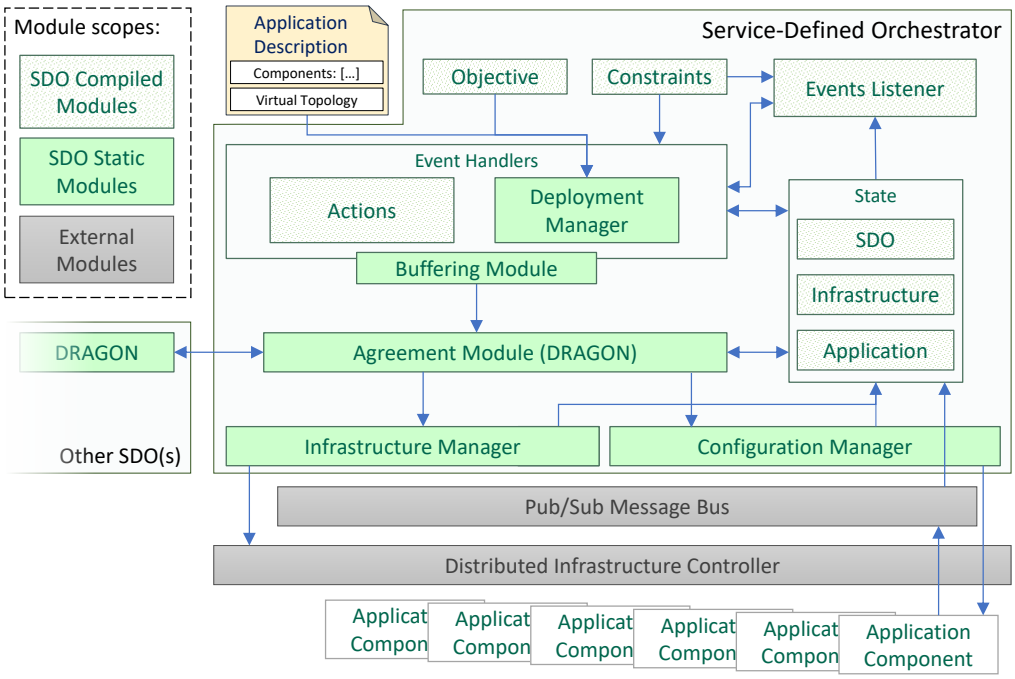


Figure 4.3: Overall architecture of the Service-Defined Orchestrator.

invokes the corresponding *Handler*, that is one (or more) of the Actions defined in the OBM. If the particular event that occurred requires to reschedule the entire deployment of the application, action $\boldsymbol{a}_o$, i.e., the application deployment, is invoked instead. This action implements the optimization of the application objective declared on the OBM and is performed by the *Deployment Manager*, which schedules a solution based on *(i)* the current state, *(ii)* defined constraints and *(iii)* the *Application Description*. Upon SDO startup, this deployment action is automatically triggered by the Event Listener.

Whenever an action is triggered, all writing operations are buffered by a helper module, which checks if any of them requires the acquisition of additional resources from the physical infrastructure. If this is the case, the execution is mediated by an *Agreement Module*, which, through a distributed consensus algorithm, negotiates resource assignment with other SDOs operating over the same infrastructure (details are given in Chapter 5). After the agreement, adjustments deriving from executing the action are propagated to the relevant modules.[1] In particular, an *Infrastructure Manager* acts as an interface towards the infrastructure controller and is in charge of allocating resources on needed hosting nodes and scaling up/down instantiated components. On the other hand, a *Configuration Manager* pushes any

[1]Note that such adjustments may derive both from the execution of a local Action, and from any change on the equilibrium with external SDOs.

new configuration on the appropriate deployed component. Both changes in infrastructure and on components are also propagated into the corresponding portion of state maintained within the SDO. Additionally, the state is also updated any time a change notification is received from the infrastructure or any application component. Such communication occurs over a pub/sub based message bus, while the state and configuration of each component are described using the YANG language.

4.4.3 A practical use case: the video streaming application

We now provide a practical example of declaring an orchestration strategy through our model. As a reference use case, we use a video streaming application. For the sake of brevity and clarity, we only focus the scope of a single component within the service run-time.

Let us consider a video streaming application whose components are: *(i)* a video transmitter (the media source), *(ii)* a transcoder, *(iii)* a web server and *(iv)* a series of clients consuming the output video streams. The transcoder takes as input the original video stream and generates multiple output streams at different bit rates and resolutions, so that each client may select the most appropriate stream based on the available bandwidth. In particular, it is in charge of three tasks: *transsizing*, i.e., resizing the frames from the original media source; *transrating*, that is reducing the bit rate of the stream in order to reduce the required bandwidth; *transcoding*, i.e., re-encode the stream using a different codec, e.g., because certain codecs may not be available on some end devices, or may require different computational resources or energy. Let us assume that the available implementation is configurable with the number of output streams to be generated. Each of these configurations has associated a minimum requirement in terms of CPU resources. Within the video streaming applications, the streaming protocol (e.g., RTMP, HDS, HLS) performs periodical measures to estimate the end-to-end bandwidth and, whenever this is not suitable for the current bitrate, switches to a lighter stream among those generated by the transcoder. As it performs computation intensive tasks multiple times on the original source to generate different output streams, the transcoder may suffer in situations where assigned resources are not enough to guarantee the proper generation of the desired output streams and the availability of resources is temporary scarce.

In such a case, an example of a service-specific orchestration strategy is to fix the number of output streams (transcoder configuration) according to the available resources. A constraint is defined on a variable of the Infrastructure State, i.e., the available CPU resources. The constraint definition references a parameter in the transcoder component configuration: when the available CPU value drops below the requirement specified for the current configuration, the constraint is violated and an event is triggered. The action associated with this violation features a single write operation defined as follows: the variable to modify is on the Application State,

i.e., the set of output streams; the new value to assign is computed by selecting, among the available setups, the one that *(i)* provides the higher number of output streams and *(ii)* fits the newly available CPU resource. This run-time orchestration strategy, alone, does not require any write operation on the Infrastructure State.

4.5 Experimental Results

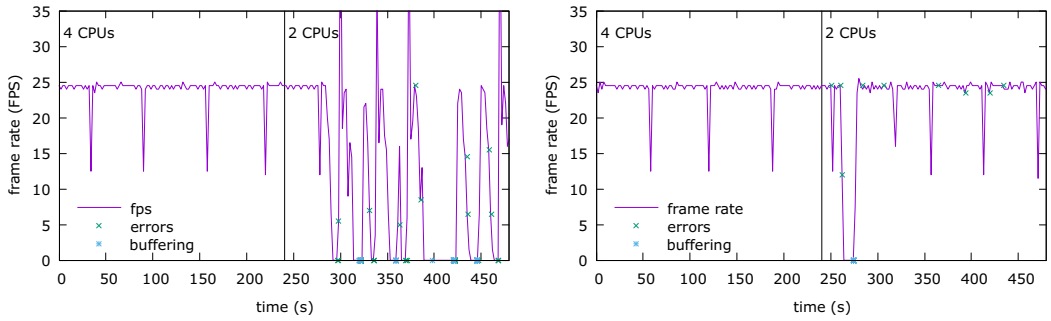
To validate our approach, we implemented a prototype of the SDO compiler. Our code is available at [61, 60, 34]. Our evaluation analyzes three use cases: stream management on a video streaming application, cache placement for a CDN provider and process migration for mobile gaming.

We show the advantages for service providers when deploying their applications on an infrastructure that adopts our service-defined orchestration approach and does not restrict the available orchestration strategies. We aim to show that, for example, a CDN provider that relies on a third party platform/infrastructure to serve a certain area may benefit from using its own cache placement algorithm (e.g., [85]), running over DRAGON, rather than depending on a one-size fits-all embedding orchestrator.

At first, we setup a virtualization infrastructure to orchestrate the deployment and run-time of a video streaming application. Additionally, we setup a simulated environment to evaluate two different edge use cases: *(i)* cache placement for a CDN provider [158], and *(ii)* edge migration for mobile gaming [140]. In our tests, we compared the provided QoS resulting from different deployment approaches, also varying the concurrency level by adding some concurrent applications, thus evaluating the behavior when resources become scarce.

Video Streaming

We deployed an use case analogous to the one described in Section 4.4.3. A VM generating an RTMP stream through FFmpeg implements the video transmitter, which sends a H264 stream of size 1080x640, with a frame rate of 24 FPS and a bitrate of 1330 kbps. Both the web server and the transcoder have been implemented through the Wowza software [168], generating 9 streams at different bit rates (from 176x144 to 1080x640) using three different codecs (VP8, H263 and H264), with a resulting bitrate ranging from 150 kbps and 1000 kbps. All streams use AAC (96 kbps) as audio codec. The streaming server delivers the output streams using the HLS streaming protocol. We deployed all the server-side components on a KVM based infrastructure (Hypervisor Debian Linux 4.14.0-3 on i7-6700 CPU 3.40 GHz, RAM 32 GB). On a second machine, we run the VLC software to consume the output streams and measure the QoS in terms of frame rate; we perform measurements through the libVLC library, periodically evaluating the number of frames that are correctly shown, their size, the player state (Playing, Stopped, Buffering, etc.), the



(a) One-size fits-all approach

(b) Service-Defined approach.

Figure 4.4: Frame rate over time for a video streaming application in a resource constrained situation. A Service-Defined Orchestrator may effectively mitigate the QoE deterioration being able to explore alternative solutions.

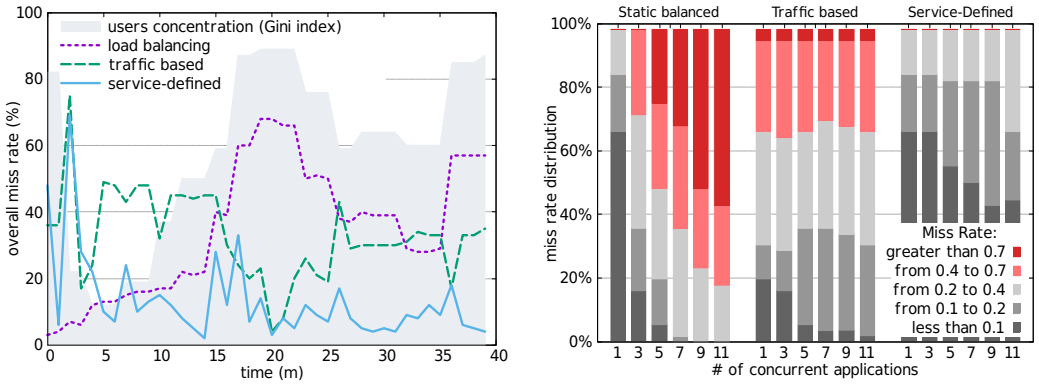
number of lost frames (i.e., arrived out of sequence) and the current bit rate. We emulated a scenario in which, after 240 seconds, some of the CPUs originally allocated for the transcoder VM are no longer available. Figure 4.4 summarizes our findings.

(i) When resources are reduced for the transcoder VM, a one-size fits-all orchestrator can neither understand that the transcoder is suffering (it has no generic parameter to base itself upon) nor it can identify a solution in such a constrained situation, i.e. the VM cannot be scaled out since there are no more resources locally and no other edge nodes are available to migrate the VM. As a consequence, Figure 4.4a shows a significant degradation of the provisioned service.

(ii) If the application is managed by an SDO, a custom action can be defined to be executed whenever it is not possible to assign a given amount of CPUs to the transcoder component; in such a case, a possible service-defined solution may configure the transcoder component to disable some of the generated output streams (e.g., those at a higher resolution), thus reducing its workload and resource requirements. Figure 4.4b shows that this behavior effectively preserves the frame rate after switching to a lighter configuration.

CDN Caches

A CDN provider provisions content caches over an edge network where user density dynamically changes across compute nodes. The objective of the provider is to minimize the average miss-rate occurring on deployed caches. The CDN application should be adapted on events where a set of users shifts from a node to another. In our tests we simulated a set of 100 users moving over a network of 10 edge computing nodes. To visualize the user distribution among nodes, we also report the Gini index (a high index indicates that most users are located near few host nodes). We summarize our findings in a few take home messages:



(a) Miss rate over time.

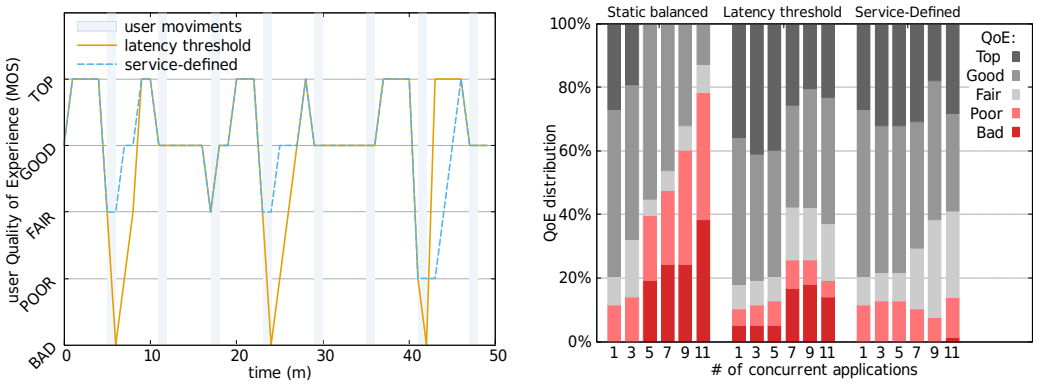
(b) Miss rate distributions.

Figure 4.5: Evaluation of a CDN cache provisioning application comparing different placement strategies: (a) miss rate over time varying the geographical users distribution; (b) distribution of measured miss rate varying the number of concurrent applications.

(i) A one-size fits-all approach that places caches by balancing the resource consumption per node achieves good performance when users are well distributed, but the number of miss-rate grows fast when the concentration increases (Figure 4.5a). A similar result is obtained by statically partitioning the resources among coexistent applications (Figure 4.5b) when their number is high with respect to the available resources.
(ii) A one-size fits-all approach that places caches according with the traffic load on each node achieves optimal miss-rates when users are concentrated on few nodes, while the performance is poor otherwise. This is because a low traffic amount on a certain node does not necessarily mean that users are consuming less variety of contents. Figure 4.5b shows a slight degradation when increasing the concurrency.
(iii) If application caches are placed by an SDO based on current miss-rate on each node, optimal miss-rate both for low and high users concentration is achieved (Figure 4.5a). Moreover, note how Figure 4.5b does not show a noticeable QoS degradation when increasing the number of concurrent SDOs, showing the scalability of our approach. This is achieved through our coordination algorithm that will be detailed in Chapter 5.

Mobile Gaming

A gamer moves into an area served by multiple edge nodes. Whereas it may be convenient to relocate (part of) the game application components to better fulfill the latency requirements, the relocation may happen in a crucial phase of the game, causing undesirable service degradation [140]. Therefore, if the deployment is managed by an SDO, it may be instructed to recognize the time frame in which a relocation is most appropriate (e.g., after the gamer reaches a checkpoint or during the loading of a new level).



(a) QoE over time.

(b) QoE distribution.

Figure 4.6: Evaluation of a mobile gaming application for different deployment strategies: (a) QoE over time perceived by a user moving in different areas; (b) QoE distribution varying the number of concurrent applications.

In our tests we simulated a user moving every 6 minutes across a network of 10 edge nodes. We measured the Quality of Experience perceived by the user based on latency and packet loss, using the same Mean Opinion Score (MOS) described in [82] for medium-paced games. Our findings are summarized as follows (Figure 4.6ab):
(i) Statically partitioning resources between applications does not scale (Figure 4.6b): the application may be unable to migrate components on needed nodes, since resources are assigned to other peers, despite not being currently used.
(ii) If the resources are managed by a one-size fits-all orchestrator that minimizes the end-to-end latency, the user often experiences a QoE level that we label as `bad` due to some process relocation occurring during the game session (Figure 4.6a). Figure 4.6b shows that the percentage of `bad` QoE measurements even may increase with the concurrency.
(iii) If the relocation decision is taken by an SDO, and resources are dynamically assigned with DRAGON, it is possible to define a behavior that does not migrate the service rapidly whenever the user moves away; even if this may temporarily increase the latency, it prevents undesirable service degradation during a game session and the overall perceived QoE results improved (Figure 4.6a). Figure 4.6b also shows that this approach scales well with the number of concurrent SDOs.

4.6 Conclusion

In this chapter we proposed a Service-Defined approach for orchestrating applications in cloud/edge infrastructures. The proposal enables individual applications to define their own orchestration strategy by means of a behavioral declarative model, which is used to dynamically generate a service-specific orchestrator

(Service-Defined Orchestrator - SDO). This component is in charge of both the deployment (e.g., resource allocation) and run-time orchestration (e.g., scaling, reconfiguration) of the given application. Being service providers free to define their preferred Orchestration Behavioral Model instance, each application may benefit from ad-hoc orchestration strategies encompassing service-specific metrics, objectives, reaction to custom events and special constraints. We evaluate our Service-Defined approach over three representative edge use cases, showing how infrastructure/platform providers may enable their customers (i.e., service providers) to implement the preferred orchestration strategy for their services, without the restrictions of relying on a conventional one-size fits-all orchestrator.

The coexistence of multiple orchestrators operating over the same infrastructure introduces the problem of coordinating resource allocation while preserving the resource assignment optimality. In the next chapter we propose a time-bounded distributed approximation algorithm that solves the problem of optimally partitioning a shared pool of resources between multiple SDOs.

Chapter 5

A Distributed Orchestration Algorithm for Edge Computing Resources with Guarantees

Edge Computing brings the flexibility and scalability of virtualization technologies at the edge of the network, enabling service providers to deploy new applications over a richer network infrastructure. In Chapter 4 we proposed a distributed orchestration approach where a small-scoped Service-Defined Orchestrator is assigned to each application, thus enabling custom optimization. However, the coexistence of such variety of decision entities on the same infrastructure exacerbates the already challenging problem of coordinating resource allocation while preserving the resource assignment optimality. In fact, *(i)* each application can potentially require different optimization criteria due to their heterogeneous requirements, and *(ii)* we may not count on a centralized coordination due to the highly dynamic nature of edge networks. To solve this problem, this chapter presents DRAGON, a Distributed Resource AssiGnment and OrchestratioN algorithm that seeks optimal partitioning of shared resources between different applications running over a common edge infrastructure. We designed DRAGON to guarantee both a bound on convergence time and an optimal (1-1/e)-approximation with respect to the Pareto optimal resource assignment. We evaluate convergence and performance of DRAGON on a prototype implementation, deploying both on physical nodes and within a simulated environment. Results assess the benefits of DRAGON compared to traditional orchestration approaches.

The work presented in this chapter has been partially published in [35] and [36].

5.1 Introduction

The emerging Edge Computing paradigm has enabled service providers to supply a large variety of new applications, which benefit from the presence of storage and computing facilities at the edge of the network, as well as reduced latency toward end-users. Distributed content delivery and caching, Internet of Things, disaster response, vehicle-to-everything automotive, and video acceleration are only some of the multitude of services [158] that can benefit from being deployed at the edge of the network. Thanks to virtualization technologies, such a different applications can be isolated and simultaneously run on separated slices of the same shared physical infrastructure.

In cloud-based environments, the slicing operation is often delegated to a centralized Orchestrator [57], that usually exploits some one-size-fits-all policies (e.g., energy-saving, number of used nodes, load balancing) to decide *(i)* where to place service components, *(ii)* how many resources have to be assigned to each of them [74] and *(iii)* the set of metrics/events signaling that the service has to be rescheduled (e.g., because of an unexpected load increase).

As highlighted in Chapter 4, the scattered nature of the edge infrastructure, along with the largely heterogeneous set of applications, suggests that such a centralized approach may be sub-optimal or not applicable in Edge Computing. Therefore, using multiple Service-Defined Orchestrators to allocate resources of different services over the same physical infrastructure seems a natural approach to enable service-centric optimization. However, coordinating such a plethora of applications without relying on a centralized orchestrator brings to light several challenges. How could several optimization processes, each operating with different goals and policies, converge to a globally optimal resource management over a shared edge infrastructure? How could we avoid violations of global policies or feasibility constraints of several coexisting applications? How can we guarantee convergence to a distributed resource allocation agreement and performance optimality given the NP-hard [7] nature of the service placement problem?

To answer these questions, this chapter presents DRAGON, an asynchronous Distributed Resource AssiGnment and OrchestratioN algorithm. DRAGON leverages the max-consensus literature and the theory of submodular functions to enable a set of applications, featuring diverse objectives and optimization metrics, to reach an agreement on how infrastructure resources have to be (temporary) assigned, without the necessity of a centralized orchestrator.

We first introduce the (NP-hard) *Applications-Resources Assignment Problem* and use linear programming to model its objective and constraints (Sections 5.3). Finding a centralized optimal solution is often infeasible even for a single optimizer. We use the solution to the centralized problem as a baseline global optimal to show DRAGON's performance optimality guarantees.

Then, we detail our DRAGON asynchronous algorithm (Section 5.4) and we

show how it provides non-improvable guarantees on resource assignment performance to a set of independent edge application, and an upper bound on convergence time (Section 5.5).

Finally, we evaluate both performance scalability and convergence properties of DRAGON, comparing them with traditional approaches and deploying both on physical nodes and within a simulated environment (Section 5.6). Our findings confirm the applicability of this approach in edge infrastructures and the performance advantages over conventional one-size-fits-all orchestration paradigms.

5.2 Related Work

Despite recent work on edge computing proposes ad-hoc optimization focusing single edge applications separately [109, 13, 165], it is still unclear how such a variety of service embedding algorithms can coexist on a shared infrastructure without undermining the overall performance optimality. Indeed, most of the work on the orchestration of infrastructure resources focus on the VNF deployment problem proposing algorithms that rely on a centralized solver [52, 51, 150]. Some works as [96, 127] investigate the problem of joint orchestration among multiple infrastructure providers, proposing distributed optimization approaches. In [96], the authors propose a game-theoretic approach, while [127] illustrates a decentralized algorithm on top of an existing multi-domain architecture developed in the 5Gex project [18]. The 5Gex Project also identifies the difference between *Resource Orchestration*, service-agnostic and performed at the infrastructure level, and *Service Orchestration*, i.e., service-specific management of a single slice [69]. However, service orchestrators are only theorized and no focus is given on how to coordinate resource partitioning among them, as this is mostly outside the scope of the project.

In cloud environments, Mesos [77] enables dynamic resource partitioning and allows the coexistence of diverse cluster computing frameworks, each one featuring different scheduling needs. It exploits a master that assigns resources dynamically by making offers to demanding frameworks. However, mandating the existence of such a component may not be suitable in a scenario where services are executed on scattered compute nodes, e.g., at the edge of the network, which features arbitrary topologies and multiple providers. In this context, we should rely on solutions that provide decentralized consensus (e.g., Paxos [94] and RAFT [119]) to reach agreement on resource assignment. In particular, RAFT is implemented in widespread SDN controllers to enable data-store replication and resiliency among multiple controller instances. Some of its limitations have already been highlighted in [136], where authors also propose an enhancement of RAFT to improve recovery times on the specific use case of SDN Controllers. More generally, none of [94, 119] simultaneously provides *(i)* guarantees on convergence time and performance, and *(ii)* a fully distributed approach.

5.3 Problem Definition and Modeling

This section defines the (NP-hard) *applications-resources assignment problem* by leveraging linear programming.

Let us model an *application* as a multiset — a set in which element repetition is allowed — whose elements are selected among N_μ (abstract) application components to be embedded on a shared (physical) edge infrastructure. A *component* is an abstract instance of a physical function, e.g., a load balancer, a video transcoder or a content cache, which can be run by selecting the best possible physical *implementation* among the N_f available ones. In fact, each implementation may feature different characteristics such as execution environment (virtual machine, container, dedicated hardware), required resources, or the capability to provide a specific level of QoS. or provided level of QoS.

The infrastructure is partitioned in N_v hosting nodes, each one with potentially different physical capacities. We assume that each function consumes a given amount of resources such as CPU, storage, memory, network bandwidth, etc., which are modeled with N_ρ different types.

Finally, let us consider N_a applications, all simultaneously demanding resources from a shared edge infrastructure, each one following a potentially different optimization strategy. We assume that the application itself will select the best (feasible) implementations that are required to realize its composing services, then allocate them in the most appropriate location.

Our goal is to maximize a global utility U while finding an infrastructure-bounded applications-resources assignment that allows the deployment of each application. We define an applications-resources assignment to be *infrastructure-bounded* if the consumption of all assigned components allocated on each hosting node does not exceed the $\boldsymbol{\rho}_n$ available resources on that node.

We model the applications-resources assignment problem with an integer program; its binary decision variable x_{ijn} is equal to one if an instance of the implementation j has been assigned to application i on hosting node n and to zero otherwise.

$$\text{maximize} \qquad \sum_{i=1}^{N_a} \sum_{j=1}^{N_f} \sum_{n=1}^{N_v} U_{ijn}(\boldsymbol{x}_i) x_{ijn} \tag{1.1}$$

subject to

$$\sum_{i=1}^{N_a}\sum_{j=1}^{N_f} x_{ijn}c_{jk} \leq \rho_{nk} \qquad \forall k \in \mathcal{K},\ \forall n \in \mathcal{N} \tag{1.2}$$

$$\sum_{j=1}^{N_f}\sum_{n=1}^{N_v} x_{ijn} = \sum_{m=1}^{N_\mu} (\sigma_{im})y_i \qquad \forall i \in \mathcal{I} \tag{1.3}$$

$$\sum_{j=1}^{N_f}\left(\sum_{n=1}^{N_v} x_{ijn}\right)\lambda_{mj} \geq y_i \qquad \forall m \in \mathcal{M},\ \forall i \in \mathcal{I} \tag{1.4}$$

$$\sum_{n=1}^{N_v} x_{ijn} \leq 1 \qquad \forall j \in \mathcal{J},\ \forall i \in \mathcal{I} \tag{1.5}$$

$$\sum_{j=1}^{N_f} x_{ij} \geq 1 - N_f y_i \qquad \forall i \in \mathcal{I} \tag{1.6a}$$

$$\sum_{j=1}^{N_f} x_{ij} \leq 1 N_f y_i \qquad \forall i \in \mathcal{I} \tag{1.6b}$$

$$x_{ijn} \in \{0,1\} \qquad \forall (i,j,n) \in \mathcal{I} \times \mathcal{J} \times \mathcal{N} \tag{1.7a}$$

$$y_i \in \{0,1\} \qquad \forall i \in \mathcal{I} \tag{1.7b}$$

$$c_{jk} \in \mathbb{N} \qquad \forall (j,k) \in \mathcal{J} \times \mathcal{K} \tag{1.7c}$$

$$\rho_{nk} \in \mathbb{N} \qquad \forall (n,k) \in \mathcal{N} \times \mathcal{K} \tag{1.7d}$$

$$\lambda_{mj} \in \{0,1\} \qquad \forall (m,j) \in \mathcal{M} \times \mathcal{J} \tag{1.7e}$$

$$\sigma_{im} \in \{0,1\} \qquad \forall (i,m) \in \mathcal{I} \times \mathcal{M} \tag{1.7f}$$

where $\boldsymbol{x}_i \in \{0,1\}^{N_f \times N_v}$ is the *assignment vector* for application i, whose $j^{th} \times n^{th}$ element is x_{ijn}. The auxiliary variables y_i are equal to 1 if at least an instance of any function has been assigned to application i, and 0 otherwise (constraints 1.6a, 1.6b, 1.7b). The index sets are defined as $\mathcal{I} \triangleq \{1,\dots,N_a\}$, $\mathcal{M} \triangleq \{1,\dots,N_\mu\}$, $\mathcal{J} \triangleq \{1,\dots,N_f\}$, $\mathcal{K} \triangleq \{1,\dots,N_\rho\}$ and $\mathcal{N} \triangleq \{1,\dots,N_v\}$. Variable ρ_{nk} represents the amount of resource k available on node n; furthermore, we denote $\boldsymbol{\rho}_n \in \mathbb{N}_n^N$ the overall capacity of node $n \in \mathcal{N}$. With $c_{jk} \in \mathbb{N}$ we capture the cost of implementation j in terms of resource k; thus, we name $\boldsymbol{c}_j \in \mathbb{N}_\rho^N$ the cost vector of implementation $j \in \mathcal{J}$. We set $\lambda_{mj} = 1$ if the abstract component m can be implemented (i.e., deployed) through j, while $\sigma_{im} = 1$ if application i needs service component m. Note how constraint (1.2) ensures that the solution is infrastructure-bounded, while constraints (1.3 and 1.4) avoid partial allocations.

The *utility function* models the overall gain $U_{ijn}(\boldsymbol{x}_i)$, i.e., the utility that the system gains by assigning $\boldsymbol{c}_j$ resources to application i, allowing it to add the implementation j to its assignment vector $\boldsymbol{x}_i$. Note that the gain does not depend merely from the given component; in fact, it depends *(i)* on the chosen implementation and *(ii)* on the node selected for the deployment. In formulating the utility function, we assume that different utilities of different applications can be normalized so that

their sum is consistent, since originally each of them may refer to arbitrary physical quantities and scales.

Please note that, as we model each application as a multiset of components, no dependencies/connections between components are considered (e.g., maximum latency that can be tolerated between a pair of components, which pair is supposed to generate more traffic, etc.). We assume that such kind of parameters should be modeled separately, as they are of competence of every single application (e.g., of its Service Defined Orchestrator). Therefore, in our approach we do not directly employ such additional parameters in the "coordination" process for the dynamic and optimal resource partitioning.

5.4 Distributed Resource AssiGnment and OrchestratioN (DRAGON)

In this section, we describe DRAGON (Distributed Resource AssiGnment and OrchestratioN), a novel approximation algorithm that we designed to solve the NP-hard Problem 1 through a distributed approach.

5.4.1 DRAGON Overview

Each application i runs a DRAGON agent, which iterates between a local and a distributed phase. Locally, the application builds its assignment vector $\boldsymbol{x}_i$, i.e., the set of component implementations to be deployed on each node. This is used to participate in a resource election process by voting resources needed on each hosting node. Each vote models the benefit that an application would gain from the resources demanded on a given node and is directly related to the application private utility. Voting and elections are performed at the node level. At first (*Orchestration Phase*), each agent performs the election locally, based on its state awareness. During an *Agreement Phase*, agents communicate and update their votes to ensure the convergence of the election process by mean of max-consensus. Applications that are "elected", i.e., that they win the distributed election, gain the right to allocate the demanded amount of (virtual) resources on a certain number of (physical) nodes[1].

Note that the assignment vector $\boldsymbol{x}_i$ of each application i does not need to be exchanged. Agents are aware of the resource demand from their peers but unaware of the details regarding which application components they wish to allocate.

To detail the algorithm, we give the following definitions:

[1]Note how this is different from several existing leader election protocols (e.g. [118], [43]) that are based on auctions, as they assume that a given item can only be assigned to a single player.

Definition 6. *(private utility function $\boldsymbol{u}_i$). Given a set $\mathcal{I}$ of applications allocating a set $\mathcal{J}$ of component implementations over a set $\mathcal{N}$ of hosting nodes, we define private utility function of application $i \in \mathcal{I}$, and we denote it with $\boldsymbol{u}_i \colon \mathcal{J} \times \mathcal{N} \to \mathbb{R}$, the utility $u_{ijn} \in \mathbb{R}$ that application i gains by adding implementation $j \in \mathcal{J}$ to its assignment vector $\boldsymbol{x}_i$ and deploying it on node $n \in \mathcal{N}$, i.e., implementing an application component on n through j.*

Each application may have a different (conflicting) objective and may have no incentive to disclose its utility; however, our model, and so our algorithm, maximizes a global objective (Equation 1.1), that in DRAGON is a policy. Since we assume that a Pareto optimality is sought, the global utility is a function of the applications private utilities, i.e.,

$$\boldsymbol{U}_i(\boldsymbol{x}_i) = f(\boldsymbol{u}_i(\boldsymbol{x}_i)),\ \forall i \in \mathcal{I}.$$

DRAGON needs a vote vector that we define as follows.

Definition 7. *(vote vector $\boldsymbol{v}^i$). Given a distributed voting process among a set $\mathcal{I}$ of N_a applications, allocating resources on a set $\mathcal{N}$ of N_v hosting nodes, we define $\boldsymbol{v}^i \in \mathbb{R}_+^{N_a N_v}$ to be the vector of current winning votes known by application $i \in \mathcal{I}$ on hosting node $n \in \mathcal{N}$. Each element $v^i_{\iota n}$ is a positive real number representing last vote of application ι on node n as known by i, if i thinks that ι is a winner of the election phase for node n. Otherwise, $v^i_{\iota n}$ is* 0.

Since applications compute resource assignments in a distributed fashion, they could possibly have different views until an agreement on the election winner(s) is reached; we use the apex i to refer to the vote vector as seen by application i at each point in the agreement process. During the algorithm description, for clarity, we omit the apex i when we refer to the local vector (the same applies also for the following vectors).

Definition 8. *(demanded resource vector $\boldsymbol{r}^i$). Given a voting process among a set $\mathcal{I}$ of N_a applications on N_ρ different types of shared resources distributed among a set $\mathcal{N}$ of N_v hosting nodes, we define as demanded resource vector $\boldsymbol{r}^i \in \mathbb{N}_+^{N_a \times N_v \times N_\rho}$, the vector of total resources currently requested by each application on every node; each element $\boldsymbol{r}^i_{\iota n} \in \mathbb{N}^{N_\rho}$ is the amount of resources requested by application $\iota \in \mathcal{I}$ on node $n \in \mathcal{N}$ with its most recent vote $v^i_{\iota n}$ known by $i \in \mathcal{I}$.*

Definition 9. *(voting time vector $\boldsymbol{t}^i$). Given a set $\mathcal{I}$ of N_a application participating to a distributed voting process over a set $\mathcal{N}$ of N_v hosting nodes, we define as voting time vector $\boldsymbol{t}^i \in \mathbb{R}_+^{N_a \times N_v}$, the vector whose element $t^i_{\iota n}$ represents the timestamp of the last vote $v^i_{\iota n}$ known by $i \in \mathcal{I}$ for application $\iota \in \mathcal{I}$ on node $i \in \mathcal{I}$.*

We also give the following definition of neighborhood:

Algorithm 1 `DRAGON` for application i at iteration t

1: `orchestration`$(\boldsymbol{v}(t-1),\ \boldsymbol{r}(t-1),\ \boldsymbol{\rho})$
2: **if** $\exists \iota \in \mathcal{I}:\ v_\iota(t) \neq v_\iota(t-1)$ **then**
3: `send`$(i',\ t),\ \forall i' \in \bar{\mathcal{I}}_i$
4: `receive`$(i',\ t),\ \forall i' \in \bar{\mathcal{I}}_i$
5: `agreement`$(i',\ t),\ \forall i' \in \bar{\mathcal{I}}_i$

Definition 10. *(neighborhood $\bar{\mathcal{I}}_i$). Given a set $\mathcal{I}$ of applications, we define neighborhood $\bar{\mathcal{I}}_i \subseteq \mathcal{I} \setminus \{i\}$ of application $i \in \mathcal{I}$, the subset of applications that can directly communicate with i.*

The notion of neighborhood is generalizable with the set of agents reachable within a given latency upper bound.

We are now ready to describe DRAGON (Algorithm 1), by detailing its two main phases.

5.4.2 Orchestration Phase

After the initialization of local vectors $\boldsymbol{v}(t)$, $\boldsymbol{r}(t)$ and $\boldsymbol{t}(t)$ for the current iteration t (Algorithm 2, line 2), each DRAGON agent uses Algorithm 2, line 8 to elect the current winners according to the known votes updated at the last iteration. If agent i has been outvoted (Algorithm 2, line 5), the algorithm starts to iterate among *(i)* an *embedding routine* (Algorithm 2, line 6), which computes the next suitable assignment vector $\boldsymbol{x}_i$ maximizing i's private utility, *(ii)* a *voting routine* (Algorithm 2, line 7) where agent i votes for the resources that follow the last computed assignment vector and *(iii)* the *election routine* (Algorithm 2, line 8), which uses votes to compute winning agents.

The iteration continues until agent i does not get outvoted anymore (Algorithm 2, line 9). This may happen if either *(i)* the selected assignment vector allows i to win the election or *(ii)* there are no more suitable assignments $\boldsymbol{x}_i$ (then no new votes have been generated).

Remark. *To guarantee convergence, DRAGON forbids outvoted applications to re-vote with a higher utility value on resources that they have lost in past rounds. Re-voting is, however, allowed only on residual resources.*

Note that an asynchronous agreement may never terminate unless we forcefully timeout the consensus process. However, we use the theory of max-consensus to show that the agreement stops as long as we have reliable communication and each vote traverses the network at least once (Section 5.5).

Algorithm 2 `orchestration` for application i at iteration t

Input: $\boldsymbol{v}(t-1),\ \boldsymbol{r}(t-1),\ \boldsymbol{t}(t-1),\ \boldsymbol{\rho},\ \boldsymbol{c}$
Output: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t)$

1: **if** $t \neq 0$ **then**
2: $\quad \boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t) = \boldsymbol{v}(t-1),\ \boldsymbol{r}(t-1),\ \boldsymbol{t}(t-1)$
3: **do**
4: $\quad \bar{v_i} = v_i(t)$
5: $\quad$ **if** $v_i(t-1) \neq 0 \ \wedge\ v_i(t) = 0$ **then** ▷ outvoted
6: $\quad\quad$ `embedding`(t) ▷ find next $\boldsymbol{x}_i$ maximizing $\boldsymbol{u}_i$
7: $\quad\quad$ `voting`$(\boldsymbol{x}_i,\ \boldsymbol{c})$ ▷ vote $\boldsymbol{x}_i$ using $\boldsymbol{U}$
8: $\quad$ `election`$(\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{\rho})$
9: **while** $\bar{v_i} \neq v_i(t)$ ▷ repeat until not outvoted

Algorithm 3 `voting` for application i at iteration t

Input: $\boldsymbol{x}_i,\ \boldsymbol{c}$
Output: $\boldsymbol{v}_i(t), \boldsymbol{r}_i(t), t_i(t)$

1: $t_i(t) = t$ ▷ vote time
2: **if** $\boldsymbol{x}_i \neq \mathbf{0}$ **then** ▷ valid assignment
3: $\quad$ **for all** $n \in \mathcal{N}$ **do**
4: $\quad\quad r_{ink}(t) = \Sigma_j x_{ijn} c_{jk},\ \forall k \in \mathcal{K}$ ▷ resources required on node
5: $\quad\quad v_{in}(t) =$ `score`$(\boldsymbol{x}_i, n)$ ▷ vote new assignment

Embedding Routine

Either during the first iteration ($t = 0$), or any time application i is outvoted, DRAGON invokes an embedding routine (Algorithm 2, line 6) that, based on the private policies of i, computes the next best suitable assignment vector $\boldsymbol{x}_i$. Therefore, this routine is in turn private for each application, and strictly dependent on the specific nature of the application itself (each of them may follow a different deployment strategy, seek optimization of specific metrics and even feature additional deployment constraints). If DRAGON is running within a Service-Defined Orchestrator (Chapter 4), when a set of operations is pending upon the execution of an action, the framework builds the proper routine and passes it to the DRAGON agent, in order to perform the desired changes under distributed agreement. For what concerns DRAGON, this routine can be viewed as a private decision process that selects, for each component needed by the application, both the implementation $j \in \mathcal{J}$ to be used and the node $n \in \mathcal{N}$ where j should be deployed.

Voting Routine

After a new assignment vector has been built, each DRAGON agent executes a voting routine, updating the time of its most recent vote; if the assignment vector is valid, all demanded resources are updated and voted, through a *score function* derived from the global utility (Algorithm 3). Since voting is performed at node

Algorithm 4 `election` for application i at iteration t

Input: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{\rho}$
Output: $\boldsymbol{v}(t)$

1: **do**
2: **for all** $n \in \mathcal{N}$ **do**
3: $\mathcal{W}_n = \texttt{node\_election}(\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ n,\ \boldsymbol{\rho}_n)$
4: $\mathcal{W}^F = \texttt{election\_recount}(\mathcal{W}_n\ \forall n)$ ▷ detect false-winners
5: $\boldsymbol{v}_\iota = \mathbf{0},\ \forall \iota \in \mathcal{W}^F$ ▷ reset false-winners votes
6: **while** $\mathcal{W}^F \neq \varnothing$ ▷ repeat until no false-winners are detected
7: $\boldsymbol{v}_\iota = \mathbf{0},\ \forall \iota \in \mathcal{I} \setminus \bigcup_{n \in \mathcal{N}} \mathcal{W}_n$ ▷ reset votes that did not win

Algorithm 5 `node_election` on node n at iteration t

Input: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ n,\ \boldsymbol{\rho}_n$
Output: $\mathcal{W}_n$

1: $\bar{\boldsymbol{\rho}}_n = \boldsymbol{\rho}_n$ ▷ residual resources
2: $\mathcal{W}_n = \varnothing$ ▷ winner set
3: **do**
4: $\mathcal{I}_b = \{i \in \mathcal{I} |\ r_{ink}(t) \leq \bar{\rho}_{nk}, \forall k \in \mathcal{K}\}$ ▷ valid candidates
5: $\omega = \arg\max_{i \in \mathcal{I}_b \setminus \mathcal{W}} \left\{ \frac{v_{in}(t)}{\|\boldsymbol{r}_{in}(t)\|} \right\}$ ▷ candidate with the highest vote
6: $\mathcal{W}_n = \mathcal{W}_n \cup \{\omega\}$ ▷ add to winners
7: $\bar{\rho}_{nk} = \bar{\rho}_{nk} - r_{\omega nk},\ \forall k \in \mathcal{K}$ ▷ decrease residual resources
8: **while** $\mathcal{I}_b \setminus \mathcal{W} \neq \varnothing$ ▷ repeat until no candidate remains

level, this routine generates a vote for each hosting node involved in the current assignment $\boldsymbol{x}_i$. Although the raw global utility itself may be used as score function to compute votes, in Section 5.4.4 we give recommendations on which function should be used to guarantee convergence and optimal approximation bound (Section 5.5). Since the value of t_i is updated in any case (Algorithm 3, line 1), if application i does not find any suitable assignment vector, the recent timestamp associated with an empty vote will let its peers know that i agrees with an election definitively lost.

Election Routine

The last step of the *Orchestration Phase* (Algorithm 2, line 8) is a resource election that decides which applications are capable of allocating the demanded resources on the chosen hosting nodes (Algorithm 4). Based on the most recent known votes $\boldsymbol{v}(t)$, the related resource demands $\boldsymbol{r}(t)$ and the capacity $\boldsymbol{\rho}_n$ of each node, this procedure selects applications by means of a greedy approach. For every node $n \in \mathcal{N}$ (Algorithm 4, line 3-4), the `node_election` subroutine (Algorithm 5) *(i)* discards every application whose demanded resources $\boldsymbol{r}_i$ exceed the residual node capacity and *(ii)* selects the one with the highest ratio vote to demanded resources (Algorithm 5, lines 4-5). The one elected is then added to the winner set of that particular node and the amount of resources assigned to the new winner is

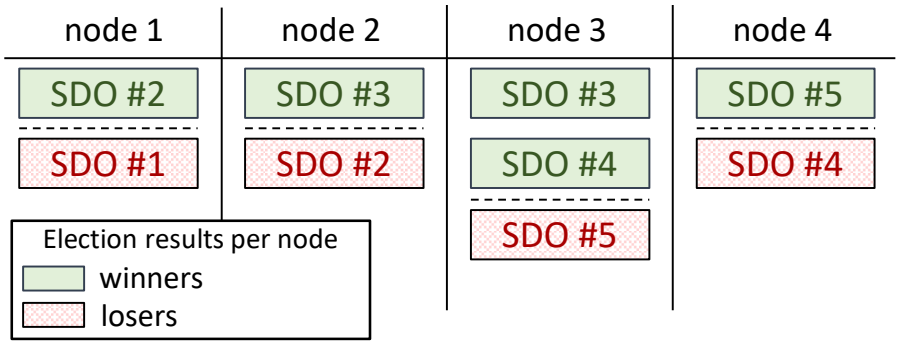


Figure 5.1: Example of false winners after an election routine: SDO #2 prevents #1 to allocate needed resources on node 1, although #2 cannot be deployed, since it lost elections on node 2.

removed from the residual ones (Algorithm 5, lines 6-7). The greedy election on each node ends when either all candidates result winners, or residual node resources are not enough for any of those remaining.

In Section 5.5 we show that the greedy heuristic gives guarantees on the optimal approximation.

Remark. *In DRAGON an assignment $\boldsymbol{x}_i$ is considered valid only if application i wins all elections on each node n involved in the assignment $\boldsymbol{x}_i$. If any node election is lost, DRAGON resets the vote vector and a new assignment is built from scratch to avoid suboptimal assignments.*

Since elections are performed separately for each node of the infrastructure, the election routine includes a conflict resolution subroutine named *election-recount* (Algorithm 4, line 4), which handles potential suboptimality deriving as a result of the election process. Let us consider the assignment scenario in Figure 5.1; most resources of node 1 have been assigned to application #2, thus preventing the deployment of application #1; however, having #2 lost the election on node 2, releases its previous vote on node 1 at the next iteration. Therefore, app. #1 could be considered a winner.

The election-recount subroutine copes with this problem by identifying which applications should be removed from the election so that the solution is optimized. We call these applications *false-winners*, i.e., applications that only won a subset of the needed nodes, preventing peers that would maximize the global utility to win. False-winners are identified recursively. Given a potential false-winner ω, i.e., it lost elections on a subset of needed nodes, the idea is to temporarily extend residual resources on these nodes, whether that amount of resources have been assigned to other false-winners during the previous election round. If the extended residual resources are still in deficit with respect to the resources that ω needs, then the candidate is a false-winner as well.

For instance, let's consider the election results in Figure 5.1. To determine if application #2, that is a winner for node 1, is valid or not, we check if it is possible

to free some resources on node 2, where it lost. On that node, the only winner is application #3; however, it is a valid one, since it has not lost any other needed node. Therefore, application #2 definitely lost node 2, and can be removed from the winners of node 1 (it is a false-winner), thus enabling application #1 to win the elections. Situations in which some Applications *cross-lose* nodes are resolved in favor of the one whose vote on any node is the highest; see e.g., Figure 5.1: applications #4 and #5 *cross-lost* nodes 3 and 4.

The election process is repeated until the recount subroutine does not detect any false-winners (Algorithm 4, line 6). When the election result is confirmed, votes of applications that did not win the election are reset (Algorithm 4, line 7).

5.4.3 Agreement Phase

Once vectors $\boldsymbol{v}^{i'}$, $\boldsymbol{r}^{i'}$ and $\boldsymbol{t}^{i'}$ are received from every neighbor i', each agent runs an Agreement Phase. During this phase, applications make use of a consensus mechanism to reach an agreement on their vote vector $\boldsymbol{v}^i$, hence on the overall resources assignment (Algorithm 6). By adapting the definition of *consensus* [103] to the application-resources assignment problem, we define our own notion of consensus on the election results as follows:

Definition 11. *(election-consensus). Let us consider a set $\mathcal{I}$ of N_a applications sharing a computing edge infrastructure through an election routine driven by, for each application $i \in \mathcal{I}$, the vote vector $\boldsymbol{v}^i(t) \in \mathbb{R}_+^{N_o \times N_\upsilon}$, the demanded resource vector $\boldsymbol{r}^i(t) \in \mathbb{R}_+^{N_a \times N_\upsilon \times N_\rho}$ and the voting time vector $\boldsymbol{t}^i(t) \in \mathbb{N}^{N_a \times N_\upsilon}$. Let $e : \mathbb{R}_+^{N_a \times N_\upsilon}, \mathbb{N}^{N_a \times N_\upsilon \times N_\rho} \to 2^{\mathcal{I}}$ be the election function, that given a vote vector $\boldsymbol{v}$ and the demanded resources $\boldsymbol{r}$ gives a set of winners. Given the consensus algorithm for application i at iteration $t+1$, $\forall \iota \in \mathcal{I}$,*

$$\begin{aligned} v_\iota^i(t+1) = v_\iota^{i'}(t),\ \boldsymbol{r}_\iota^i(t+1) = \boldsymbol{r}_\iota^{i'}(t), \\ \text{with } i' = \arg\max_{i' \in \bar{\mathcal{I}}_i \cup \{i\}} \{t_\iota^{i'}(t)\}, \end{aligned} \tag{5.2}$$

election-consensus *among the applications is said to be achieved if $\exists \bar{t} \in \mathbb{N}$ such that, $\forall t \geq \bar{t}$ and $\forall i, i' \in \mathcal{I}$,*

$$\begin{cases} e(\boldsymbol{v}^i(t), \boldsymbol{r}^i(t)) \equiv e(\boldsymbol{v}^{i'}(t), \boldsymbol{r}^{i'}(t)) \\ \boldsymbol{v}_\iota^i(t) \neq \boldsymbol{0} \iff \iota \in e(\boldsymbol{v}^i(t)),\ \forall \iota \in \mathcal{I}, \end{cases} \tag{5.3}$$

i.e., on all applications the election function computes the same winner set and only winner votes are non zero.

The agreement on votes of an application ι is performed by every application i once received vectors $\boldsymbol{v}^{i'}$, $\boldsymbol{r}^{i'}$ and $\boldsymbol{t}^{i'}$ from each i' in its neighborhood, comparing them and selecting the most recent information received, if any (Equation 5.2).

Algorithm 6 `agreement` with SDO i' at iteration t

Input: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t),\ \boldsymbol{v}^{i'}(t),\ \boldsymbol{r}^{i'}(t),\ \boldsymbol{t}^{i'}(t)$
Output: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t)$

for all $\iota \in \mathcal{I}$ **do**
  for all $n \in \mathcal{N}$ **do** ▷ for every hosting node
    if $t_{\iota n}(t) < t^{i'}_{\iota n}(t)$ **then** ▷ the vote is new
      $v_{\iota n}(t) = v^{i'}_{\iota n}(t)$
      $r_{\iota nk}(t) = r^{i'}_{\iota nk}(t),\ \forall k \in \mathcal{K}$
      $t_{\iota n}(t) = t^{i'}_{\iota n}(t)$

Since DRAGON is asynchronous by design, at each iteration t the agreement phase can start even if agents have received vote messages from only a subset of their neighbors.

5.4.4 Recommendations on the score function

DRAGON's score function is a policy. Many policies may work well in practice, but in some cases they may lead to arbitrarily bad performance. As we will see in the next section, DRAGON guarantees both convergence and a given performance lower bound as long as the function maximized during the election routine is submodular (Definition 12). In this section we give recommendation on the score function $\mathcal{V}$ that each application should use during the voting routine described in Algorithm 3 to satisfy this property. Analytic results are shown in the next section.

Let $\mathcal{U}_{in}(\boldsymbol{x}_i) = \Sigma_j U_{ijn}(\boldsymbol{x}_i) x_{ijn}$ be the overall node utility of application i on node n. To guarantee convergence of the election process, we let each peer i communicate its vote on node n obtained from the score function:

$$\mathcal{V}_i(\boldsymbol{x}_i, \mathcal{W}_n, n) = \min_{\omega \in \mathcal{W}_n} \{\mathcal{U}_{in}(\boldsymbol{x}_i), \mathcal{S}_{in}(\omega)\}, \tag{5.4}$$

where $\mathcal{W}_n \subseteq \mathcal{I}$ is the current winner set for node n, i.e., $v_{\omega n}(t) \neq 0\ \forall \omega \in \mathcal{W}_n$, and $\mathcal{S}_{in}$ is defined as

$$\mathcal{S}_{in}(\omega) = \begin{cases} +\infty & \text{if } i \text{ never voted on } n, \\ \|\boldsymbol{r}_{in}(t)\| \frac{v_{\omega n}(t)}{\|\boldsymbol{r}_{\omega n}(t)\|} & \text{otherwise.} \end{cases}$$

Since $\mathcal{U}_{in}(\boldsymbol{x}_i) \geq 0$ by definition, if i computes each vote with the function $\mathcal{V}$, it follows that, $\forall (i, n) \in \mathcal{I} \times \mathcal{N},\ \mathcal{V}_i(\boldsymbol{x}_i, n) \geq 0$. Note how, if it is not the first time that i votes on n, the vote $v_{in}(t)$ generated at iteration t never results as an outvote of any application that has been previously elected on node n, during the election process described in Algorithm 5.

5.5 Convergence and Performance Guarantees

In this section, we present results on the convergence properties of our DRAGON distributed approximation algorithm. As in Definition 11, by convergence we mean that a valid solution to the applications-resources assignment problem is found in a finite number of steps. Moreover, starting from well-known results on submodular functions, in this section we show that DRAGON guarantees an $(1 - e^{-1})$-approximation bound, and that this bound is also optimal, i.e. there is no better guarantee, unless $NP \subseteq DTIME(n^{O(\log \log n)})$.

Note that, if (5.4) is used as score function, the election routine of DRAGON is equivalent to a greedy algorithm attempting to find, for each node n, the set of winner applications $\mathcal{W}_n \subseteq \mathcal{I}$ such that the set function $z_n : 2^{\mathcal{I}} \to \mathbb{R}$, defined as

$$z_n(\mathcal{W}_n) = \sum_{\omega \in \mathcal{W}_n} \mathcal{V}_\omega(\boldsymbol{x}_\omega, \mathcal{W}_n, n), \tag{5.5}$$

is maximized. By construction of $\mathcal{V}$, we have that z_n is monotonically non-decreasing and $z(\varnothing) = 0$.

Definition 12. *(submodular function). A set function $z : 2^{\mathcal{I}} \to \mathbb{R}$ is submodular if and only if, $\forall \iota \notin \mathcal{W}' \subset \mathcal{W}'' \subseteq \mathcal{I}$,*

$$z(\mathcal{W}'' \cup \{\iota\}) - z(\mathcal{W}'') \leq z(\mathcal{W}' \cup \{\iota\}) - z(\mathcal{W}'). \tag{5.6}$$

This means that the marginal utility of adding ι to the input set, cannot increase due to the presence of additional elements. Next we show that the total score z_n (5.5) is submodular. Our intuition behind its submodularity is that the score function $\mathcal{V}_n$ can, at most, decrease due to the presence of additional elements in $\mathcal{W}_n$. Formally, we have:

Lemma 5.5.1. *z_n (5.5) is submodular.*

Proof. Since $\mathcal{W}'_n \subset \mathcal{W}''_n$, we have

$$\min_{\omega \in \mathcal{W}''_n} \left\{ \|\boldsymbol{r}_{\iota n}(t)\| \frac{v_{\omega n}(t)}{\|\boldsymbol{r}_{\omega n}(t)\|} \right\} \leq \min_{\omega \in \mathcal{W}'_n} \left\{ \|\boldsymbol{r}_{\iota n}(t)\| \frac{v_{\omega n}(t)}{\|\boldsymbol{r}_{\omega n}(t)\|} \right\},$$

and so, for (5.4),

$$\mathcal{V}_\iota(\boldsymbol{x}_i, \mathcal{W}''_n, n) \leq \mathcal{V}_\iota(\boldsymbol{x}_i, \mathcal{W}'_n, n). \tag{5.7}$$

By definition of z_n, the marginal gain of adding ι to $\mathcal{W}_n$ is

$$z_n(\mathcal{W}_n \cup \{\iota\}) - z_n(\mathcal{W}_n) = \mathcal{V}_\iota(\boldsymbol{x}_i, \mathcal{W}_n, n), \forall \iota \notin \mathcal{W}_n \subseteq \mathcal{I},$$

therefore, substituting in (5.7), we have the claim. □

Convergence Guarantees

A necessary condition for convergence in DRAGON is that all applications are aware of which are the winning votes for a hosting node. This information needs to traverse all applications in the communication network (at least) once. Theorem 5.5.2 shows that a single information traversal is also sufficient for convergence.

The communication network of a set of applications $\mathcal{I}$ is modeled as an undirected graph, with unitary length edges between each couple $i', i'' \in \mathcal{I}$ such that $i'' \in \bar{\mathcal{I}}_{i'}$ and $i' \in \bar{\mathcal{I}}_{i''}$, being $\bar{\mathcal{I}}_{i'} \subseteq \mathcal{I} \setminus \{i'\}$ and $\bar{\mathcal{I}}_{i''} \subseteq \mathcal{I} \setminus \{i''\}$ respectively the neighborhoods of i' and i''.

Theorem 5.5.2. *(Convergence of synchronous DRAGON). Consider an infrastructure of N_υ hosting nodes, whose resources are shared among N_a applications through an election process with synchronized conflict resolution over a communication network with diameter D. If the communications occur over a reliable channel and the function (5.5) maximized during the election routine is submodular, then DRAGON needs at most $N_a^2 N_\upsilon D$ iterations to converge.*

Proof. We first show by induction that agents agree on the first k assignments in at most kN_aD iterations. Given the submodularity of z_n, the assignment $(i_1^\star, n_1^\star)$ with the highest vote computed at iteration 1 can be outvoted at most $N_a - 1$ times, i.e., until every agents voted on node $n_1^\star$ at least once. Since each time D iterations are needed to propagate the vote, every agent will have agreed on the highest vote $v_{i_1^\star n_1^\star}$ at most after N_aD iterations. Let us suppose that at iteration hN_aD all agents agree on the first k-best assignments. Since the next-best vote propagated at iteration $k+1$ can be outvoted at most $N_a - 1$ times, it follows that every agent will have agreed on $(i_{h+1}^\star, n_{h+1}^\star)$ by iteration $hN_aD + N_aD$. Then, together with $(i_1^\star, n_1^\star)$ being agreed to at N_aD, every agent will have agreed on $(i_k^\star, n_k^\star)$ within kN_aD iterations. In DRAGON each compute node may be assigned to each application, then, in the worst case there is a combination of N_aN_υ assignments. Therefore, agents reach agreement in at most $N_a^2 N_\upsilon D$ iterations. □

As a direct corollary of Theorem 5.5.2, we compute a bound on the number of messages that applications have to exchange in order to reach an agreement on resource assignments. Because we only need to traverse the communication network at most once for each combination applications per hosting nodes $(i, n) \in \mathcal{I} \times \mathcal{N}$, the following result holds:

Corollary 5.5.2.1. *(DRAGON Communication Overhead). The number of messages exchanged to reach an agreement on the resource assignment of N_υ nodes among N_a non-failing applications with reliable delay-tolerant channels using the DRAGON algorithm is at most $N_{msp} N_a^2 N_\upsilon D$, where D is the diameter of the communication network and N_{msp} is the number of links in its minimum spanning tree.*

Performance Guarantees

The election routine in DRAGON is trivially extended with partial enumeration [88], leading to the following two results (for brevity, the extension has been omitted in Algorithm 4).

Theorem 5.5.3. *(DRAGON Approximation Bound). DRAGON extended with partial enumeration yields an* $(1 - e^{-1})$*-approximation bound with respect to the optimal assignment.*

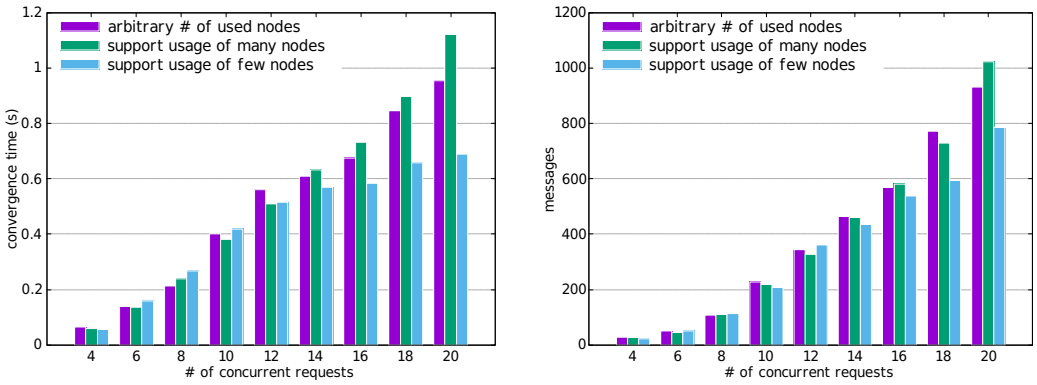
Proof. (sketch) During the election routine, DRAGON uses a greedy heuristic to assign node resources to a set of winners $\mathcal{W}_n$. The objective of the heuristic is to maximize the value of the set function $z_n(\mathcal{W}_n)$ without exceeding the node capacity (knapsack constraint). From a recent result on submodular functions [156], we know that a greedy approximation algorithm used to maximize a non-decreasing submodular set function subject to a knapsack constraint is bounded by $(1 - e^{-1})$ if the algorithm is combined with the enumeration technique due to [88]. Being the set function $z_n(\mathcal{W}_n)$ positive, monotone and non-decreasing, it remains to show that the overall utility maximized by DRAGON is submodular, which comes from Lemma 5.5.1; hence the claim holds. □

Theorem 5.5.4. *(DRAGON Approximation Optimality). The DRAGON approximation bound of* $(1 - e^{-1})$ *is optimal, unless* $NP \subseteq DTIME(n^{O(\log \log n)})$.

Proof. (sketch) To show that the approximation bound given by DRAGON is optimal, we first show that the applications-resources assignment problem addressed by DRAGON can be reduced from the (NP-hard) *budgeted maximum coverage problem* [88]. Given a collection S of sets with associated costs defined over a domain of weighted elements, and a budget L, find a subset $S' \subseteq S$ such that the total cost of sets in S' does not exceeds L, and the total weight of elements covered by S' is maximized. We reduce the applications-resources assignment problem from the budgeted maximum coverage problem by considering *(i)* S to be the collection of all the possible set of applications, i.e., $S = 2^{\mathcal{I}}$, *(ii)* L to be the total amount of resources available on the hosting node (in this particular case $N_\rho = 1$), and *(iii)* weights and costs to be, respectively, votes and demanded resources of each application. Since [88] shows that $(1 - e^{-1})$ is the best approximation bound for the budgeted maximum coverage problem unless $NP \subseteq DTIME(n^{O(\log \log n)})$, the claim holds. □

5.6 Experimental Results

To validate the approach presented in this paper, we implemented a prototype of DRAGON, available at [33]. Our evaluation focuses on assessing both DRAGON's



(a) Convergence time.

(b) Number of exchanged messages.

Figure 5.2: Convergence evaluation of DRAGON for different system policies.

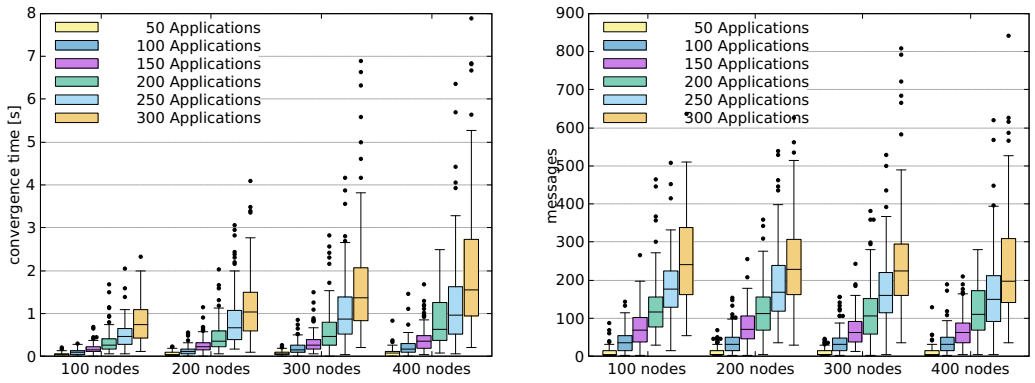
asynchronous convergence properties and performance. We deployed our prototype on an environment with 4 physical nodes (deployed on the CloudLab distributed research infrastructure [134]), each with a different amount of computing resources (CPU, memory, and storage). We run 6 diverse application components, whose implementation can be chosen among 9 different options; on average, each implementation uses about 13% of a node capacity. These numbers, combined with the rest of our parameter space, allowed us to test the behavior of the algorithm when the hosting resources are saturated, even running a moderate number of applications. All tests have been repeated varying the number of concurrent applications. To test scalability, we also run larger experiments by deploying our prototype on a simulated environment.

Convergence Evaluation

DRAGON convergence properties have been evaluated by measuring the time needed to reach consensus and the total number of messages exchanged. To stress the convergence of the algorithm, we evaluated it when up to 20 allocation requests arrive simultaneously.

Figure 5.2 shows our results comparing three system policies: *(i)* components of an application are preferably allocated on the lowest number of nodes; *(ii)* components of an application are spread across as many nodes as possible; *(iii)* no preference on the number of nodes is given. For each configuration, we ran 25 instances, gradually varying the average number of services per application (with averages from 2.4 to 3.6 services). Plots show mean values; all confidence intervals (not shown) were statistically significant.

In particular, Figure 5.2a shows the mean convergence times. We found that, when a large number of applications interact, encouraging the system to use fewer nodes significantly lowers convergence time. Some consequences of this policy are *(i)*



(a) Convergence times.

(b) Number of exchanged messages.

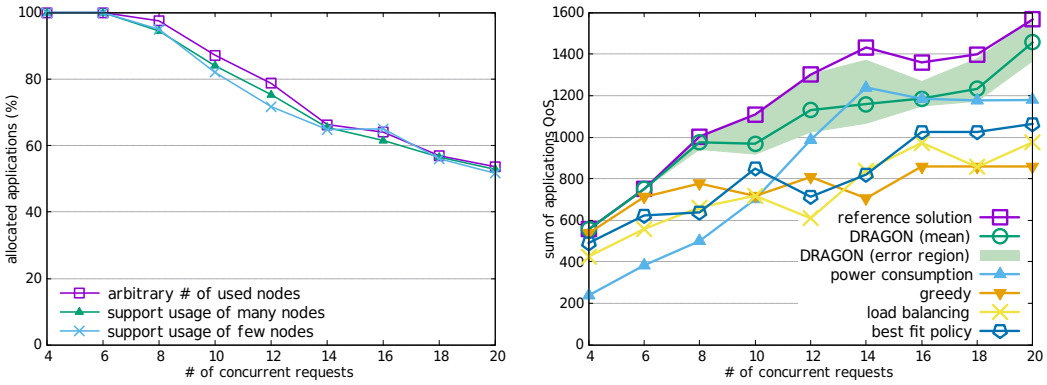
Figure 5.3: DRAGON convergence evaluation on large scale simulation varying the number of concurrent applications and available hosting nodes, where resource allocation requests are randomly performed over time.

a reduced probability to lose a node election and *(ii)* re-voting on residual resources located on additional nodes is discouraged. Hence, the highest convergence times have been registered enforcing the usage of many nodes, while convergence is slightly faster when applications are free to arbitrarily decide the number of nodes to use. This pattern is more evident for a number of applications greater than 10.

The total number of exchanged messages follows a similar behavior (Figure 5.2b). However, in this case the previous trend is not marked as for convergence times. This means that changing this policy does not seem to significantly impact the number of messages that DRAGON needs to exchange to reach convergence.

Other than offline deployment, we also evaluated online convergence on a large scale simulation, where an increasing number of applications demand resources over time (Figure 5.3ab). The convergence is evaluated on the variation of the number of applications and hosting nodes. Figure 5.3a shows that the number of concurrent applications affects convergence times more than the number of available nodes. Although this result is expected (see Theorem 5.5.2), the increase of processing time may be partially due to the limited number of physical CPUs (the simulation runs on an i7-4770 CPU @ 3.40GHz, where each DRAGON agent is a separate process).

Figure 5.3b shows that increasing the number of available nodes does not introduce noticeable variations on the total number of exchanged messages. This result suggests that the number of steps DRAGON requires to converge (hence also the number of exchanged messages) does not significantly depend on the number of nodes in the problem. Since, instead, Figure 5.3a highlights that convergence times increase, we can conclude that what changes is the duration of every iteration, as more nodes need to be processed.



(a) Percentage of allocated applications.

(b) Sum of deployed applications QoS.

Figure 5.4: Performance evaluation of DRAGON comparing (a) different system utilities and (b) DRAGON solutions against *(i)* three one-size fits-all common approaches and *(ii)* a reference solution obtained running a centralized solver.

Performance Evaluation

Figure 5.4a compares DRAGON performance for the same three system policies previously introduced. The plot shows the percentage of applications successfully deployed after the distributed assignment process. We found that, when the number of concurrent applications stays below 8, all requests are allocated, since the overall resources demand (considering average size functions) is bounded by the total amount of available resources. Above that threshold, all analyzed policies achieve approximately the same average allocation ratio, with the exception of the "few-nodes-policy", whose allocation ratio is lower for less than 12 applications, although it shows the fastest convergence time (Figure 5.2). This is because, when resources on the already used nodes terminate, this policy discourages the usage of residual resources available on other nodes. However, this disadvantage disappears as the number of applications grows, since the system implicitly introduces more allocation options. This result suggests that DRAGON allocation ratio scales well with the application concurrency regardless of the system policy.

Finally, to evaluate our performance in practice we compared DRAGON with traditional orchestration approaches. In particular, we compare against three one-size fits-all allocation policies, i.e., a centralized orchestrator that uses the same objective function to optimize the deployment of every application. The evaluated optimization policies are as follows: *(i)* minimization of total *power consumption*, *(ii) greedy* selection of the potentially best performing component implementations, *(iii) load balancing* among nodes. Figure 5.4b also shows the performance obtained by switching between these three policies based on which one *fits best* the needs of each application. Additionally, we plot the *reference solutions*, obtained by running a centralized solver to Problem 1. Values obtained with this experiment set have been used as reference to evaluate the other approaches.

Figure 5.4b compares solutions in terms of *overall Quality of Service*, i.e., the sum of the QoS obtained by each application successfully deployed[2]. Varying the number of concurrent applications, for each configuration we ran DRAGON multiple times. Results are shown with a 95% confidence interval. Centralized algorithms always give the same solution. Results show that deploying each application according to its own objective through DRAGON provides a considerably higher QoS compared to one-size fits-all approaches, despite DRAGON being a distributed algorithm. In particular, for less than 8 concurrent requests, i.e., before resources start to run out, DRAGON is always equivalent to the reference solution, considered as optimal. For a higher number of requests (and thus, of distributed instances), as expected, the mean QoS departs from the optimal. However, the total QoS continues to grow, following the trend of the reference solution. This result suggests that DRAGON effectively prefers the deployment applications that introduce higher utilities to the overall solution.

Other findings from Figure 5.4b are summarized as follows. *(i)* A common objective that minimizes power consumption provides poor total QoS, except for a high number of applications, since this strategy accommodates the largest number of requests. *(ii)* Greedly selecting the best performing implementation provides high values of overall QoS only when there are few applications. Finally, *(iii)* switching among different common strategies based on the one that fits best each application does not necessarily provide a higher QoS. This is because some generic allocation strategies work well when they are applied to the whole set of applications (e.g., load balancing and power minimization). Noticeable, none of the one-size fits-all approaches is able to increase the overall QoS after resources are saturated.

5.7 Conclusion

In this chapter, we complemented the work on Service-Defined Orchestrations by proposing DRAGON, a time-bounded distributed approximation algorithm that solves the problem of optimally partitioning a pool of resources between multiple edge applications (hence, multiple Service-Defined Orchestrators). DRAGON allows such applications to coexist over a shared infrastructure by means of a dynamic agreement on how resources have to be (temporary) assigned to the existing application. We used linear programming to define and model the application-resources assignment problem, that DRAGON solves in a distributed fashion providing guarantees on both convergence time and performance. Our evaluation assesses convergence and performance properties, comparing different policies of our system and showing benefits over conventional orchestration approaches. Together with

[2]The QoS of each application have been modeled through its private utility. Values have been normalized between 0 and 100 for each physical function.

Service-Defined Orchestration, DRAGON enhances the capabilities of Edge Infrastructures; indeed, an infrastructure provider may enable their customers (i.e., service providers) to deploy applications through the preferred embedding algorithm and orchestration strategies, without the restrictions deriving by relying on a conventional one-size-fits-all orchestrator.

Chapter 6

Configuring Services in Highly Modular Environments: a Model-Based Solution

With the transformation introduced by Edge Computing, the highly modular and distributed infrastructure becomes populated with new arbitrary facilities (e.g., application components, network functions, SDN applications, and more), which are generally exploited by service providers to compose the final service. In this process, service providers may require a mechanism to access the run-time state of microservices below (e.g., NAT, traffic monitors), monitoring them and potentially modify their configuration to deliver the proper service composition, possibly through a uniform API. Unfortunately, most of existing microservice facilities feature ad-hoc interfaces, while many other are not even designed by taking into account the possibility to be used as part of higher-level service workflows, hence not providing an adequate interface that would allow overarching services to exploit their features. In this chapter, we address this problem by proposing an approach to allow access to the run-time state of arbitrary components by mean of high-level model-based structures. In particular, we focus the use case of SDN applications, which are available on modern network controllers and usually lack of interoperability and configurable interfaces, although they could be exploited to compose richer services (e.g., by service-layer orchestrators). The mapping from the high-level data model to the actual data representation within the SDN application is enabled by a suite of algorithms that are generic enough to operate independently of the actual source code of the application, thus avoiding undesired and invasive modifications to existing services.

The work presented in this chapter has been first published in [40].

6.1 Introduction

New generation programmable networks require enough flexibility to cope with the emerging communication requirements (e.g., QoS, security) of modern high-level services (e.g., Smart City, Cloud Robotics) operating on top of a distributed and dynamical environment of networked smart systems (e.g., sensors, smart terminals, and buildings). These smart environments require punctual and reactive network and service management operations to adapt systems to context changes and to assure and preserve proper levels of user experience [124].

The solutions discussed in the previous chapters enable the use of multiple infrastructure components in the scope of composite high-level workflows. This can lead to the delivery of richer added-value applications enabled by control and coordination tasks performed by higher-level service control applications (that we shorten in ServiceApps). For instance, Service-Defined Orchestrators (Chapter 4) are in charge of collecting context information and taking coordinated actions in order to handle service deployment and prevent/recover from service corruption or degradations. Similarly, an access control system extended with anomaly detection in resource usage [32] may retrieve data throughput on a per-user flow basis by leveraging a traffic monitoring analytics tool available on board of an SDN controller, leveraging a traffic shaper in case a misuse is detected.

In general, a ServiceApp, while composing the final workflow, may need to actively interact with diverse lower-level components both to collect their current run-time state and to modify previously established configurations. Unfortunately, many of the existing low-level facilities are not designed with the possibility to be used as part of higher-level service workflows. This problem is particularly evident in the case of Software-Defined Networks (SDN), where SDN applications (shortened in SDNApps) perform arbitrary network-related tasks such as traffic monitoring, firewall, NAT, deep packet inspection. In fact, SDNApps generally only offer a partial view of their run-time state, which is kept in completely arbitrary structures and exposed (if any) through custom APIs. This heterogeneity in SDN application may prevent ServiceApps from taking effective run-time decisions based on the current context of the underlying network layer, or at best, makes their implementation more difficult and less flexible.

To cope with the aforementioned challenges, in this chapter we present an abstraction layer enabling ServiceApps to dynamically inspect the run-time state and/or change the current configuration of any SDNApp based on implementation-agnostic data-models, also enabling ServiceApps to promptly react to any event occurring in the controlled infrastructure. We design our system to also allow the integration of components that were not originally engineered to export their data to an external consumer (e.g., service orchestrators).

More specifically, our contribution is twofold.

First, we propose an overarching software architecture that enables novel ServiceApps to leverage run-time state and configuration of multiple infrastructure-specific components (SDNApps), with the aim of creating complex workflows spanning across multiple infrastructure domains (e.g., possibly managed with different SDN controllers), and independent from the actual implementation of any given network application. The proposed architecture leverages an abstraction layer based on a YANG [23] data model associated with each SDNApp, which describes its configuration and run-time state in an implementation-independent way, and a set of application-agnostic high-level APIs that enable an external module (i.e., ServiceApp) to access and monitor such data.

Second, we define a set of application-agnostic *mapping algorithms* that map incoming requests for a specific data, specified according to the high-level YANG abstraction, into a read/write operation of the actual run-time internal variable(s) of the SDNApp. These mapping algorithms are generic enough to be independent from the specific SDNApp logic (and source code). Moreover, they can be put into operation without requiring the application developer to create a specific code to provide access to the selected data.

We evaluate the proposed algorithms in terms of execution time of both read and write operations, as well as notification latency. Moreover, we assess the performance of our approach under specific use cases and evaluate its overhead comparing with direct access to the application variables. Results show that the execution of the proposed software framework introduces a relatively low overhead and provides insights on how to optimize notification performance. Finally, we also provide a discussion on the advantages of the proposed approach against the requirements SDNApps should fulfill to be part of the proposed software framework.

This chapter is structured as follows. Section 6.2 analyzes the related work. Section 6.3 presents some use cases that highlight possible deployments where the solution presented in this chapter can be adopted. Section 6.4 presents the overall software architecture. Section 6.5 describes the algorithms that transparently map the SDNApps variables in a YANG-based structure. Validation, including both mapping algorithms and the complete architecture, is carried out in Section 6.6. Finally, Section 6.7 discusses the main lessons learned while prototyping and validating this approach, and Section 6.8 concludes the chapter.

6.2 Related Works

A number of recent works address the management of the state of SDN applications [21, 27, 71]. In particular, OpenState [21] and P4 [27] propose modern data planes that allow instantiating stateful flow rules whose output may change based on previously processed traffic. In [71], instead, authors address the problem of consistency between the control plane and a stateful data plane. However, these works

focus on the limitation of standard SDN data planes (usually OpenFlow) rather than considering the problem of exposing the application state to an external and not pre-defined, service.

The Open Daylight controller [108] features a Model-Driven Service Adaptation Layer (MD-SAL), also available in other proprietary software such as Cisco Network Services Orchestrator [44], which employs user-defined YANG models to provide messaging and data storage functionality that simplifies the development of new SDN applications. The approach we propose in this chapter has a different scope, being applied at a different level, i.e., to export the features of existing SDN applications to services outside the controller domain.

The problem of handling the run-time state of an application is also tackled in other fields different from SDN. For instance, works such as [130, 129, 65] investigate the problem of automatic Virtual Network Functions (VNFs) scaling and/or creating (either hot or cold) standby copies of a given network service instance. More specifically, Pico [129] proposes an approach to move and/or duplicate portions of state among replicas; state is managed in a flow-centric way through FreeFlow [130], which models the state of a VNF as a flow table (where each line is usually identified by the TCP/IP five-tuple) and requires a per-VNF agent able to *get*, *put* and *migrate* flows. OpenNF [65] defines a northbound interface oriented to the use case of moving state between multiple instances. Although authors mention that some VNFs may not provide access to their run-time state, they do not focus this problem, In general, [130, 129, 65] address the problem of moving the state on new VNFs instances that are *analogous* to the original one, omitting the case in which the state has to be shared among applications that feature different purposes. Moreover, these works are oriented to the particular case of a flow-based state (that is typical in middle-boxes), i.e., the state can always be modeled as a table where a row (or a set of) concerns a precise traffic flow. By contrast, we provide an approach to model any state of the application, not only flow-based ones. Finally, we focus on SDN applications that, being software bundles running *within* the environment an SDN controller (e.g., exploiting the OSGi technology), they may have implementation constraints that prevent from trivially exposing the desired configuration interface through ad-hoc agents.

In the area of service orchestration, a number of research works are devoted to creating integrated workflows across application and network layers. The EU H2020 SELFNET project [116] focuses on an autonomic network management framework based on the Self-Organized Networks (SON) paradigm to significantly reduce operational costs and improving user experience. More specifically, the authors propose the deployment of a set of sensors to dynamically identify possible issues that are then solved by a set of actuators. While such an approach fits nicely with the final goal of keeping QoS under control, it may not be suitable for a broader scope, such as to enable generic services and applications to communicate and share arbitrary information. With this respect, our work is a first, partial implementation of a

possible 5G Operating System (5GOS) first presented in [107], which extends to a distributed network infrastructure the concept of "everything as a service" [125].

Ultimately, the analysis of the state of the art reveals a lack of solutions to enable SDNApps to effectively share their internal run-time state and to communicate with upper-layer functionalities (i.e., network services or application-layer orchestrators) and, thus, to be part of composite and dynamically-established workflows. In this direction, this work proposes a software architecture with an abstraction layer to enable the real-time exposition and modification of SDNApps internal variables, where data are exported according to high-level model-based structures. The mapping from the high-level data model to the actual data representation within the SDNApps is enabled by a suite of algorithms that are generic and independent from the specific source code of the SDNApp, thus avoiding undesired and invasive modifications to existing applications.

6.3 Use Cases

The capability to collect the run-time state of a running SDNApp and modify it transparently at run-time (either by issuing the proper configuration commands or by crafting its state, if needed) brings important advantages in several use cases. Some possible examples are provided below.

SDN-based Intrusion Prevention System (IPS)

The first use case is depicted in Figure 6.1. An IPS (ServiceApp) coordinates a workflow involving the current status of an SDN-based Intrusion Detection System (IDS) and/or additional information collected by other SDNApps (e.g., network monitors) to feed an SDN-based firewall with the proper policies to block incoming attacks or suspicious traffic patterns. For instance, the IPS may need to observe the run-time state of the IDS (e.g., the internal variables that keep the list of policy violations) and of a network monitor to become aware of any anomalous state change and thus to detect anomalous traffic patterns (e.g., spikes in non-working hours). As soon as one among a set of particular changes occurs (i.e., a threat is detected by the IDS or anomalous traffic is observed through the network monitor), the IPS service triggers the creation of the required firewall rules that are then injected in the target SDNApp to protect the network infrastructure.

Migration Service

Another common use case is the migration of a running SDNApp, e.g., a NAT, to a new location, for instance, to follow user movements [157]. In this case, a Migration Service (i.e., ServiceApp) may need to acquire the run-time state of the existing SDNApp instance to properly bootstrap the new instance with the

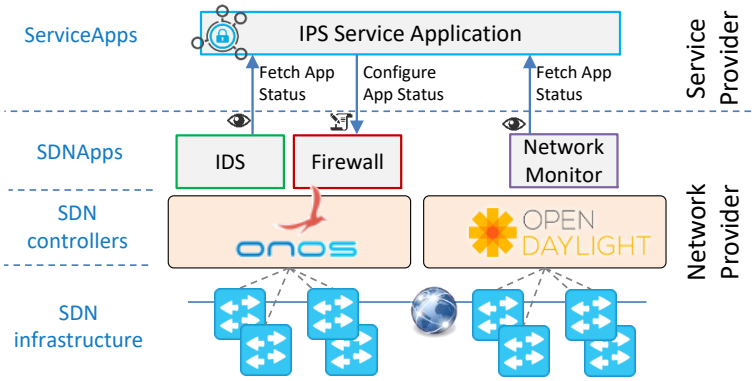


Figure 6.1: Use case example: an Intrusion Prevention System service (*ServiceApp*) requiring the access to the run-time state of multiple SDN applications (*SDNApps*).

current/correct state, which in case of a NAT, consists in the network address translation table. This technique avoids copying the entire memory of the SDNApp such as in the traditional Virtual Machine migration approach [15] and it exploits the idea of moving only semantically rich information as proposed in [65]. In this respect, an application-agnostic Migration Service that operates on whatever application internal data structure is possible only thanks to the capability of dynamically inspecting the run-time state of the SDNApp under consideration.

Service Orchestrators

This use case refers to the deployment and lifecycle management operations carried out by a Service-Defined Orchestrator (SDO — Chapter 4 on a set of infrastructure components that jointly realize an end service. During the service lifecycle, the Orchestrator may leverage data analytics tools to derive given performance indicators to optimize Network Service management operations or to prevent SLA violations. Indeed, data analytics tools run applications (i.e., SDNApps) that collect monitoring data from the underlying cloud (e.g., VNFs and virtual links) or network resources (e.g., OpenFlow switches and links) and then aggregate them to derive consolidated performance data indicators (e.g., user data flow throughput) [66]. These data may be used by the SDO to maintain an up-to-date view of resource usage and/or performance offered by the underlying infrastructure to promptly react in case of service degradations due to concurrent usage of resources from many different Network Services or due to any adverse event (e.g., service outages, network congestions, improper usage of VNFs). In this context, application-agnostic data models may foster composite and comprehensive service lifecycle workflows where both cloud and network resource status information are considered to offer high-quality services and highly effective resource utilization.

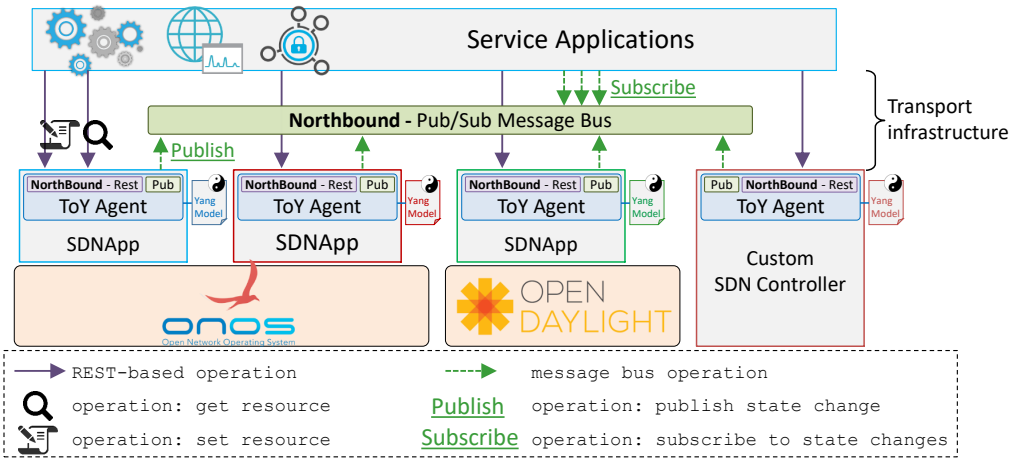


Figure 6.2: Overall software architecture. ServiceApps (on top) relies on a northbound interface (composed by the Pub/Sub message bus and on the Rest-based channel) to perform `get`, `set` and `subscribe` operations on the run-time state of SDNApps. These are accessible through this northbound thanks to the YANG-based mapping performed by ToY Agents.

6.4 Architecture

This section presents the software architecture that enables ServiceApps to read and modify the run-time state of SDNApps[1], while being agnostic to the implementation-specific data structures they use internally. The overall architecture is depicted in Figure 6.2. ServiceApps are envisioned to exploit SDNApps across SDN controllers while relying on the following main components: *(i)* a YANG-based *data model* associated with each SDNApp and describing its run-time state through high-level and application-independent structures (Section 6.4.1); *(ii)* a *communication infrastructure* enabling interactions between ServiceApp and SDNApps where exchanged information and the communication APIs are defined by the YANG data models associated with SDNApps (Section 6.4.2); and *(iii)* a *ToYang (ToY) Agent*, injected in each SDNApp that enables ServiceApps to access SDNApps run-time state according to the specified implementation-independent YANG data model (Section 6.4.3). The remainder of this section details these components.

6.4.1 A data model for SDNApps

To avoid ServiceApps to be aware of each SDNApp internal data structures and custom APIs, we define a high-level *model-based API*. This interface provides data

[1]In the remainder of this chapter, the expression "access the run-time state" will be used to indicate both these operations.

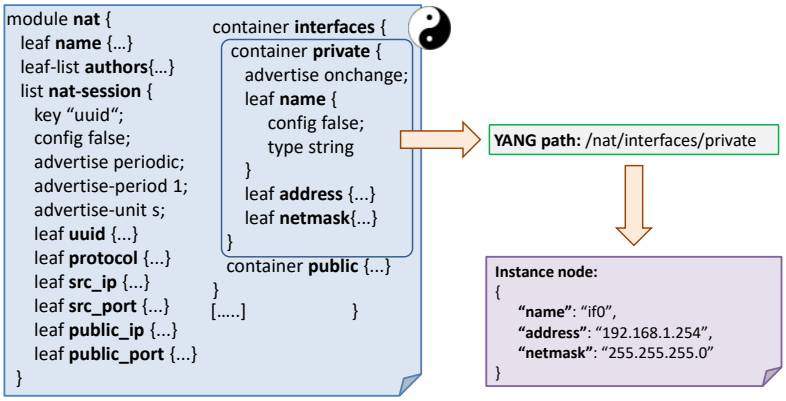


Figure 6.3: On the left, a YANG data model for the NAT application. On the right, an example of YANG path (the one of the *"private interface"* container highlighted in the model) and an instance node showing a possible run-time value.

through common syntax and modular structures defined by formal models written using the YANG modeling language [23]. Through the data model abstraction, an SDNApp maintainer can arbitrarily decide the portion of the internal state that has to be exposed to external services, e.g. omitting critical data such as authentication details. Each SDNApp is associated with its own data model, which is used to describe: *(i)* which data is exposed by the SDNApp, *(ii)* how to access such data with the desired granularity and *(iii)* how exposed data is structured. The way we represent this information within the data model is defined below. An example of a data model, associated with a NAT application, is shown on the left of Figure 6.3.

Each node of the data model (e.g., module, container, leaf) represents a *resource*, which is uniquely identified through a *YANG path*. The YANG path associated with each resource is generated from the data model using rules derived from RESTCONF [22] and shown in Table 6.1. Particularly, rows `#1` to `#7` of the table represent the basic rules, which can be combined as shown in row `#8` to specify any element of the data model. As shown in the table, the YANG path is an URI built as an ordered list of *YANG labels*, chaining the names of all the YANG elements that enclose that resource, from the most external to the resource itself, separated by the character `"/"`. For instance, in the data model provided in Figure 6.3, the container `private` is enclosed in the container `interfaces`, which is in turn enclosed in the module `nat`; therefore, the resource `"private"` is identified by the YANG path `/nat/interface/private`. YANG paths may enclose predefined portions (e.g., `/nat/nat-session`) and parametric portions (e.g., `/{uuid}`). In particular, resources within a collection are identified by YANG paths that include a *parametric label* (Table 6.1, rows `#5` and `#7`); this label corresponds to the value of the `key` leaf in the list node description. As an example, in Figure 6.3 each element of the `nat-session` list is identified by a YANG path in the form `/nat/nat-session/{uuid}`; thus, to identify a specific element of the

Table 6.1: Deriving the YANG path from the YANG data-model.

#	YANG element	YANG path	Example from Figure 6.3
1	Entire data-model	/module-name	/nat
2	Container	[...]/container-name	[...]/interfaces
3	Leaf(/Anydata)	[...]/leaf-name	[...]/name
4	Entire leaf-list	[...]/leaf-list name	[...]/authors
5	Element in a leaf-list	[...]/leaf-list/{value}	[...]/authors/Castellano
6	Entire list	[...]/list-name	[...]/nat-session
7	Element in a list	[...]/list-name/{key-field}	[...]/nat-session/0xa26
8	Generic element	/element1/.../elementN	/nat/interfaces
			/nat/interfaces/private
			/nat/nat-session
			/nat/nat-session/0x26
			/nat/nat-session/0x26/dst_ip

`nat-session`, the parametric label `{uuid}` must be replaced by the instance value of the `uuid` field of the desired element, e.g., `0x26` as in Table 6.1.

Each element is associated with a `config` statement that indicates whether a resource is writable (`config true`) or read-only (`config false`). For instance, the resource `name` in Figure 6.3, which in this case represents the name of a network interface, cannot be modified by an external ServiceApp. Moreover, inspired by recent works in IETF [45], we associate a resource with a new `advertise` statement (defined as a YANG extension) indicating in which situations the SDNApp advertises the current resource value to the outside world: `onchange`, advertised any time the value changes; `onthreshold`, advertised just when the value exceeds a specific threshold; `periodic`, advertised with a certain frequency, regardless of whether it has changed or not (this is the case of the `nat-session` resource shown in Figure 6.3); `ondemand`, which means it is never advertised, hence the value must be explicitly requested by the ServiceApp.

High-level ServiceApps use YANG paths to subscribe/get access to SDNApps resources. Figure 6.3 shows an example of a JSON structure derived from the data model that a ServiceApp may use to modify the configuration of the NAT private interface. In the remainder of this chapter, a JSON formatted according to (a piece of) YANG data model is called *instance node*. It may contain either the new configuration to be assigned to a resource, or its current value.

6.4.2 Communication infrastructure

To enable SDNApps to receive configurations from ServiceApps and to expose their own run-time state in accordance with the data model, we define a logical *communication infrastructure* that offers two types of connections: a direct (REST-based) communication channel and a shared publisher/subscriber message bus.

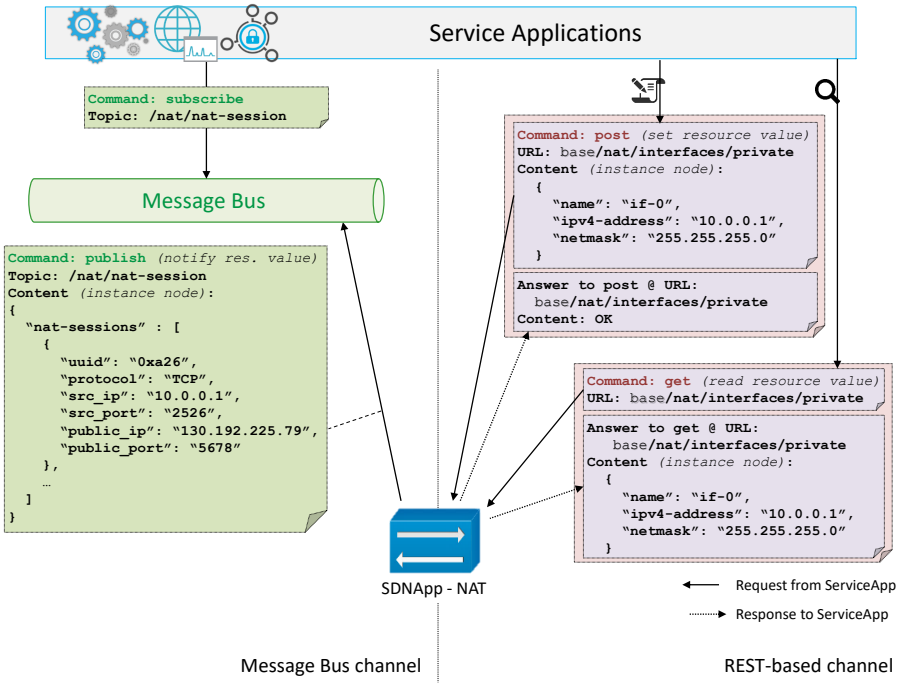


Figure 6.4: Example of messages sent on the message bus (in the left) and on the REST-based channel (in the right), defined according to the data model of Figure 6.3.

The first relies on a set of REST APIs, used by ServiceApps to send `get()` and `set()` commands to the ToY Agents of SDNApps, in order to explicitly retrieve or add/modify resources described in the associated data models.

This message bus channel is instead shared between entities that communicate through the publish/subscribe paradigm. In our proposal, ServiceApps act as subscribers, while the ToY Agents are the publishers. Particularly, they publish notifications on updates of resources marked as `onchange`, `onthreshold` or `periodically` in the data model (e.g., notifications may enclose the new value, the fact the resource has been deleted, and more). Subscriptions are handled hierarchically, thus when a ServiceApp subscribes on a given resource, it will receive updates on any nested resource.

As shown in Figure 6.4, in both channels the URI to be used by a ServiceApp to access a specific resource is dynamically derived from the YANG path associated with the resource of interest, becoming either a *URL* in the direct channel, or *topic* in the shared message bus. Data exchanged in both channels are instance nodes structured as defined within specific SDNApp YANG model.

6.4.3 To-Yang Agent

The *To-Yang (ToY) Agent* is the core component of our architecture. It is imported by each SDNApp as an external module, in order to *(i)* handle the mapping of (high level) resources described in the YANG data model on (low level) run-time application variables and *(ii)* manage the interaction between the SDNApp and the communication infrastructure. Using the terminology introduced in Section 6.4.1, the ToY Agent takes care of setting the configuration and fetching the run-time state of the SDNApp, providing to the ServiceApps an interface based on YANG paths and YANG-modeled instance nodes (as shown in Figure 6.4). We designed the ToY Agent to be agnostic with regard to the different SDNApps, to operate without introducing invasive modifications on them.

In the remainder of this section, we first describe structures and mechanisms used by the ToY Agent to map run-time application state into YANG-modeled data; then we present the architecture of the ToY Agent, hence detailing all the components that implement the above mechanisms. Finally, we describe the steps required to extend an existing SDNApp with our ToY Agent, also providing an overview of the application bootstrap workflow.

Path Mapping Table and Mapping Mechanism

While the YANG data model (together with all the derived YANG resources) provides a logical, implementation-independent view of the application, each SDNApp has its own internal data structure that may be only loosely related to the logical view. Figure 6.5 shows a possible internal structure of a NAT application that is modeled through the more generic YANG in Figure 6.3. By comparing the two structures, we can easily notice, for instance, that some common fields have a different name (e.g., the `nat-session` list becomes `sessionList`) or a different "location" (e.g., the leaf `public_ip` is no longer a value inside the session list, but is stored in the `publicAddress` variable). In general, there may be arbitrary differences.

Then, since the ToY Agent communicates with ServiceApps through the high-level YANG abstraction (i.e., YANG paths and instance nodes), it needs a mechanism to identify which variables in the SDNApp code should be accessed to accomplish a request or publish updates. For example, if a ServiceApp performs a `get()` request on the YANG path `/nat/nat-session/0x26/public_ip`, the ToY Agent needs to know that this value is stored in a certain variable, e.g., in the `publicAddress` attribute of `natData` object (Figure 6.5). Instead, if the request refers to the YANG path `/nat/nat-session/0x26`, it needs to know where all needed nested values (`protocol`, `src_ip`, `src_port`, etc. in Figure 6.3) are stored within the application code in order to reconstruct the instance node.

This information is different from case to case, hence it cannot be embedded in the ToY Agent since we want this module to be agnostic with respect to both

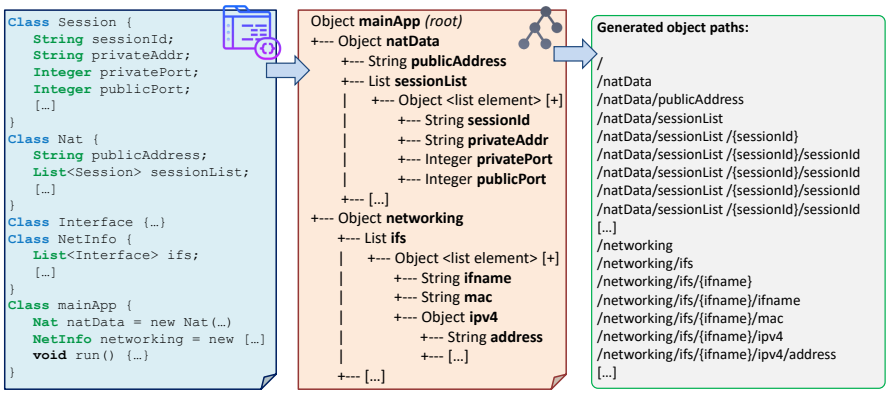


Figure 6.5: From run-time variables (in the left) to object paths (in the right).

the SDNApp implementation and the data model. For this reason, we defined a data structure called *Path Mapping Table (PMT)*, that describes each association between a YANG resource and the corresponding variable in the application source code. The ToY Agent, once imported in a particular SDNApp, exploits this information to learn, at run-time, how variables of that SDNAppare structured and how they are associated with the high-level YANG resources.

Particularly, as shown in Table 6.2, the PMT maps the YANG paths derived from the data model to the proper *object path* within the application *object tree*. We define *object tree* the structure of all the variables (i.e., all objects and their attributes) in the application source code. For instance, the box in the center of Figure 6.5 reports the object tree corresponding to the source code of the box on the left. It is worth noting that, unlike the data model, which can be associated with all the SDNApps implementing the same functionality (e.g., NAT), the object tree is specific for a given implementation, since it depends on the source code of the application itself. The first element of the object tree is called *root object*, and corresponds to the most external object/variable that we have access to (e.g., the object `mainApp` in Figure 6.5). Each element of the object tree (i.e., an application variable) is identified by an *object path* that begins with the root (character `"/"`), followed by an ordered list of attributes built as already described in Section 6.4.1 in case of YANG path. The right side of Figure 6.5 shows the object paths corresponding to the object tree depicted in the central box of the figure.

Since an SDNApp may feature a source code structure that is very different from the high-level view presented in the YANG model, this originated the necessity of supporting a complex syntax for the PMT records. In the following we present some of the mapping cases we encountered in our work, using them to intuitively describe how the PMT and its records are formatted.

The most common mapping case is for application variables that are associated with YANG resources having simply a different name (e.g., `sessionList` is mapped to the `nat-session` resource). Another case is when application variables

Table 6.2: Excerpt of PMT, referred to the data model of Figure 6.3 and to the object tree shown in Figure 6.5.

#	YANG path	Object path
1	/nat	/
2	/nat/nat-sessions	/natData/sessionList
3	/nat/nat-sessions/{uuid}	/natData/sessionList/{sessionId}
4	/nat/nat-sessions/{uuid}/src_ip	/natData/sessionList/{sessionId}/srcAddr
5	/nat/nat-sessions/{uuid}/public_ip	/natData/publicAddress
6	/nat/interfaces	/networking/ifs
7	/nat/interfaces/{noresource}	/networking/ifs/{ifname}
8	/nat/interfaces/private	/networking/ifs/if0
9	/nat/interfaces/private/address	/networking/ifs/if0/ipv4/address

do not have any mapping to a resource in the data model (e.g., variable `mac` in the object tree show in Figure 6.5), in which case no entries are featured in the PMT (Table 6.2). In some cases, resources that are *inside* the same parent element in the data model may correspond to variables located in different objects in the object tree, or vice versa. For example, the resources `nat-session` and `interfaces` (rows `#2` and `#6` in Table 6.2) are children of the `nat` node in the data model of Figure 6.3, while their corresponding objects, namely `sessionList` and `ifs`, are attributes of two different objects in the object tree of Figure 6.5. All the previous cases lead to the necessity of a record for any possible YANG path within the PMT.

A frequent mapping case is when items of a YANG list have a direct correspondence with items of a collection object (e.g., an array, a list, a set or a map). An instance of this mapping is realized with rows `#3`, `#4` and `#5` of Table 6.2. To distinguish resources in `/nat/nat-session`, the parametric label `{uuid}` is used in the YANG path, since the `uuid` leaf is the `key` for the items of the `nat-session` list according with the YANG model of Figure 6.3. To map each resource in this list with a specific object of `sessionList` in the object tree, the attribute `sessionId` is used as a parametric label of the object path, since its value corresponds to the YANG `key` field (although the application source code does not indicate in any way that such a variable is a key in the list).

Finally, we encountered some occasional cases where a *non-list* resource in the data model is mapped to a specific item of a collection (e.g., a list) in the object tree, or vice versa (e.g., the `private` resource in the data model of Figure 6.3 corresponds to a specific element of the list `ifs` in the object tree of Figure 6.5). This mapping is descried through rows `#7` and `#8` of Table 6.2. Particularly, in row `#8` the parametric label in the object path is replaced with the specific value `"if0"`, i.e., the value of the object that maps to the resource identified by the YANG path `/nat/interfaces/private`. Moreover, row `#7` maps a dummy resource (called `noresource` in Table 6.2) within YANG data model, into a generic element (identified by the parametric label) of the collection in the object tree. This way, the

ToY Agent knows that the parametric label of `ifs` is `ifname` and, e.g., `if0` is the instance value of the corresponding variable `ifname` in the application code.

To make the ToY Agent agnostic with respect to the data model and independent from the particular SDNApp internal variables structure, the PMT must be coupled with a set of *Mapping Algorithms* that exploit this data structure and enable both read and write access to the configuration and run-time state starting from its high-level representation. Particularly, given a PMT, a YANG data model and the associated SDNApp, the mapping algorithms both retrieve and set the SDNApp variables, converting YANG-modeled structures into application objects (and vice versa). As we do not want to introduce overhead on the SDNApps code by implementing an ad-hoc mapping for each of them, those algorithms are application-agnostic and operate just relying on the PMT and on the data model. Our mapping algorithms, described in detail in Section 6.5, provide the following elementary operations: *(i)* resolve the association between a YANG path and an object path and vice versa, through a lookup on the PMT; *(ii)* fetch an object starting from its object path; *(iii)* get the value of a given object and write it into the corresponding instance node (e.g., JSON), according to the YANG data model and the PMT (the YANG data model is used to derive the structure of the instance node, while the PMT shows how to map application variables into YANG resources); *(iv)* set the value of application objects starting from the corresponding instance node (e.g., JSON), according to the PMT.

ToY Agent Architecture

Figure 6.6 details the architecture of the ToY Agent. This component is linked to the original SDNApp and must be configured with both the data model and the PMT associated with the SDNApp itself. The ToY Agent architecture has been designed to work on top of any SDNApp implementation; it is described below through a top-down approach.

The northbound interface of the ToY Agent consists of two modules that connect to the communication infrastructure, namely the **YANG Based Publisher** and the **YANG based REST APIs**, which implement respectively the shared message bus publisher and the HTTP REST server described in Section 6.4.2.

The **On-Demand Access Handler** manages `get()` and `set()` commands coming from the ServiceApps through the REST APIs, relying on the `get_node` and `set_node` core primitives provided by the Mapping Library. Instead, the **Object Listener** implements the monitoring procedure needed to recognize and notify updates on the SDNApp state. It performs the following operations: *(i)* identify, through the data model, the YANG path of resources that should be exported (and when to export them); *(ii)* rely on the Mapping Library to convert these YANG paths into objects (`map_path` and $\texttt{fetch}_o$ primitives); *(iii)* build the corresponding instance node (using `get_node` primitive) and export it through the YANG Based

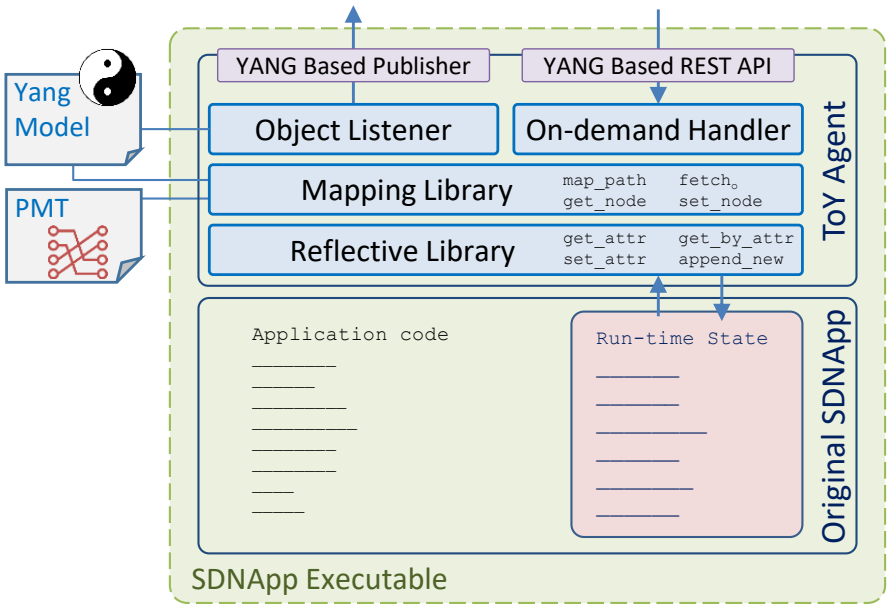


Figure 6.6: Architecture of the ToY Agent.

Publisher on the message bus whenever the condition specified in the YANG data model for that resource (`onchange`, `onthreshold` or `periodically`) occurs.

The **Mapping Library** is the core module of the ToY Agent, as it represents the bridge between the high-level YANG interface and the low-level object-based structure of application variables. It both accesses the PMT and the YANG data model, and relies on a Reflective Library to implement the mapping procedures introduced in 6.4.3, thus providing the following primitives: `map_path`, converts a YANG path to an object path through the PMT; `fetch`$_o$, returns an object starting from its object path; `get_node`, returns the instance node (e.g., a JSON structure) of the requested YANG resource, reading values from the application variables; `set_node`, modifies the application variables based on the input YANG resource instance. In Section 6.5 we formalize the algorithms that implement these procedures.

The **Reflective Library** is the component that directly interacts with the original application actually accessing its variables. Particularly, it allows to examine and modify application variables at run-time through the use of *reflection* [106] (a.k.a. *introspection*) [2]. Through the run-time introspection of application objects, the Reflective Library provides an interface built with the following primitives (also

[2]Reflection is required since the ToY Agent knows attribute names just during its execution, e.g., when a particular request is performed, deriving them from the object path retrieved in the PMT. As we validated our proposal through Java-based SDNApps, we exploited the *JAVA Reflection API* to access and manipulate run-time objects.

```
// Import the ‘‘ToY Agent’’ module into this application
import sdnapp.ToYAgent;

// This code should be added in the ‘‘bootstrap method’’ of the SDNApp
// Instantiate the ToY Agent, providing it a reference to the root object, the PMT and the model
public ToYAgent tya = new ToYAgent(this, pmt, yangModel);
// Start the ToYAgent and let's forget about it
tya.start();
```

Figure 6.7: Steps required to include the ToY Agent in an existing application.

shown in Figure 6.6): `get_attribute()`, `get_by_attribute()`, `set_attribute()` and `append_new()`. These are the only procedures that have direct access to the original SDNApp execution environment (Figure 6.6). In Section 6.5 all reflective functions are formally defined.

It is worth noting that, since the ToY Agent directly accesses to application variables, race conditions may derive from possible situations in which the ToY Agent sets a variable value on which the SDNApp itself is working on it (or even read a temporarily inconsistent piece of state). To avoid this problem, we assume that SDNApps are thread safe, i.e., that critical regions are properly protected so that the ToY Agent is prevented from modifying variables during critical operations or read inconsistent data.

Embedding Mechanism and Operating Workflow

Mapping mechanism used by the ToY Agent, as well as its architecture, have been designed to keep it independent from the particular SDNAppṪhis way, we reduce the overhead of the SDNApp developer, which must not write an ad hoc ToY Agent for each new application, thus allowing the inclusion of any SDNApp in our configuration framework almost without introducing any changes in the application code.

Figure 6.7 shows the steps that an SDNApp developer is required to carry out to make its software compatible with our framework: *(i)* include the ToY Agent module in the SDNApp; *(ii)* initialize the ToY Agent at the application bootstrapping, by providing it a reference to the root object (the most external object that can be accessed from the outside), the Path Mapping Table and the YANG data model; *(iii)* start the ToY Agent.

At the bootstrapping, the ToY Agent performs the following preliminary operations: *(i)* notifies the SDNApp identifier and the YANG data model through the message bus, so that external ServiceApps are enabled to exploit the new application; *(ii)* analyzes the data model to identify when the value of each resource should be exported (i.e., `onchange`, `onthreshold`, `periodically` or `ondemand`); *(iii)* reads the PMT to understand how to map YANG resources into application objects; *(iv)* starts an *Object Listener* (Figure 6.6) that monitors and exports objects associated with resources to be notified `onchange`, `onthreshold`, whenever

the conditions defined in the data model are satisfied, and `periodic`, whenever the appropriate timer expires; *(v)* enables the REST-based communication channel to allow ServiceApps to explicitly access the SDNApp run-time state.

6.5 Mapping Algorithms

In this section, we formalize the algorithms used to map the object-oriented internal state of an SDNApp into the high-level YANG-modeled state, and vice versa. We identified four elementary operations, implemented by as many Mapping Algorithms: `map_path()`, $\texttt{fetch}_o\texttt{()}$, `get_node()` and `set_node()`.

The operations implemented by the algorithms presented in this section constitute the Mapping Library, the core module of the ToY Agent presented in Section 6.4.3.

6.5.1 Structures and Function Notations

Table 6.3 presents formal notation for data structures that will be considered during the description of the mapping algorithms. For the sake of clearness, besides its formal description, every notation is accompanied with an example that illustrates it within the NAT use case shown in Figure 6.3. Additionally, mapping algorithms rely on the following functions.

Label Functions. Given a resource $\boldsymbol{v} \in Y$, the function $\texttt{label}_y$ returns the label $\lambda_y \in L_y$ of that resource, i.e., the name of the corresponding YANG node in the model. Given an object $\boldsymbol{o} \in O$, the function $\texttt{label}_o$ returns the label $\lambda_o \in L_o$, i.e., the name of that variable/attribute in the run-time application environment.

Parametric Label Functions. Since, as described in Section 6.4.1, resources in a list are identified by a YANG path that ends with a parametric label (namely, the name of the `key` field in the model), we use function $\texttt{param\_label}_y\colon L_y^n \rightarrow L_y$ to know the parametric label of a list resource. Analogously, we use function $\texttt{param\_label}_o\colon L_o^n \rightarrow L_o$ for collection objects. For instance, in Figure 6.3, $\texttt{param\_label}_y\texttt{("/nat/nat-sessions")} = \texttt{"hash"}$. Note that this function can be easily implemented through an inspection of the PMT.

Instance Node Helpers. Given a key-value instance node $\boldsymbol{\nu} \in \mathcal{N}_{kv}$ and a label $\lambda_y \in L_y$, we use *(i)* the function $\texttt{get\_element}(\boldsymbol{\nu}, \lambda_y)$ to get the node $\boldsymbol{\nu}_i \in \boldsymbol{\nu}$, and *(ii)* the function $\texttt{put\_element}(\boldsymbol{\nu}, \lambda_y, \boldsymbol{x})$ to set $\boldsymbol{\nu}_i$ to an input value $\boldsymbol{x} \in \mathcal{N}$, where $\boldsymbol{\nu}_i$ is the node nested in $\boldsymbol{\nu}$ with key $\boldsymbol{\lambda_y}$ (in symbols $\texttt{key}(\boldsymbol{\nu}_i) = \boldsymbol{\lambda_y}$). Given a list instance node $\boldsymbol{\nu} \in \mathcal{N}_l$, we use *(iii)* function $\texttt{append\_element}(\boldsymbol{\nu}, \boldsymbol{\nu}_i)$ to add the instance node $\boldsymbol{\nu}_i \in \mathcal{N}$ to the list $\boldsymbol{\nu}$. For example, let $\boldsymbol{\nu}$ be the instance node in Figure 6.3; then, $\texttt{get\_element}(\boldsymbol{\nu}, \texttt{"name"}) = \texttt{"if0"}$.

Reflective Library Functions. As described in Section 6.4.3, Mapping Library relies on the Reflective Library. It provides some functions that, given an object

Table 6.3: Formal notation for data structures used in the algorithms description.

Notation	**Description**	**Example**
$Y = Y_{co} \cup Y_{li} \cup Y_{le} \cup Y_{ll}$	Set of all YANG nodes in the data model, i.e., modules and containers (Y_{co}), lists (Y_{li}), leafs (Y_{le}) and leaf-lists (Y_{ll}).	`nat` and `interfaces` (Figure 6.3) belong to Y_{co}, while `nat-session` belongs to Y_{li}.
$\boldsymbol{v} \in Y$	YANG resource, i.e., each node of a YANG model.	In the YANG of Figure 6.3, each element in bold represents a resource.
$\boldsymbol{v_i} \in \boldsymbol{v}$	Resource $\boldsymbol{v_i}$ is a node directly nested in resource $\boldsymbol{v}$ (child resource).	Container `private` is a child resource of container `interfaces` (Figure 6.3).
$\boldsymbol{v}_{root} \in Y$	The root resource in Y.	The `nat` module in Figure 6.3.
L_y	Set of all YANG labels, i.e., the name of all YANG nodes in Y.	For the YANG in Figure 6.3, L_y={`"nat"`, `"name"`, `"authors"`, ...}.
$\boldsymbol{\rho}_y \in L_y^n$	YANG path of a given resource $\boldsymbol{v} \in Y$.	The YANG path of the resource highlighted in Figure 6.3 is `/nat/interfaces/private`.
$O = O_e \cup O_l \cup O_o$	Set of all application variables, i.e., "elementary objects" (O_e), e.g., integers, strings, booleans, "collection objects" (O_l), e.g., list, maps, "ordinary objects" (O_o) e.g., instances of custom classes.	In Figure 6.5, `networking` belongs to O_o, `ifs` belongs to O_l and `ifname` to O_e.
$\boldsymbol{o} \in O$	Application object, i.e., each variable/attribute in the SDNApp execution environment.	Every Java object in Figure 6.5.
$\boldsymbol{o_i} \in \boldsymbol{o}$	$\boldsymbol{o_i}$ is an attribute of $\boldsymbol{o}$ (child object).	In Figure 6.5 `publicAddress` is a child of the object `natData`.
$\boldsymbol{o}_{root} \in O$	Root object in O.	The root object of the object tree in Figure 6.5 is `mainApp`.
L_o	Set of all object labels, i.e., the name of all run-time variables of an SDNApp.	For the example in Figure 6.5, L_o = {`natData`, `publicAddress`, `sessionList`, ...}.
$\boldsymbol{\rho}_o \in L_o^m$	Object path of a given object $\boldsymbol{o} \in O$.	The object path of the list "ifs" in Figure 6.5 is `/networking/ifs`.
$\mathcal{N} = \mathcal{N}_e \cup \mathcal{N}_l \cup \mathcal{N}_{kv}$	Domain of all the possible instances (e.g., JSON) of resources described by a data model. They can be elementary nodes $\mathcal{N}_e$ (i.e., nodes without nested nodes), list nodes $\mathcal{N}_l$ and key-value nodes $\mathcal{N}_{kv}$.	The instance of a container resource is a key-value node, while the instance of leaf resource is an elementary node.
$\boldsymbol{\nu} \in \mathcal{N}$	Instance node of a resource $\boldsymbol{v} \in Y$, i.e., a piece of data that is structured as defined by the portion of YANG model associated with the resource $\boldsymbol{v}$.	An example of (key-value) instance node is the JSON data in Figure 6.3.
$\boldsymbol{\nu_i} \in \boldsymbol{\nu}$	Node $\boldsymbol{\nu_i}$ is directly nested in node $\boldsymbol{\nu}$ (child node).	In Figure 6.3, node `"name"` is directly nested in the example instance node.

$\boldsymbol{o} \in O$, allow reading and modifying the value of any $\boldsymbol{o_i} \in \boldsymbol{o}$ (i.e., any of its child objects). On ordinary objects, i.e., $\boldsymbol{o} \in O_o$, $\boldsymbol{o_i}$ is an attribute of $\boldsymbol{o}$; thus, the library provides *(i)* function `get_attribute`$(\boldsymbol{o}, \lambda_{o_i})$, that returns attribute named $\lambda_{o_i} \in L_o$ of object $\boldsymbol{o} \in O_o$, and *(ii)* function `set_attribute`$(\boldsymbol{o}, \lambda_{o_i}, \boldsymbol{x})$, that sets attribute named $\lambda_{o_i} \in L_o$ of object $\boldsymbol{o} \in O_o$ to value $\boldsymbol{x} \in O$. On collection objects, i.e., $\boldsymbol{o} \in O_l$, $\boldsymbol{o_i}$ is an item of the collection $\boldsymbol{o}$; thus, the library provides *(iii)* function `get_by_attribute`$(\boldsymbol{o}, \lambda_{o_i}, x)$, that returns the first item of collection object $\boldsymbol{o} \in$

O_l whose attribute named $\lambda_{o_i} \in L_o$ is equal to value $\boldsymbol{x} \in O$, and *(iv)* function `append_new`$(\boldsymbol{o}, \lambda_{o_i}, x)$, that appends to collection object $\boldsymbol{o} \in O_l$ a new item, whose attribute named $\lambda_{o_i} \in L_o$ is initialized to value $\boldsymbol{x} \in O$. For example, let $\boldsymbol{o}$ be the object `natData` of Figure 6.5; then a call to `get_attribute`$(\boldsymbol{o}$, `"public_Address"`$)$ returns the value stored in `natData.publicAddress`.

Fetch Resource Function. Given a YANG path $\boldsymbol{\rho}_y \in L_o^n$, to retrieve the corresponding resource $\boldsymbol{v} \in Y$ we use the function $\texttt{fetch}_y \colon L_o^n \to Y$. Note that $\texttt{fetch}_y$ can be easily implemented through a simple lookup in the YANG model.

6.5.2 Algorithms Description

We are now ready to describe the algorithms that perform the bidirectional map between run-time application variables and resources defined by the YANG data model. They constitute the **Mapping Library** module of the ToY Agent (Section 6.4.3). Since mapping algorithms exclusively rely on the content of the PMT and the YANG model, they enable the ToY Agent to be agnostic regarding the particular SDNApp implementation.

Map Path Algorithm. This algorithm establishes a link between YANG resources and application objects, since it converts a YANG path $\boldsymbol{\rho}_y \in L_y^n$ to the corresponding object path $\boldsymbol{\rho}_o \in L_o^m$, according to the information stored into the PMT. This path mapping is implemented by the function `map_path`$\colon L_o^m \to L_y^n$, whose algorithm is omitted since it is a trivial key-value lookup in the PMT, that retrieves the proper object path based on the input YANG path.

Fetch Object Algorithm. Algorithm 7 shows the $\texttt{fetch}_o$ procedure, which, given an object path $\boldsymbol{\rho}_o$, retrieves the reference to the corresponding object. To do this, it performs iterative access to run-time variables in the object tree, fetching, at each iteration, the object associated with the next label of the path $\boldsymbol{\rho}_o$ given as input (Algorithm 7, line 2). For each label, the next object is fetched through the function `get_attribute`, in case the previous object is not a collection (line 7), while `get_by_attribute` is used otherwise, passing the parametric label as attribute (lines 4-5). Both these functions are provided by the Reflective Library (as described in Section 6.5.1).[3]

Example: Referring to the object tree in Figure 6.5, lets consider object path $\boldsymbol{\rho}_o$ = `/networking/ifs/if0` as an input to the $\texttt{fetch}_o$ procedure. The algorithm iterates over the three labels of $\boldsymbol{\rho}_o$ (i.e., `"networking"`, `"ifs"` and `"if0"`) to access, each time, the next nested object until the reference to the last one (`"if0"`) is finally fetched. At the first iteration, the attribute `networking` of the root object is accessed (Algorithm 7, line 7), then, at the second iteration attribute `ifs` of

[3]In Algorithm 7, the helper function `sub_path` truncates a path to the position given as input (position is considered from the root for positive inputs, from the last label for negative ones). For instance, `sub_path("/natData/sessionList/0x26", −1) = "/natData/sessionList"`.

Algorithm 7 $\texttt{fetch}_o$ for object path $\boldsymbol{\rho}_o$

Input: $\boldsymbol{\rho}_o \in L_o^n$
Output: $\boldsymbol{o} \in O$

1: $\boldsymbol{o} = \boldsymbol{o}_{root}$
2: **for all** $\lambda_i \in \boldsymbol{\rho}_o$ **do**
3: **if** $\boldsymbol{o} \in O_l$ **then**
4: $\lambda_i^p = \texttt{param\_label}_o(\texttt{sub\_path}(\boldsymbol{\rho}_o, i))$
5: $\boldsymbol{o} = \texttt{get\_by\_attribute}(\boldsymbol{o}, \lambda_i^p, \lambda_i)$
6: **else**
7: $\boldsymbol{o} = \texttt{get\_attribute}(\boldsymbol{o}, \lambda_i)$
8: **return** $\boldsymbol{o}$

Algorithm 8 `get_node` from a YANG path $\boldsymbol{\rho}_y$

Input: $\boldsymbol{\rho}_y \in L_y^n$
Output: $\boldsymbol{\nu} \in \mathcal{N}$

1: $\boldsymbol{o} = \texttt{fetch}_o(\texttt{map\_path}(\boldsymbol{\rho}_y))$
2: $\boldsymbol{v} = \texttt{fetch}_y(\boldsymbol{\rho}_y)$
3: **if** $\boldsymbol{v} \in Y_{co}$ **then**
4: $\boldsymbol{\nu} = \texttt{dict\_node}()$
5: **for all** $\boldsymbol{v}_i \in \boldsymbol{v}$ **do**
6: $\boldsymbol{\nu}_i = \underline{\texttt{get\_node}}(\texttt{add\_path}(\boldsymbol{\rho}_y, \texttt{label}_y(\boldsymbol{v}_i)))$
7: $\texttt{put\_element}(\boldsymbol{\nu}, \texttt{label}_y(\boldsymbol{v}_i), \boldsymbol{\nu}_i)$
8: **else if** $\boldsymbol{v} \in Y_{li} \cup Y_{ll}$ **then**
9: $\boldsymbol{\nu} = \texttt{list\_node}()$
10: **for all** $\boldsymbol{o}_i \in \boldsymbol{o}$ **do**
11: $\lambda_i^p = \texttt{param\_label}_o(\texttt{map\_path}(\boldsymbol{\rho}_y))$
12: $\boldsymbol{\nu}_i = \underline{\texttt{get\_node}}(\texttt{add\_path}(\texttt{path}_y(\boldsymbol{\rho}_y, \lambda_i^p)))$
13: $\texttt{append\_element}(\boldsymbol{\nu}, \boldsymbol{\nu}_i)$
14: **else**
15: $\boldsymbol{\nu} = \boldsymbol{o}$
16: **return** $\boldsymbol{\nu}$

the object `networking` is accessed. Since `ifs` is a list, during the third and last iteration the procedure performs a lookup on the PMT to discover its parametric label λ_i^p, i.e., `ifname` (Table 6.2, row 7); the parametric label is then used to fetch the correct object inside the list `ifs`, namely, the one whose attribute `ifname` is equal to `"if0"` (Algorithm 7, lines 4-5). Since `"if0"` is the last label of the object path, finally the $\texttt{fetch}_o$ procedure returns a reference to this object (Algorithm 7, line 10).

The two main operations needed to map the SDNApp run-time variables (that represent configuration and run-time state) into a high-level YANG-modeled structure (and vice versa) are described by the recursive algorithms `get_node` and `set_node`.

Get Node Algorithm. Each time the ToY Agent receives, from a ServiceApp, a `get()` command related to a given YANG path $\boldsymbol{\rho}_y$, it uses procedure `get_node`

to create and return the corresponding instance node (i.e., JSON data). It is *(i)* structured according to the piece of data model associated with the resource $\boldsymbol{v} = \texttt{fetch}_y(\boldsymbol{\rho}_y)$ and *(ii)* filled with current values of application variables. To fill the instance node with proper values, a series of recursive calls to `get_node` are performed (for readability, recursive calls are marked with underscores in Algorithm 8). At each recursion step, the algorithm relies on `map_path` and $\texttt{fetch}_o$ procedures (line `1`) to fetch the object $\boldsymbol{o}$ that, according with the PMT, is mapped on the resource identified by the input YANG path $\boldsymbol{\rho}_y$. The algorithm starts building the node instance of the resource identified by the input YANG path $\boldsymbol{\rho}_y$. Then, at each recursion step, this path is extended in order to build and fill all inner nodes (Algorithm 8, lines `6` and `13`); these inner nodes are then added to the parent node (lines `7` and `14`)[4]. Each recursion branch stops when a leaf resource is reached; the corresponding (elementary) instance node is filled with the value of the object fetched in that recursion (lines `16-17`).

Example: We now provide an example of the `get_node` procedure, using as reference the YANG in Figure 6.3 and supposing that a ServiceApp needs to retrieve the resource identified by the YANG path $\boldsymbol{\rho}_y$ = `/nat/interfaces/private`. In Algorithm 8, line `1`, the `get_node` procedure retrieves the object path $\boldsymbol{\rho}_o$ = `/networking/ifs/if0` through a lookup into the PMT (Table 6.2, row `#8`). Then, in the same line, it gives this object path as input to the $\texttt{fetch}_o$ procedure, in order to get the reference to the corresponding object $\boldsymbol{o}$ (as described in the previous example). After the object $\boldsymbol{o}$ has been fetched, the same is done for its corresponding resource $\boldsymbol{v}$ (line `2`) through the YANG model, which is needed to know how the instance node must be structured. Since in our example the resource `/nat/interfaces/private` is a container, a key-value instance node is initialized (Algorithm 8, line `4`). Then, for each child resource, the algorithm is repeated recursively and the instance node populated with all the child-nodes obtained this way (lines `6-7`). For instance, let's consider the child resource with label `"address"`; the path $\boldsymbol{\rho}_y$ is extended with this label (i.e., `/nat/interfaces/private/address`) and `get_node` is called again. At this point, the object path obtained through the lockup into the PMT is `/networking/ifs/if0/ipv4/address`. Thus, $\texttt{fetch}_o$ accesses to the inner objects, i.e., first `ipv4`, then `address`, thus returning the last one, whose value is the IP address of the private interface. Since the resource $\boldsymbol{v}$ (i.e., `address`) of the current recursion is a leaf, this value is directly used to fill the node $\boldsymbol{\nu}$ (line `17`). The just filled instance node is then returned to the caller (line `19`) so that it can be appended to the parent node (lines `7` or `14`) The same is done for other children resources of `/nat/interfaces/private`, namely `name` and `netmask`. At the end of all the recursion steps, the just built JSON node will be

[4]In Algorithm 8, the helper function `add_path` extends the YANG path with the label given as input.

Algorithm 9 `set_node` $\boldsymbol{\nu}$ in object associated with $\boldsymbol{\rho}_y$

Input: $\boldsymbol{\rho}_y \in L_y^n$, $\boldsymbol{\nu} \in \mathcal{N}$

```
 1: ρ_o = map_path(ρ_y)
 2: o = fetch_o(ρ_o)
 3: if ν ∈ N_kv then
 4:     for all ν_i ∈ ν do
 5:         set_node(add_path(ρ_y, key(ν_i), ν_i)
 6: else if ν ∈ N_l then
 7:     for all ν_i ∈ ν do
 8:         id_i = get_element(ν_i, param_label_y(ρ_y))
 9:         append_new(o, param_label_o(ρ_o), id_i)
10:         set_node(ν_i, add_path(ρ_y, id_i))
11: else if ν ∈ N_e then
12:     o_parent = fetch_o(sub_path(ρ_o, −1))
13:     set_attribute(o_parent, label_o(o), value(ν))
```

equal to the one shown in Figure 6.3; at last, it can be returned by the ToY Agent to the ServiceApp as response to the `get()` command.

Set Node Algorithm. The `set_node` procedure (Algorithm 9) is used by the ToY Agent to modify the configuration of an SDNApp each time the ToY Agent receives a `set()` command from a ServiceApp. It takes a YANG path $\boldsymbol{\rho}_y$ and an instance node $\boldsymbol{\nu}$ (e.g., JSON) as input, and configures the SDNApp modifying its run-time variable according to the data in the instance node. To configure all run-time variables with value stored in the instance node, a series of recursive calls to `set_node` are performed (for readability, recursive calls are marked with underscores in Algorithm 9). Similarly to `get_node`, the algorithm starts by fetching the object corresponding to the input YANG path $\boldsymbol{\rho}_y$; this is configured through functions `append_new` or `set_attribute` (lines `10`, `14`), provided by the Reflective Library. At each recursion step, the base YANG path is extended with the label of each child resource, in order to fetch and configure all inner objects. When a list should be extended with a new object (lines `8-11`), first its unique ID (attribute corresponding to the parametric label) is initialized according to the value of the instance node (the attribute initialization is implemented by the function `append_new` of the Reflective Library); then, it is in turn configured through a recursive call to `set_node` (line `11`). Even in this case, each recursion branch stops when a leaf node in the YANG data model is reached; its instance value is used to configure the corresponding object (lines `14-15`)[5].

Example: Let us suppose that instance node $\boldsymbol{\nu}$ of Figure 6.3 is used to configure an application having the object tree in Figure 6.5. The YANG path $\boldsymbol{\rho}_y =$

[5]Note that, to modify the value of a leaf object, Algorithm 9 needs to fetch the reference to the parent object.

`/nat/interfaces/private` is given as input to `set_node`, together with the instance node $\boldsymbol{\nu}$. Algorithm 9 lines `1-2` fetch object $\boldsymbol{o}$ and resource $\boldsymbol{v}$. At this point, since the instance node is a key-value structure (the condition in line `3` is verified), the YANG path is extended once for each inner node (lines `4-5`), in order to continue the recursion. For instance, for the inner node with key `"address"`, the input YANG path `/nat/interfaces/private` is extended as `/nat/interfaces/private/address`; this is used, together with its value `"192.168.1.154"`, to call again the `set_node` procedure (line `5`). With this new input, due to the mapping into the PMT (Table 6.2, row `#9`), object $\boldsymbol{o}$ returned by $\texttt{fetch}_o$ in line `2` is the string attribute `privateAddress` (Figure 6.5). Since $\boldsymbol{\nu}$ is now an elementary node, object $\boldsymbol{o}$ is set to its value `"192.168.1.154"` (Algorithm 9, line `15`). In the same way, `set_node` is called recursively also for other child resources of `/nat/interfaces/private`, namely `name` and `netmask`.

The performance of the algorithms presented above is analyzed and discussed in the next section.

6.6 Experimental Results

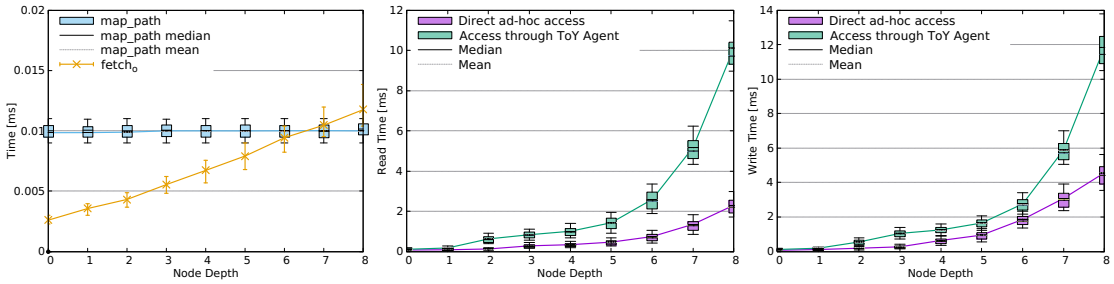
We set up a testbed consisting of a Mininet virtual network, controlled by an instance of the ONOS SDN controller[6]. Our evaluation focuses on two major sets of results. We first measure the overhead introduced by the ToY Agent for individual *read* and *write* operations, together with update notifications; to compare performance, we also run tests on ad-hoc manipulated SDNApps. Then we analyze the applicability of our approach to the use cases presented in Section 6.3.

6.6.1 ToY Agent Evaluation

To evaluate the delay introduced by the ToY Agent, we used an existing application called *"SSSA-analytic-tool"*, which serves network orchestration functions within SDN domains [112]. Since this application has been developed independently from this work, it does not natively provide any interface to allow an external service (e.g., network service orchestration) to access its state.

To integrate this SDNApp into our framework, we simply included the generic ToY Agent module (the source code of our prototype is available at [4]) as described in Section 6.4.3. Additionally, to evaluate the ToY Agent overhead against traditional ad-hoc approaches, we conducted our tests also on a second version of the *SSSA-analytic-tool*, which has been modified by means of a REST interface that

[6]In our tests, we deployed ONOS, Mininet, and all modules of our architecture on a Linux debian 4.14.13-1-amd64, on top of a physical machine with 16 GB of RAM running an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz.



(a) `map_path` and `fetch`$_o$. (b) `get_node`. (c) `set_node`.

Figure 6.8: (a) Partial times needed to *(i)* map YANG paths into Object paths and *(ii)* fetch an object reference starting from its path, for different depth levels. (bc) `get_node` (b) and `set_node` (c) total times for different depth levels; values are compared with direct ad-hoc access. Upper and lower quartils are plotted along with errorbars.

enables direct access to run-time data. In this last case, the JSON representation is strictly dependent on the application itself (it follows the same structure of the SDNApp variables) and there is no higher level common abstraction.

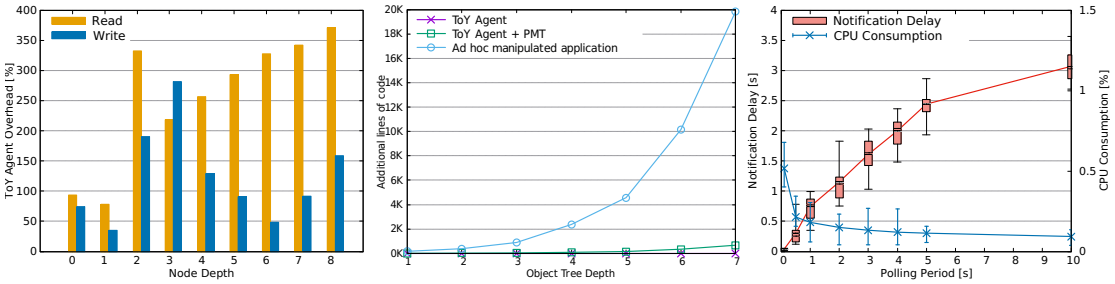
Measurements have been taken considering the variation of the `depth` parameter, i.e., the maximum length among the YANG node branches starting from the node that has to be set/retrieved by the operation to perform. For instance, all leafs have depth 0, the instance node of the *private interface* in Figure 6.3 has depth 1 (since it contains only elements with depth 0), and so on. The maximum depth of an instance node of the YANG in Figure 6.3 is 3 (we then say that the depth of the YANG model itself is 4). For validation purpose, a YANG model with depth 9 has been used (≈1000 resources), together with a PMT to associate YANG resources with the SDNApp variables.Higher levels of depth have not been considered since rarely widespread data-models exceed a depth of 6.

Access Time

We performed some `get()`/`set()` operations varying the depth parameter, thus testing the performance of each algorithm described in Section 6.5.2 and their scalability. Results taken over one thousand samples for each depth level are shown in Figures 6.8(a-c).

In particular, Figure 6.8a shows the partial times required by the `map_path` and the `fetch`$_o$ procedures. As expected, the time taken to convert a YANG path into the corresponding object is almost constant (about ten microseconds) with the depth growing, since this operation corresponds to a lookup on the PMT. The `fetch`$_o$ procedure time has the same order of magnitude, but its value slightly increases with the increase of the depth (with slow linear growth).

Figure 6.8b depicts the total time needed by the ToY Agent to perform read



(a) (b) (c)

Figure 6.9: (a) Time overhead introduced by the ToY Agent for read and write operations, at different depth levels. (b) Additional lines of code needed to enable read and write access to SDNApp variables for different depth levels. (c) ToY Agent CPU consumption and notification delay for different polling periods (upper and lower quartiles are plotted along with error bars).

(`get_node`) operations. As the graph shows, even if the curve features an exponential trend (as expected from the use of recursive algorithms), for depth levels below 6 its growth is still almost linear. This is a significant result since widespread data-models usually feature far smaller depths. Values are compared with those of the ad-hoc manipulated application. For depth levels below 7, despite the ToY Agent overhead, the total time still sticks below a couple of milliseconds, which is reasonably low for read operations of a whole SDNApp state. Similar considerations can be done for write (`set_node`) operations (Figure 6.8c), which in general require slightly higher times.

Figure 6.9a depicts the proportionate delay overhead introduced by the ToY Agent normalized on the baseline direct access to SDNApp variables. The graph shows that, especially for write operations, there is no direct connection between the proportionate overhead and the depth level of the accessed node, i.e, increasing the depth level does not necessarily lead to a higher percentage of overhead compared with direct access. This result suggests that the algorithm is more susceptible to the particular nature of the data that each request has to manipulate (e.g., differences between the YANG and the internal representation, data structures used to implement object collections, etc.).

Finally, please note that this overhead does not refer to any particular application since it is based on measurements taken using a large data model that features arbitrary data structures. The correlation between the overhead and the specific use case, along with considerations about the involved data structures, will be analyzed in Section 6.6.2.

Programming Overhead

The overhead in terms of additional line of codes that the two approaches introduce to the original SDNApp is shown in Figure 6.9b. In Section 6.4.3 we have

already shown that 3 lines of code are enough to import and run our ToY Agent into an existing SDNApp. On the other hand, the "ad-hoc" application has been extended with a minimal Jersey REST server that is integrated with ONOS. This required a few extra classes with a considerable amount of additional code, as long as some modification in the original SDNApp source code. Ad-hoc modifications are detailed below.

For the REST interface, we needed two small classes plus the implementation of a controller object, featuring two methods for every single object and variable to be exported. In our implementation, we needed 9 lines of code on the REST controller for a minimal `get()` operation and 15 lines for each `set()`. Of course, this grows exponentially with the depth level (for our setup, we implemented just the methods needed to measure the performance, i.e. two for each depth level). Since ONOS manages the REST service through an inversion of control paradigm, no additional code has been required to run it. However, getter/setter methods have been added to the attributes of the root class, to allow the REST controller to access them (4 attributes, for a total of 32 lines). Most importantly, all existing classes have been extended with two custom methods to perform JSON parsing and serialization (≈25 lines per class, depending on the number of attributes to be exposed). Totals for each depth level are shown in Figure 6.9b. Since the code overhead in the ToY Agent is extremely low and constant (3 lines of code), the graph also takes into account the lines of the PMT, despite this not being an application invasive modification. The figure clearly assesses that, from the application maintainer point of view, the programming overhead to enable read and write access through our approach is negligible compared with an ad-hoc manipulation of the SDNApp.

Notification Delay

As described in Section 6.4.2, ServiceApps may subscribe to desired resources to be notified when something within them changes. Since variables are monitored with a polling mechanism, we measured the notification delay together with the CPU consumption of the system, on the variation of the polling frequency.Figure 6.9c summarizes the results taken over one hundred samples. The graph shows how, obviously, with a growing polling period, the time needed to receive the notification increases, while the CPU consumption decreases (CPU consumption values are shown in the vertical axis on the right side). Measured values show that with a polling period of ≈0.8 s or greater, CPU consumption remains almost constant (with a slow decrease from ≈0.2% to ≈0.1%), while the notification latency starts to grow the trend is logarithmic, but the curve slope is significantly high up to a polling period of 5 s). A good compromise, therefore, is a polling period of 0.8 s, which provides low CPU consumption (≈0.2%) with a reasonable average notification latency of ≈0.5 s.

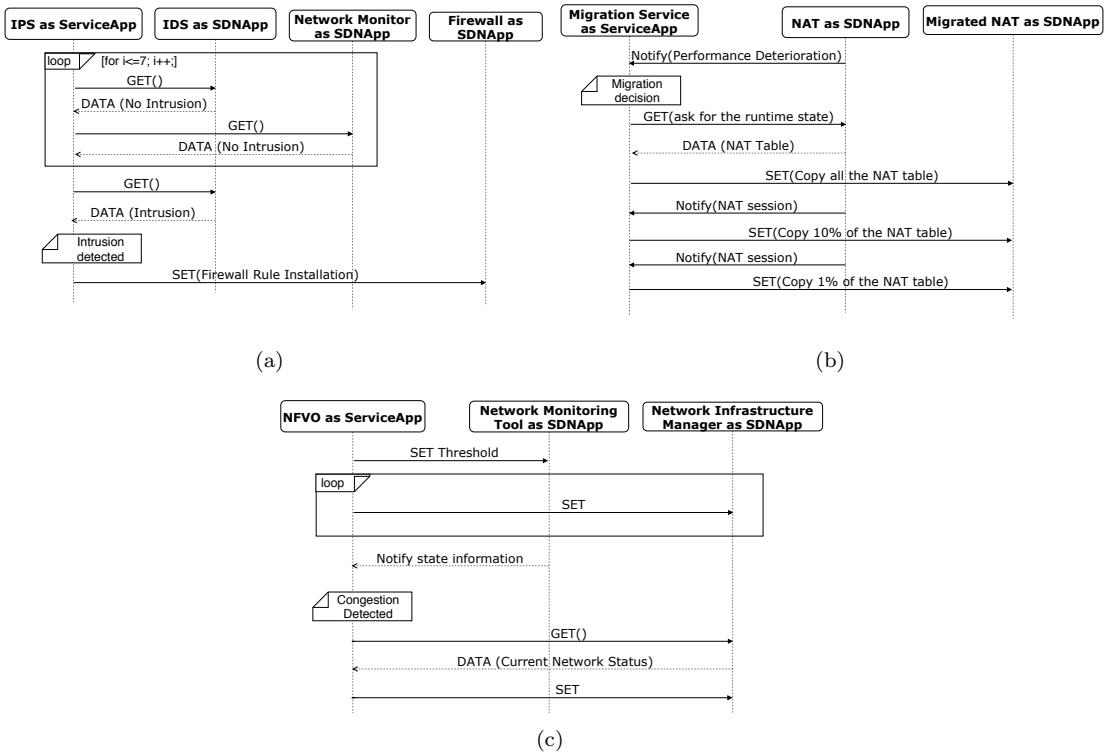


Figure 6.10: Use case workflows for (a) IPS (b) Service Migration and (c) Service Orchestration scenarios. The figures detail the list of operations that ServiceApps execute on SDNApps.

6.6.2 Use Cases Evaluation

In this subsection, we compare the performance of the ToY Agent against ad-hoc direct access in some specific scenarios, to evaluate the applicability of different use cases. For this purpose, we rely on the use cases already discussed in Section 6.3, whose workflows are detailed in Figure 6.10.

IPS workflow. In Figure 6.10a we assume that the ServiceApp (IPS) periodically performs *read* operations toward the IDS and the Network Monitor SDNApps to collect, analyze and aggregate the network status. Once an intrusion is detected, the IPS performs a *write* operation on the firewall SDNApp to install a new rule to protect the network infrastructure.

Migration workflow. A migration service (Figure 6.10b) waits for threshold-based notifications from the NAT SDNApp, which suggests a possible deterioration in QoS that requires a migration. At that point, the ServiceApp first performs a *read* operation to acquire the current run-time state of the existing NAT instance. Then, it writes the whole state to a new SDNApp instance. We then assume that other smaller *write* operations will be executed, e.g., to update additional parameters that changed in the meanwhile.

Orchestration workflow. In this scenario (Figure 6.10c), the orchestrator executes some *write* operations to setup virtual links connecting VNFs of a network service. The orchestrator may set some QoS threshold values according to negotiated SLA (e.g., maximum bandwidth for virtual links). When an SLA is violated (e.g., due to congestion in the network), the orchestrator receives a notification from the network monitoring tool. Suddenly, it first fetches the current status of the network and then, through a *write* operation, properly configures flow entries for the virtual link that is suffering from the congestion, thus satisfying the SLA.

From the above descriptions, we conclude that each use case has different characteristics in terms of interactions between the ServiceApps and SDNApps, and in terms of exchanged data. For what concerns the IPS scenario, as stated in [161], a tuned IDS produces an average of 3000 alerts per day, among which only 13.2% report an intrusion. Based on these data, we can assume that, for each 7 *read* operations, each performed every 30 seconds, 6 of them do not report any intrusion, thus exchanging tens of bytes. Once an intrusion is detected, a slightly bigger *read* that retrieves all the related information is performed, followed by the small *write* operation that sets the firewall rule. These data are modeled with depth 1 instance nodes. On the contrary, in the migration service scenario, a threshold-based notification is immediately followed by the decision of migrating the SDNApp, to avoid performance deterioration. At this point the ServiceApp and the SDNApp exchange huge amount of data compared to the previous scenario, to migrate the application state (the whole NAT table in a big network may consist of around 10K of entries). The *write* operation which sets this amount of data modeled with a 3-depth instance node requires hundreds of milliseconds if the incremental overhead of all the table entries (depth 1) is also considered. The subsequent *write* operations regard few entries of the table and then require less time (tens of milliseconds). Finally, the service orchestration scenario mainly requires *write* operations that exchange only a few bytes (value of the threshold, flow rules parameters) and which require less than one millisecond. The number of *write* operations necessary to install the Network Service depends on the particular scenario. In this numerical evaluation, we have considered the scenario described in [67], where 5 virtual links are installed.

According to these assumptions, we evaluated, for each use case, the overall time necessary to execute all the operations described above using both *(i)* the ToY Agent and *(ii)* ad-hoc manipulated applications with direct access to run-time state. Results are reported in Table 6.4. Moreover, to visualize the overhead that the TOY Agent introduces in each use case, we normalized the overall delay to the baseline value needed with direct access to application variables (Figure 6.11). A particular high overhead has been registered for the IPS scenario compared to the others. It appears that the high proportion of *read* operations that characterizes this use case constitutes a penalty for using the ToY Agent. In fact, as discussed in Section 6.6.1 (Figure 6.9a), mapping algorithms seem to suffer, on average, more for *read* operations than for *write* ones, when compared to direct object access.

Table 6.4: Delay introduced by the ToY Agent for three different use cases compared with direct ah-hoc access (average times taken over thousand samples are shown).

Use Case	Direct	ToY Agent
IPS	0.86 ms	1.59 ms
Migration	1516.41 ms	2298.81 ms
Orchestration	0.81 ms	1.13 ms

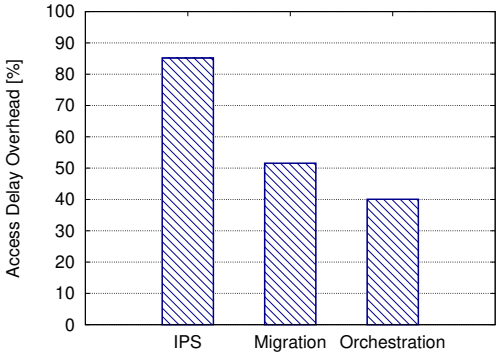


Figure 6.11: Overall percentage overhead introduced by the ToY Agent for three different use cases.

6.7 Discussion

Experimental validation demonstrates that our approach achieves its objectives while guaranteeing minimum implementation effort and reasonable execution overhead. More specifically, developers of SDNApps need to perform very few modifications to their source code to enable automatic state inspection (i.e., only 3 lines of codes were added to the `SSSA-analytic-tool` application, as shown in Figure 6.7). Furthermore, as shown in Section 6.6, the execution time of the above algorithms is reasonably low (i.e., units of milliseconds for SET/GET operations). It is worth noticing that the proposed approach is rather general, even if we limited our focus on SDNAppsin this work. In fact, the creation of an abstract data model for each application and the capability to provide automatic access to its internal data may be useful in other contexts beyond SDN applications and controllers.

Our work pointed out several requirements for current SDNApps to be able to exploit the capabilities of our state inspection library, to enable the seamless adaptation of existing applications to the new service orchestration architecture. First, it requires SDNApps to be thread safe, e.g., to avoid that the ToY Agent modifies the value of a particular variable while the application is executing a critical section. Second, SDNApps must be written in a language (e.g., Java, Python) that supports reflection to access variables or a similar mechanism, such as Java Reflective APIs [62].

Some limitations of this approach can be envisioned as well. First, the State Agent must use polling cycles to discover any modification to relevant run-time variables of the SDNApp. According to our tests, this may have a negligible impact in terms of CPU cycles in case the update frequency is kept on the order of tens of milliseconds (which looks reasonable for most of the applications), but it may worsen in case higher frequencies are required.

Second, the language that has to be used to create the *object path* in the PMT is

rather complex. This is due to the possible mismatching between the YANG data model the internal structure of the application. However, in our experience, the effort required to design a YANG-native interface and implement the required interactions with the northbound API (e.g., in case the SDNApp has to be developed from scratch) is much higher than the one guaranteed by our approach.

Third, our approach is not available as long as precompiled applications are concerned, which may be rather common when VNFs are deployed in a cloud-based data center. However, our experience shows that also the above applications can be ported to the proposed architecture by using a mixture of bash scripting, monitoring of in-kernel structures, and writing the equivalent of a wrapper ToY agent that implements the required northbound interface. For instance, the firewall used in Section 2.5 is an `iptables`-based VM integrated with a preliminary version of the architecture presented in this chapter.

6.8 Conclusion

In this chapter, we presented an approach to make the internal state of SDN applications available to external services, granting both read and write access to run-time variables and also the possibility to receive notifications with state updates any time something relevant changes. This gives service providers more flexibility while exploiting infrastructure facilities to composing their final service workflows.

To model the arbitrary, possibly complex, run-time state of SDNApps into high-level structures, we used the YANG modeling language. A set of algorithms maps the low-level data structure of the SDNApps state into the high-level YANG one. Such algorithms are independent of the particular structure and rely just on information kept into the SDNApp specific YANG model and Path Mapping Table. This allows applying such an approach to arbitrary SDNApps, without introducing changes on the application code. Performance validation showed that the delay introduced by the and mapping algorithms scales well even for unusually big YANG models with high depths. By analyzing the applicability of this approach on different use cases, it appears that the introduced access overhead is reasonable for most scenarios. Moreover, numerical data show that the delay overhead is widely compensated by the few programming requirements, in terms of additional lines of code, compared with an ad-hoc manipulation of each application.

Although our validation has been carried out in the SDN context, this work can easily accommodate other kinds of applications as well. This may allow high-level services to exploit any kind of network service across heterogeneous infrastructures (e.g., SDN-based networks, data centers, modern CPEs).

Chapter 7

Service management on disrupted infrastructures: Æther

In this chapter, we overview the applicability of new generation service facilities in the particular case of Industry 4.0. Indeed, Internet of Things is bringing flexibility for service management in an increasing number of industrial scenarios, such as mining operations, building sites, smart agriculture, and more. In such scenarios, the infrastructure is constituted by fleets of heavy-duty vehicles, which communicate utilizing opportunistic connections that exploit their movements and available transmission technologies, since a fixed network infrastructure is often not available. Enabling service management in such kind of network introduces several challenges: multiple communication technologies (and even protocol stacks) are used to establish opportunistic connections; latency may or may not constitute a constraint affecting routing; existing applications commonly adopt protocols that rely on end-to-end connections (e.g., MQTT, HTTP); appropriate service discovery strategies should be adopted. This chapter proposes Æther, a service-oriented communication system that enables IoT services to transparently operate on a disrupted network, by providing Service Discovery, Virtual Services, and service-based routing optimization, with advantages over state-of-the-art algorithms. Æther also provides a framework that enables both the development of new services, as well as support to existing applications through transparent protocol gateways. We tested our prototype both on physical devices and through larger-scale simulations, assessing the advantages of Æther and its applicability in real scenarios.

Part of the work presented in this chapter has been first published in [39] and [38].

7.1 Introduction

Industry 4.0 is gaining momentum by facilitating operations in many areas through the flexibility of the Internet of Things (IoT). The availability of exploitable services on board of devices on the field and industrial equipment enables a domain of new possibilities for smart deployment and automation. However, guaranteeing stable communication between end devices is not trivial in many Industrial IoT (IIoT) scenarios. Indeed, operations are often conducted in challenging environments, e.g., remote mines, building sites, large fields in smart agriculture, and more. In these scenarios, end devices may be fleets of heavy-duty vehicles, construction equipment, tractors, or even smart agriculture sensors, which may not always be connected to fixed and reliable network infrastructure.

In such environments, end devices may spend long periods (even their entire operating cycle) in remote locations, without access to a WAN connectivity. As shown in Figure 7.1, communication usually occurs over small-range wireless channels (Wireless Mesh Network) and the network infrastructure presents the characteristics of a Delay/Disruption Tolerant Network (DTN). In DTNs, devices exploit their movements to establish *opportunistic connections*, in order to exchange data during occasional contacts. Messages are steered towards their destination through intermediate relays by means of the *store-carry-forward* paradigm.

To enable deployment and consumption of heterogeneous IIoT services in such kind of environment, several challenges must be solved. Diverse communication technologies may be involved to establish opportunistic connections: some devices may use IPv4, other IPv6, some not even features an L3 layer (e.g., Bluetooth). Latency may or may not constitute a constraint: this is highly dependent on the nature of the service to be consumed or of the operation to be performed and may somehow affect the optimal routing decision. Existing IoT applications rarely are suited to be used on disrupted networks, as they usually rely on the existence of end-to-end connections (e.g., MQTT protocol). Finally, the heterogeneity and dynamicity of services to be provisioned in such disrupted networks require to take into account sophisticated service discovery paradigms.

To jointly address these challenges, we propose Æther, a service-oriented communication system that provides automation and flexibility to services that operate without the support of a fixed network infrastructure. Æther is transparent with regard to the underlying communication technologies and provides service-oriented facilities, such as Service Discovery, Virtual Services, and service-based routing optimization, with advantages over state-of-the-art algorithms. The framework enables the development of new services and transparent support for existing applications, with particular focus on the MQTT and HTTP protocols, which are largely used in IoT. We provide extensive experiments on our prototype, by both deploying on physical devices and performing large scale simulations. Our Experiments assess the applicability of Æther in real scenarios.

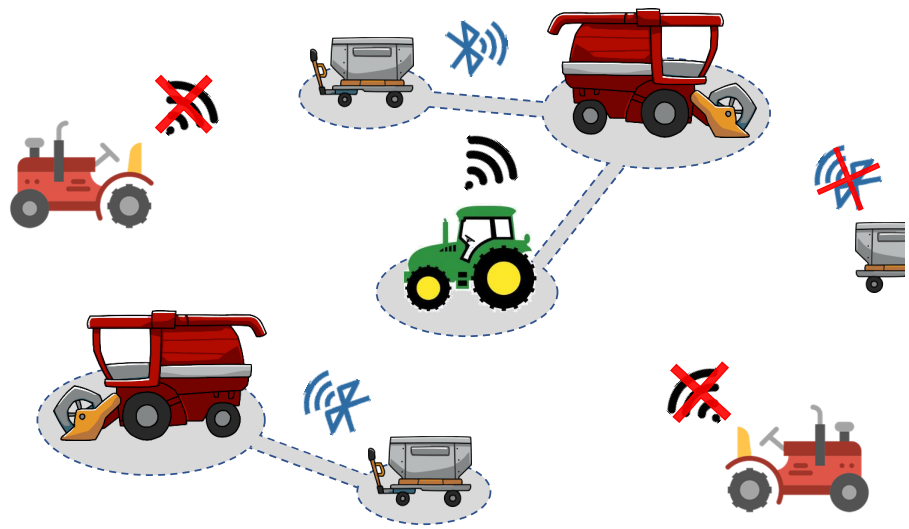


Figure 7.1: Opportunistic connections in a disrupted environment.

The remainder of this chapter is organized as follows. The next section highlights our contribution compared to the existing work. Section 7.3 presents the proposed communication architecture, providing details on its core modules. Section 7.4 describes all the higher-level facilities that constitute the Æther Framework, while our experiments are discussed in Section 7.5. Finally, we conclude in Section 7.6.

7.2 Related Work

IoT and Opportunistic Networks

The state-of-the-art on IoT in industries is summarized in [47], where authors describe key applications and identify the following as main technical challenges: service discovery methods and object naming services [11, 110], scalability on the number of connected things [110]; heterogeneity of underlying communication protocols [155]; lack of architectures for sensor networks communication, resilience to physical network disruption, and node peering [14]. Works such as [12, 102] propose to extend IoT connectivity over disrupted environments with the help of the DTN approach. Particularly, [12] provides an implementation of the Constrained Application Protocol (CoAP) over DTN to enable IoT devices based on CoAP to operate on a disrupted environment, while [102] analyzes the behavior of MQTT for Sensor Networks (MQTT-SN) over a DTN implementation. With Æther, we extend the

existing work by proposing a complete framework that operates over a disrupted environment, focusing transparency toward existing IIoT applications, heterogeneous communication protocols, service dissemination and routing optimization.

Service Discovery

In IoT, numerous protocols exist implementing service discovery. For instance, Zeroconf [153] provides automatic location of network facilities given an interconnected set of devices, while Universal Plug and Play (UPnP) [80] is a standard for pervasive peer-to-peer connection of services and devices. However, tools and standards are mainly designed for residential networks and may not be suitable for ad-hoc and disrupted networks. Indeed, the make large use of LAN-based facilities, and rely on protocols such as HTTP, thus requiring end-to-end connectivity. In the case of Mobile Ad-hoc Networks Service Discovery is a largely studied problem [164]. Main challenges and possible approaches have been identified and solutions proposed. Many existing approaches are based on directories, i.e., some intermediate nodes store service information and facilitate the communication between clients and servers. [152] provides a centralized directories approach, where one or more fixed nodes are used for the directory role, with limited flexibility. Multiple approaches have been proposed to implement dynamic distributed directories: some create dynamic backbones [93, 63], whose each node will eventually store up to date service information; others [90, 89, 138] implement distributed directories by forming hierarchical clusters of service providers, each electing a directory that will synchronize with peers; some solutions are based on distributed hash tables [141, 145], that is, service information matching a given hash key is always stored in a directory node of a particular region based on that key, which can be found by clients by using the same hash function.

On the other hand, directory-less approaches rely either on the broadcasting of periodic advertisement [117, 29, 95] issued by service providers (proactively) or on the broadcasting of service requests from clients (reactively) [114, 55]. Another category of Service Discovery approaches proposed in literature operate cross-layer, exploiting messages already exchanged by the routing protocol below to also embed service information on them. This can significantly reduce the communication overhead. [64] shows that an Ad-hoc On-demand Distance Vector routing protocol extended with service discovery features produces up to 50% less message overhead and up to 7 times lower discovery latencies. Benefits have also been measured in [10], where messages of an existing service discovery protocol were embedded in the routing messages. Noticeably, [72] proposes a novel routing protocol which natively provides its own service discovery. Taking into account such existing work on MANET, within our work we describe a Service Discovery protocol specifically designed to work in Æther which operates on a Disrupted/Deelay Tolerant Networks and is oriented to industrial scenarios. We adopt an integrated approach

where the service discovery and the routing module both benefit from interacting with each other in a service-aware environment.

Routing

Diverse approaches have been proposed to address the problem of routing in DTN [30]. Due to the particularly scattered nature of such networks, many algorithms adopt message replication. Epidemic [162] implements an unbounded replication of every carried message among nodes so that these eventually reach the destination. This provides maximum delivery ratio, but also high resource utilization. Spray and Wait [151] limits the overhead of Epidemic by allowing a maximum number of replicas for each message. The vanilla version only allows the first node to replicate, while others wait until the destination is met, while the binary version implements a propagation tree that allows a multi-hop relay. Based on the rationale that, in realistic scenarios, encounters between nodes are never totally random, some approaches calculate utilities to limit the replication process to convenient connections. This is the case of PRoPHET [98], which maintains a set of probabilities of successful delivery toward known destinations, thus replicating only when the encountered device features a higher probability than the local one. Another replica-based algorithm is MaxProp [28], which estimates the cost to a destination through link probabilities and order output queues accordingly (MaxProp is further discussed in Section 7.3.2). A different approach is taken by forwarding-based algorithms. To drastically reduce the resource utilization, such approaches attempt to route a single copy of the message towards the destination, usually sacrificing the delivery ratio. The basic idea is to identify the best relay before to forward the message, despite simpler implementations even select an arbitrary node. First Contact [81] always forwards to the first encountered node unless this already carried the same message in the past. In Seek and Focus [151], the message is forwarded at random in a first "seek" phase, which switches to a "focus" phase when a node that encountered the destination more recently is in proximity. Some forwarding-based approaches assume to operate over networks where nodes have access to global information, such as Inter-Platenary Networks. Here the most common approach is to compute shortest paths through Dijkstra taking into account movements over time. Examples are Delay-Tolerant Link State Routing (DTLSR) [49] and Contact Graph [9]. As part of this work, we propose a routing optimization on existing algorithms based on additional knowledge coming from the service layer. The rationale behind this approach is that Æther is a service-oriented architecture by construction, thus the routing layer may benefit from SLA information natively.

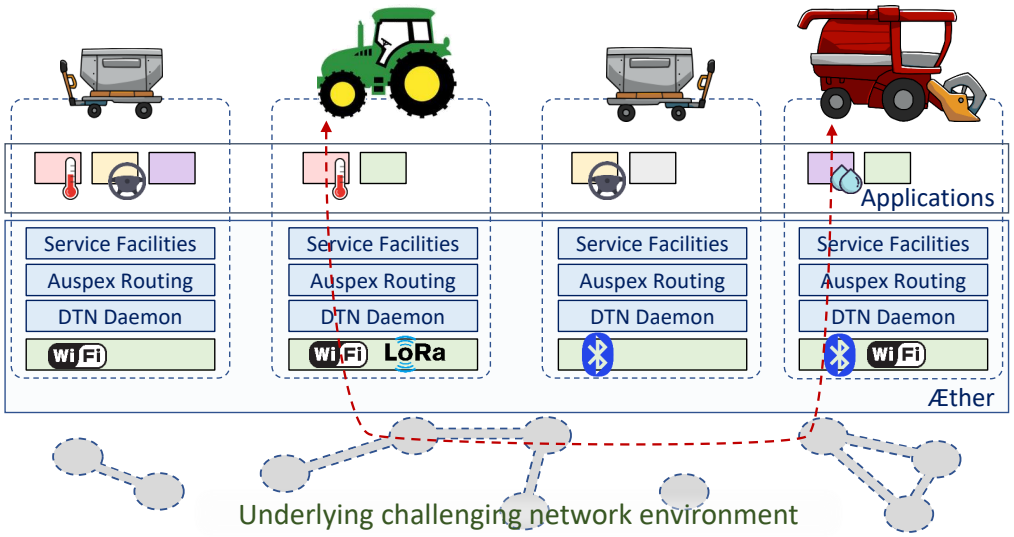


Figure 7.2: Overview of the Æther disrupted communication scenario. Applications on board of scattered devices can interact through the disrupted network environment using high-level service-oriented facilities.

7.3 Æther Architecture

The main principles of the approach adopted by Æther are depicted in Figure 7.2. Heterogeneous communication technologies available on each device constitute the Transmission Layer of the challenged network, on top of which communication is enabled by mean of a Delay/Disrupted Tolerant Network (DTN) Layer. On a DTN, the flow of data from a source node toward the destination is implemented exploiting the continuous movement of the operating machinery, and the ad-hoc connections they may establish (*store-carry-and-forward* paradigm). Data is delivered by the DTN protocol, which extends the existing network stack (e.g. TCP/IP, Bluetooth, etc.) with the *bundle protocol* layer. This encapsulates application messages, which are then delivered hop-by-hop to another DTN node, based on tunable forwarding behaviors [175]. Each Æther Node runs a DTN Daemon and is identified by an Endpoint Identifier (EID) (e.g., `dtn://device1`), while each application is identified by extending the local node EID with an unique *application token* (e.g., `dtn://device1/app1`).

On top of the DTN Layer, Æther implements a series of service-oriented facilities, that are, Service Discovery, Virtual Services, and service-aware Routing. These components, along with the DTN Daemon, constitute the core of Æther. Additionally, Æther provides a series of APIs that enable applications running on the device to exploit such facilities. Figure 7.3 details the architecture of Æther at the node level, highlighting its modules and their interactions. The remainder of this section details each core component.

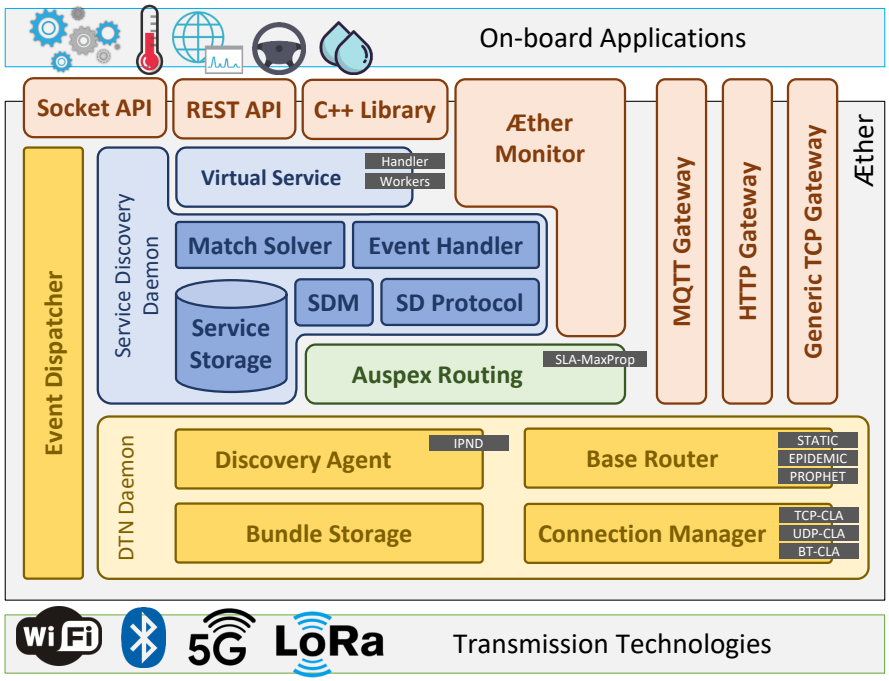


Figure 7.3: Æther node architecture.

7.3.1 The DTN Daemon

On each Æther node, the Bundle Protocol is implemented by the DTN Daemon. To deliver data from a source node to a destination one, this is first enclosed in a bundle and sent to other peers through a small-range connection. The decision to forward a bundle to a given neighbor (or even to a set of them) is based on the adopted routing algorithm. The forwarding process is repeated by the second node when it detects other opportunistic connections until the data carrier enters in range of the destination node; hence the data bundle is delivered to the destination (together with other bundles possibly collected in the meanwhile).

We build Æther on top of IBR-DTN [139], a project that implements the bundle protocol together with some basic DTN features. In the following, we provide some details on IBR-DTN, which are then followed by a description of some improvements and extensions we made to the original system in the first place, to use it as the base for our system.

The architecture of IBR-DTN is based on an event-oriented communication paradigm. An *Event Switch* takes care of dispatching each event to the module that should handle it. Supported events regard storage operations, changes in the state of the neighborhood, incoming bundles, triggering of routing operations, and more. The management of the neighborhood is demanded to the *Discovery Agent*, which implements the Internet Protocol Neighbor Discovery (IPND) [54] to identify nearby peers and perform the binding process, associating the network

address of the peer node with its EID. The actual communication with a neighbor node is instead managed by the *Connection Manager*, which handles sending and receiving of bundles by exploiting different *Convergence Layer Adapters* (CLAs). A CLA is responsible for the encapsulation of the bundle protocol on the higher layer of a given network stack (e.g., TCP, TLS). IBR-DTN allows implementing any additional CLA as a module of the Connection Manager. The store-carry-and-forward paradigm is implemented through the primitives offered by a *Bundle Storage* module. Finally, a *Base Router* takes care of the bundle routing task, which is performed according to the decision of the routing algorithm provided as a plugin. IBR-DTN already offers plugins supporting static, epidemic, and PRoPHET routing algorithms.

To build our system on top of IBR-DTN, we made some modifications that are now part of the main repository). Among the extensions, we enabled the DTN daemon to run the IPND Neighbor Discovery protocol on top of IPv6, so that device-to-device communication may occur without the necessity of an IPv4 address assignment procedure (e.g., DHCP). Additionally, we implemented a new Convergence Layer Adapter for Bluetooth, hence increasing the range of devices on which Æther can be transparently deployed.

7.3.2 Auspex Routing

As Æther features a given level of service awareness by construction, it is convenient to optimize the DTN routing based on service-oriented consideration, rather than merely rely on the statistics of the opportunistic connections. For this purpose, Æther provides the *Auspex Routing* module, which extends existing routing algorithms with service-oriented strategies and also provides useful data that can be exploited by other Æther modules such as the service discovery.

To design Auspex, we first performed a preliminary evaluation where we studied available protocols in DTN and compared their performances on different metrics. We set up a simulated scenario using The ONE simulator [87], with 19 devices moving within an area of 50×100 m^2 at different speeds (between 0 and 5 Km/h). Figure 7.4 shows that MaxProp outperforms other algorithms both on delivery probability and on the average time needed to deliver a bundle. Moreover, excluding First Contact, which is not a replica based algorithm (that is, it immediately delete a bundle on the first relay), MaxProp also features the lowest "storage time", i.e., the time that a bundle spends in the node storage. Similar results have been obtained even varying the setup on *(i)* number of nodes, *(ii)* devices boundaries, *(iii)* mobility patterns (more details can be found in [123]).

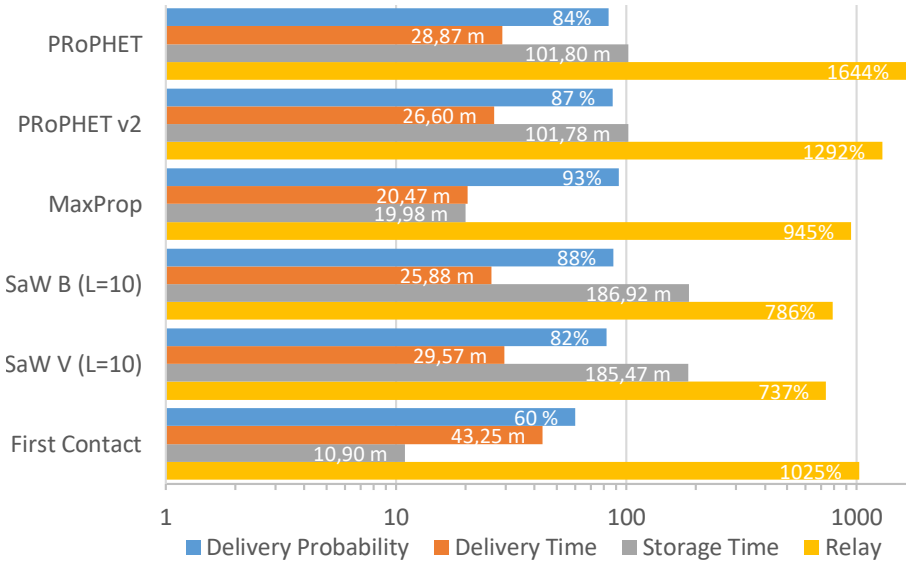


Figure 7.4: Performance of main DTN routing algorithms compared.

MaxProp

Based on the above considerations, we decided to first implement the MaxProp algorithm within the Base Router of the DTN Daemon. Then, we evaluated any room for improvement and designed Auspex accordingly. MaxProp is a replica-based routing algorithm, in that it replicates messages in the storage when a new neighbor is encountered. The replication process is optimized by the adoption of the following mechanisms. Each node estimates the probability to meet peers by maintaining a vector $\boldsymbol{v} \in \mathcal{R}$ of probabilities, where each value is initialized to $1/(n-1)$, where n is the number of known nodes. Whenever node j is encountered, v_j is incremented by 1, then all values are normalized so that their sum adds to 1. Such probabilities are exchanged during each connection and used to estimate the cost of a path $\boldsymbol{p}$ as $\sum_{j \in \boldsymbol{p}}(1-v_j)$. The cost to a destination is given by the path with the smallest total cost. MaxProp keeps bundles ordered so that when a transmission opportunity occurs, priority is given to those featuring a lower destination cost. An exception is made for younger bundles, i.e., those with a hop count less than a given threshold, which always obtain a higher priority. On the other hand, whenever new bundles should be stored, if the storage limit has been reached, bundles with the lowest priority are discarded. Once a bundle is delivered to its destination, an acknowledgment mechanism allows nodes to stop replicating that bundle and delete it from their storage.

Constraints Enforcing

Auspex operates on top of the DTN Daemon Base Router, optimizing the currently adopted algorithm (e.g., MaxProp) by *(i)* affecting its decision to replicate a bundle while in an opportunistic connection with a peer node, *(ii)* exchanging additional information within the protocol messages, *(iii)* extending the bundle header with any service-level constraint. Being the entire structure of Æther service-oriented, it is possible to access additional information concerning the bundles in the storage, such as the target service and the specific stream they belong to, as well as any service-specific constraint they should satisfy while being routed toward the destination. For instance, a given service on board of a sensor device may stream periodic updates on a measured value, which features a given temporal validity. Thus a given data becomes useless if delivered too late. By means of the Service Discovery, any service-related constraint may be fetched directly from the service description, without the need of a particular user interaction nor of a specific configuration of the service itself. Whenever an application generates a new bundle to be conveyed into the DTN, Auspex matches the source and destination apps with entries of the service discovery, searching for any constraint to be enforced. The bundle header is then extended with the proper service constraints before being stored.

On each device, the Auspex module will then have access to this information, which is used to enforce service constraints whenever a routing decision is taken. In the case of MaxProp, Auspex affects the decision of replicating a bundle during an opportunistic connection by skipping all bundles that would not be likely to satisfy all their service constraints by being replicated to that particular node. This effectively fixes the transmission priority so that nodes better exploit opportunistic connections replicating the correct bundles and identifying the proper contacts to convey particularly constrained streams. Section 7.5 provides implementation details for the use case of end-to-end delay enforcement. In our evaluation, we show that this approach effectively provides improvements over MaxProp in terms of net delivery, even if service-level constraints are enforced to only a subset of the messages.

7.3.3 Service Discovery

In Æther, devices can provide and/or consume any kind of service. A vehicle moving on farmland can be equipped with a sensor device, which provides, for instance, measurements on the humidity of the field. On the other hand, devices acting as actuators (e.g., sprinklers) may need to rely on the humidity service mentioned above to decide which action to perform in a given moment. Additionally, the sprinklers may expose an API to allow their activation from an external device (e.g., an operator with a web interface or another IoT device with a sophisticated algorithm onboard).

The above scenario requires that devices are aware of services available in their

vicinity, as well as of additional information associated with the services themselves, such as network-related QoS (e.g., latency and consequent time validity of data/actions), service-specific properties (e.g., the precision of measured data), location, energy-based information (e.g., device autonomy), level of reliability (e.g., how often the device with a given service is reachable in the disrupted network?), and more. In the following, we describe the Service Discovery (SD) system designed to work within Æther, which provides an efficient way to model service information, to exchange it among devices and to maintain a suitable set of such data on board of each device, so that they can easily benefit from existing facilities. The Service Discovery takes into account that the network is disrupted, then some services may respond with a too high delay or not to be available anymore. Our protocol should facilitate applications to specify preference on services to be selected, to automatically discard those that can neither respond within a certain time bound nor provide up-to-date data, to transparently replace services in case of failure, etc.

Service Description

We now detail the data model used to describe the details of a given service. The proposal takes into account existing works [164, 73, 17] specializing them for the Æther use case. Figure 7.5 depicts the complete Service Description Model (SDM), while most relevant fields are described in the following.

Æther uses *categories* to cluster services by means of a hierarchical structure. Due to the unpredictability of existing services, only a few general categories are fixed and known by everyone (Figure 7.6). Each service may specify additional subcategories extending the default tree. Since most likely a consumer would not know the exact category to which desired services belong to (aside to the possibility that equivalent services are registered to slightly different category paths), the SDM allows specifying multiple categories. Service categories may also help to aggregate information about existing services during advertisement propagation and state maintenance.

Other than categories, the service description features a list of *keywords*, which can help to match requests by providing alternative semantic descriptions of the service. For instance, a sensor that provides temperature measurements may feature the keywords “temperature”, “heat”, “thermal”, etc. The existence of keywords and categories simplifies discovery in a highly heterogeneous and not a priori configured environment.

An *expire* field specifies the temporal validity of the service description; if no new information is received within the expiration time, Æther nodes should discard the service entry.

The SDM also features information about the *device* running the service, such as its battery, computational load, whether the device is moving or not, current location and covered area (different shapes can be specified, options are not reported

```
DATA                                          | TYPE
----------------------------------------------|-------------
Service                                       |
+-- name                                      | string
+-- category*                                 | string
+-- keyword*                                  | string
+-- device                                    |
|   +-- node-id                               | string
|   +-- energy                                |
|   |   +-- capacity                          | int
|   |   +-- residual                          | int
|   |   +-- load                              | int
|   +-- geolocation                           |
|       +-- moving                            | boolean
|       +-- location                          |
|       |   +-- description                   | string
|       |   +-- point                         |
|       |       +-- lat                       | float
|       |       +-- long                      | float
|       +-- area-coverage                     |
|           +-- [...]                         |
+-- property*                                 |
    +-- name                                  | string
    +-- value                                 | any
    +-- unit                                  | string
    +-- operator                              | <, >, =
+-- function*                                 |
    +-- name                                  | string
    +-- description                           | string
    +-- input*                                |
    |   +-- name                              | string
    |   +-- description                       | string
    |   +-- unit                              | string
    +-- output*                               | list
        +-- name                              | string
        +-- description                       | string
        +-- unit                              | string
```

Figure 7.5: Service Description Model (SDM).

in Figure 7.5 for the sake of readability).

Service should be specified with its *properties*, which are used by consumers to understand if the service matches desired requirements (e.g., level of accuracy over a certain threshold). Properties are specified in the SDM through the following structure: *(i)* the *name* identifying the property, *(ii)* the actual *value* of the property (string, boolean or numeric), *(iii)* its type through the field *unit* (if numeric, the measure unit is given) and an *operator*, which provides an indication of how the value should be intended (specific value, such as measurement accuracy, or bound

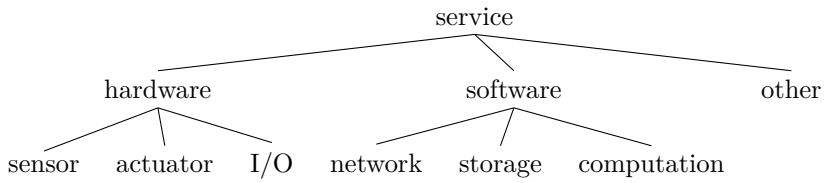


Figure 7.6: Default categories pre-defined in Æther service description. Other custom categories can be defined.

Table 7.1: Mandatory properties for predefined categories.

Property	**Categories**
Version	all
Protocol	all (one or more can be specified)
Capacity	all software categories (network throughput - MBps, storage - GB, computation - CPU/Memory/GPU)
Physical Quantity	sensor and actuator (string)
Refresh Rate	sensor (seconds)
Precision	sensor (same unit of functions output)
Delay	actuator (seconds)

on a certain quantity that the service may provide, e.g., maximum temperature that can be measured). Predefined categories also feature a set of mandatory properties to be specified (see Table 7.1).

A service, in its general form, may provide different functional invocations. For instance, a temperature sensor may allow to fetch just the last measured value, to wait until the device moves to a particular location of where the measurement is needed, or even to start streaming periodic temperature measurements. All these possibilities are described through a list of *functions*, each one providing: *(i)* the *name* identifying the function, *(ii)* an human-readable *description* of the function, *(iii)* a list of *input* required to invoke the function (e.g., location of which the temperature measurement is required), and *(iv)* a list of *outputs*, i.e., information returned by invoking the function (e.g., the temperature value). Both inputs and outputs are described through a name, a description, and a *unit* field, which specifies the measuring unit, or merely its data type.

Discovery Approach

In Æther, nodes periodically exchange information about the list of all know services, updating the local Service Storage. At the same time, entries of the local storage are enriched with further information regarding the involved remote nodes,

collected by interacting with the Auspex Routing Module. The Service Storage enables customers to be aware of existing services when they need them (proactive discovery), by querying the local *Service Discovery Daemon*, which maintains information about known services available in the network. However, a node may also be configured not to store any information about existing services, e.g., due to constrained resources on board. In such a case, the interaction between a service and a customer is mediated by the intermediate nodes (directories), which, upon an explicit service request (reactive discovery), may have a fresh enough matching entry in their service list. Services are advertised with a period that is tunable and highly dependant on the mobility level of the particular network.

Upon a service query is received from an onboard application, the SD Daemon may *(i)* find a match in the local list of known services or *(ii)* propagate a service request in the network, acting in reactive mode. In the first case, all relevant fields of the request are used to find matches on the list. At first, categories and keywords are used to fetch services of the desired type; if a specific function is specified on the query, the SD also filters out services not featuring it. Then, any additional query parameter is searched among the properties of each service description, and the proper comparison is performed (based on the operator field of the property and the query specification). A separate check is performed if the query features a maximum delay specification: in such a case, the SD Daemon interacts with Auspex in order to identify nodes toward which the statistical latency exceeds the desired one, and all services located on those nodes are filtered out.

Sometimes an SD query may be propagated as a request toward the network (reactive mode). This may happen whether because the node is not storing exchanged services information locally, or none of the known services matches the query. Since such messages are forwarded in multicast across the DTN, it is important to limit its propagation range to avoid excessive network overhead, particularly in scenarios where a high number of nodes act as customers. To do so, bundles encapsulating SD requests messages are configured with the proper number of maximum hops propagation, which is dynamically calculated exploiting the Auspex routing module. This is done by detecting the farthest destination that can be likely reached still satisfying any delay requirements specified in the service, and the request range is then set accordingly This process also considers the additional overhead introduced by the SD. If the request features no delay requirement, the range is first set to a small value (based on the particular network size) and iteratively incremented whether a match is not found nearby.

Messages

Table 7.2 lists messages exchanged through the SD protocol. The *Advertisement* message carries a list of services, each formatted as described in Section 7.3.3. It is sent both when the periodic advertisement of all known services is performed (in

Table 7.2: Service Discovery Messages.

Message Type	Description	When
Advertisement	List of services, each as in 7.3.3.	*(i)* periodically, all known services (multicast); *(ii)* as response to a *request*, only matching services (unicast).
Request	All fields are optional filters: *name* of an already known service; *categories* and *keywords*; *specifications* to be matched by the service properties; *functions*.	(i) with empty filters, used as solicitation (unicast); *(ii)* whenever a needed service cannot be solved locally.
Remove	Identifier of the service and timestamp.	When a previously exposed service is withdrawn.

multicast), and in response to a request message (in unicast). In this last case, the advertisement message only contains the services matching the request.

The role of the *Request* message is twofold. When sent in unicast (and with all fields unset), it solicits a neighbor node to send back all known information; this is sent from new nodes joining the network for the first time in order to quickly populate their local database. Instead, when sent in multicast it is used to search for a particular set of services in the network, based on fields specified in the request message. The field *name* is used when the goal is to locate a particular (already known) service, as the results should match the above-specified string. Instead, by specifying *categories* and *keywords* fields, provided results will have more flexibility. Additionally, the field *specifications* is used to filter on services satisfying certain properties, such as "protocol equals to MQTT", "precision less than or equal to 1 °C". A separate specification is the *delay tolerance*: this, when given, is handled by interacting with the Auspex module, which checks the eligibility of a given service based on available statistics on the communication between the two involved nodes. Further available filters regard the possibility to search for services providing a specific function.

When an application withdraws an exposed service, the local SD Daemon propagates a *Remove* message, which features the name of the service (identifier), and a timestamp. This explicitly instructs other nodes to discard the service entry, without waiting for its expiration time.

Module Architecture

The architecture of the SD module is depicted in Figure 7.7 and described in the following. A northbound interface enables local applications to perform the following operations: *(i)* get the currently known information from the local *Service Storage*, *(ii)* register a new service (that will then be advertised through the SD protocol), *(iii)* update or withdraw a previously registered service, *(iv)*

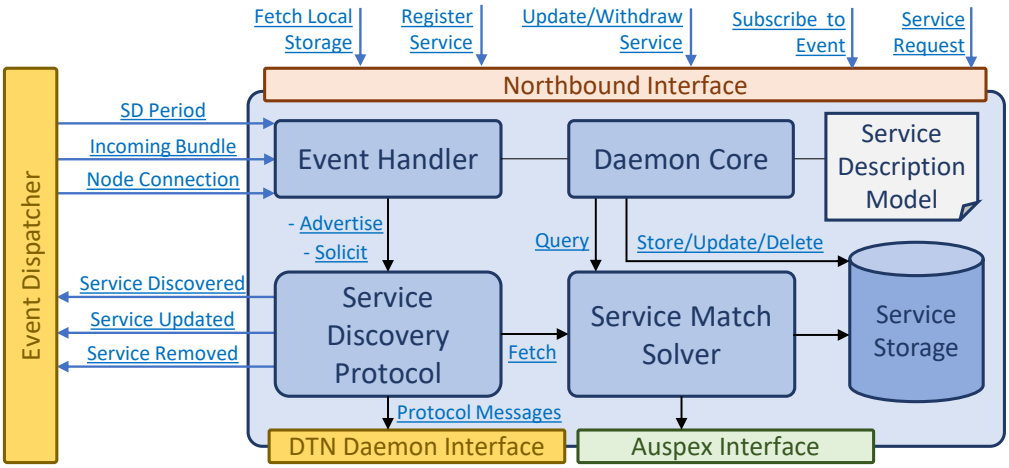


Figure 7.7: The Service Discovery module.

subscribe on events generated from the service discovery (e.g., listen for a particular service to be updated), and *(v)* request for a service matching the specified query. Such a northbound interface is then exported towards onboard applications by the REST and the Socket-based APIs (see Section 7.4.2). Additionally, the SD module interacts with the DTN Daemon in order to exchange protocol messages through bundles and to implement the protocol exploiting the Event Switch, which is used both to listen and generate events. Operations are triggered by an *Event Handler* on three conditions, which are described in the following. A `PERIODIC_SD` event, whose period is set based on the network characteristics, forces the local SD Daemon to prune all expired services from the local database and to propagate an advertisement of the currently known services. If the old advertisement message is still in the bundle storage, it is removed. The `INCOMING_BUNDLE` event may trigger different actions based on the received message: *(i)* if the message is an *Advertisement*, the database is updated accordingly and `SERVICE_DISCOVERED` and `SERVICE_UPDATED` events are triggered for any new/modified service in the message; *(ii) Remove* messages trigger similar actions, with the `SERVICE_DELETED` event; *(iii) Request* messages force the daemon to query the local storage by mean of a *Match Solver*, then building an advertisement message with compatible services, which is sent back to the request source. The SD daemon also listens for `NODE_CONNECTION` event, which triggers the generation of an (empty) Request Message, used to solicit the new neighbor to exchange its service database. The solicitation message is only sent if the node proactively stores information on existing services.

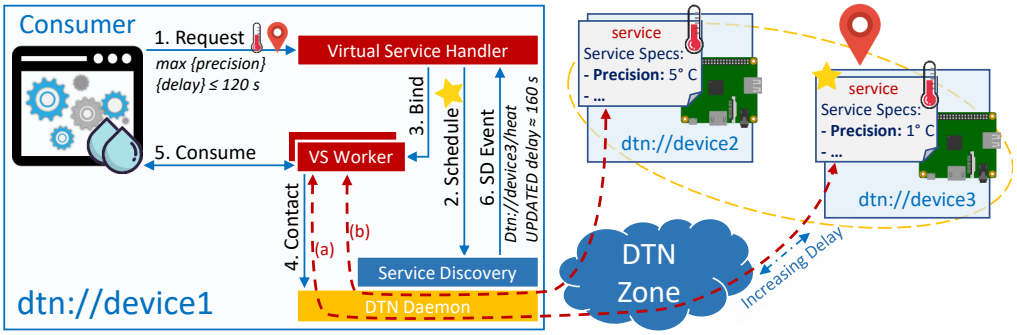


Figure 7.8: Workflow of a Virtual Service Request. The VS Handler first schedules the best available option according to the request parameters, binding a VS Worker on the selected remote service. Upon receiving an event that makes the currently bound service inadequate on the delay requirements, the Handler repeats scheduling and binding, making the worker switch to a different remote service (communication channel changes from *(a)* to *(b)*).

7.3.4 Virtual Services

Through the Æther SD system, an application is enabled to search the DTN for existing services, filter on desired properties and potentially implementing a strategy to dynamically select the one that best fits its demands and replace it in case of failure. However, this requires many interactions with the SD Daemon, as well as potentially complex scheduling algorithms to be implemented within the application, not to mention the additional complexity at the application level. Therefore, Æther features a Virtual Service module, which provides easy interaction with available services, implementing scheduling, optimization, communication and fault recovery in a way that is transparent from the consumer application perspective.

As shown in Figure 7.8, a *Virtual Service Handler* interacts with the Service Discovery module and exposes an interface toward the local applications. This component is in charge of *(i)* handling incoming requests from consumer applications, *(ii)* interacting with the SD to select the most appropriate service among the available ones, according to the customer specifications, *(iii)* running the SD Workers, each acting as a proxy between a customer and the actual service, and *(iv)* listening for events that may trigger the rescheduling of already bound services.

Virtual Service requests are similar to queries performed toward the Service Discovery northbound, except they *(i)* feature mandatory specification of the supported protocol and the description of the needed functions, and *(ii)* allow to specify one or more optimization parameters, which are used to privilege certain services over others (e.g., the closest one, the more accurate, etc.).

The *Virtual Service Worker* bridges the communication between a customer and the remote service currently assigned to it, forwarding messages whose content is compliant with the functions defined in the SDM. This way, the customer interacts with a local proxy on a stable channel, without caring about the actual node

hosting the service, and the disrupted communication. The worker also monitors the status of the remote service: whenever the performance goes below certain tolerated margins, which are given within the virtual service request (e.g., maximum tolerated communication latency), the VS Worker invokes the VS Handler for a rescheduling. Additionally, the VS Handler listen on events raised by the Service Discovery module: whenever a new service is discovered, or an old one updated, if this is a better alternative to one of the already bound ones, the corresponding VS Worker is updated so that it transparently switches to the new one. Similarly, whenever one of the bound services is removed, or it is updated so that it does not match the customer requirements anymore (as in the scenario shown in Figure 7.8), the VS Handler interacts again with the SD to find the next best match and update the VS Worker accordingly.

7.4 Æther Framework

Æther constitutes a framework for both the development and deployment of new generation IoT-based applications on top of the disrupted and heterogeneous infrastructure. The framework itself is service-centric, providing abstractions that make the interactions between services transparent with respect to both the network and the DTN architecture built on top of it. This section describes the abstractions, tools, and APIs through which Æther allows applications deployed on top of it to exploit the features provided by its core modules.

7.4.1 Protocol Gateways

Æther supports also traditional TCP/IP applications not compatible with the bundle protocol, which communicate through the underlying network by mean of a *Protocol Gateway*. Its purpose is to hide to the application the fact that its communications occur over a DTN, i.e., that in a given moment, the destination of messages at the application layer could not be reachable at all. In this way, provided that the proper gateway for the protocol used by the application is available, this can communicate over the disrupted network without any modification, as the gateway will be in charge to transparently convey its messages in the DTN, and deliver back the expected responses whenever they are available.

Æther provides gateways for both MQTT and HTTP, as these protocols constitute the standard for communication at the application level in IoT. Besides, the system features a more generic gateway able to propagate TCP datagrams.

MQTT Gateway

The MQTT Gateway collects messages generated on board of a device (e.g., a sensor), which need to be routed toward the device running the MQTT Broker.

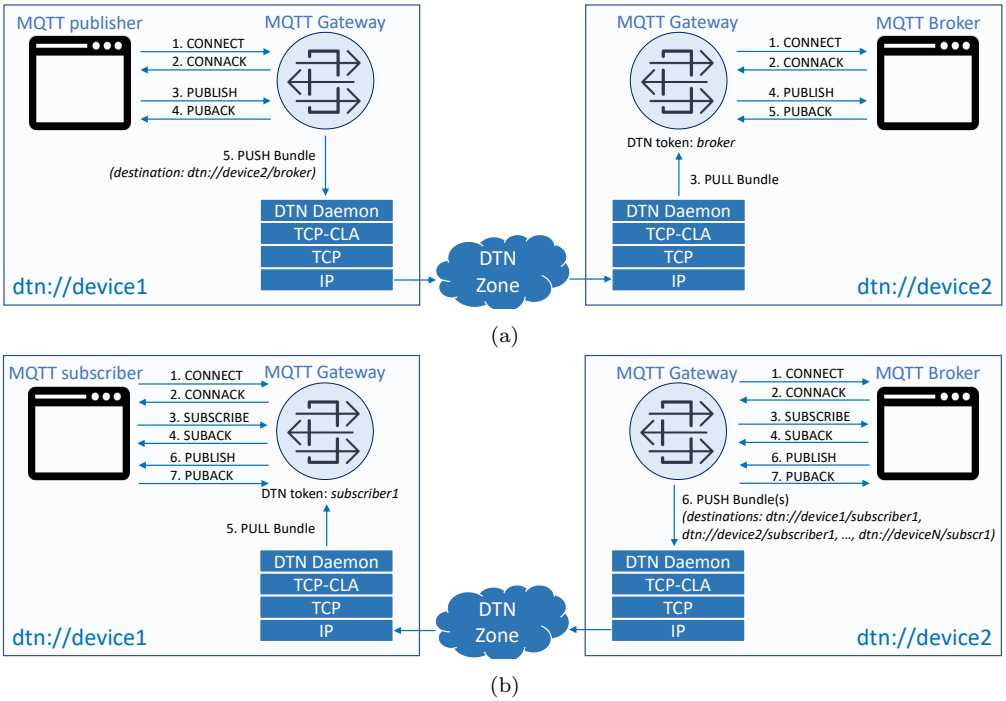


Figure 7.9: MQTT gateway in (a) upstream and (b) downstream communication.

The MQTT Gateway architecture was first presented in our preliminary work [39].

Figure 7.9a details both the architecture of an end device that produces and publishes data and the one of the node hosting the broker receiving such data. On both the devices, the DTN Daemon extends the network stack with the support for the bundle protocol and with a TCP Convergence Layer Adapter. On the end device, an MQTT Publisher application generates data that is first routed to the local MQTT Gateway. The role of the latter is twofold: it *(i)* emulates a local MQTT Broker thus allowing the application to transparently connect and publish data, apparently relying only on the MQTT protocol, and (ii) encapsulates received messages into bundles with destination `dtn://deviceX/broker`, which are sent to the DTN Daemon. Bundles are then pushed on the output queue, thus making them ready to be forwarded to other Æther nodes. Similarly, the MQTT Gateway located on the destination device acts as a bridge between the DTN stack and the MQTT protocol: it *(i)* registers the application token, namely "broker", on the DTN Daemon in order to "extract" all bundles with destination `dtn://deviceX/broker` from the DTN and *(ii)* sends data received in this way to the broker application, by performing an MQTT publish that preserves the original MQTT payload.

On the other hand, messages flowing from the Broker to an MQTT Subscriber may have multiple destinations, as there may be more than one subscriber on the same topic (e.g., firmware update on a specific device model, a fleet consuming a

particular service, and more). In this case, some more information is needed to travel toward the node hosting the Broker, as opposed to the normal data flow. In particular, the broker-side MQTT Gateway needs to receive information about existing MQTT connections and the details of any subscribe message that has been generated. To collect such information, a mechanism similar to the one described for the upstream communication is used, enabling the second MQTT Gateway to know which topics have been subscribed by each device and to perform, for each of them, the actual *subscribe* toward the local broker. Eventually, each *publish* message that is forwarded by the broker is mapped by the local Gateway to the correct destination EID and propagated into the DTN. This mechanism is detailed in Figure 7.9b.

Note that in both cases, applications operate on an abstracted MQTT layer that transparently allows them to send MQTT protocol messages regardless of the presence of a network connection.

HTTP Gateway

This gateway encapsulates HTTP data within the bundle protocol, enabling transparent communication between any couple of web client and server. This is particularly useful as REST APIs are widely adopted by existing services, and, additionally, deployed devices often feature web-based configuration dashboards, whose access from any terminal connected to the DTN could be beneficial. The HTTP Gateway features a Name Resolution module and the actual bridge that conveys HTTP data within bundles. On the client-side, the gateway acts as a proxy, intercepting every request whose URL follows the pattern `<app>.<device>.dtn/<resource>`, from where the destination EID can be reconstructed. Such requests are handled as follows: *(i)* a new bundle is initialized through the Æther socket-based APIs (Section 7.4.2), with destination EID set to `dtn://<device>.<app>`; *(ii)* all the headers are copied in the bundle, together with the URL and the request body, if any; *(iii)* host is always set to "localhost", as the original one will not have any meaning from the server-side; *(iv)* the bundle is finally sent toward the DTN network using the socket-based APIs.

Server-side, the Gateway is configured to manage a known list of web services, which can be local or remote (in case a non-DTN connection is also available on the device). The gateway listens for any bundle that features as destination an EID associated with any of the web services on the list. This configuration list must feature, for each EID, *(i)* the associated IP address, and *(ii)* the TCP port where the service is listening. The gateway handles incoming bundle by reconstructing the HTTP request based on the information kept in the relevant entry, thus forwarding the requests to the correct service and interacting with the DTN Daemon to initialize a new bundle, which is used to write back the HTTP response.

The implementation of the HTTP Gateway is based on Tinyproxy [160], which

has been modified to interact with the Æther bundle interface and extended with the support to HTTP/1.1. This was needed to enable persistent connections (with support to the `Connection:Keep-Alive` header), to send multiple requests in pipeline, and to support the CONNECT method and WebSockets.

Generic TCP Gateway

In addition to application-protocol specific gateways, Æther also extends any kind of application that communicates over the TCP protocol to operate over the disrupted environment. The gateway keeps a port table that is feed with information from the underlying Service Discovery. Whenever a service featuring the TCP protocol is discovered, the Gateway looks for the *tcp-port* and *ip-address* properties, and stores them in an entry with the corresponding EID from the service description. When the gateway intercepts the SYN segment from a local client featuring a remote destination matching one of the entries of the table, it first handles the TCP handshake locally, then starts encapsulating all data of the stream in bundles that are conveyed to the DTN Daemon. The destination EID is set according to the information available in the entry fed by the service discovery. On the destination node, if the server is bundle-aware, this will directly receive the content from the DTN Daemon. Otherwise, a server-side TCP gateway will demultiplex received data to the correct server application, opening a local TCP connection and forwarding the content of all the received bundle, as well as encapsulating all responses in bundles that are sent back to the client-side gateway. Additionally, the server-side TCP gateway is also in charge to register the local servers to the Service Discovery module to be announced.

7.4.2 Framework APIs

Apart from the Gateway interfaces described above, mainly exploited by services that are agnostic to the disrupted network below, applications running on the device may also directly interact with Æther using bundle-aware APIs. These allow exploiting all the facilities offered by the system, i.e., Service Discovery, Bundle Protocol, Events, and more.

The IBR-DTN framework already features *Socket-based APIs*, which require to open a TCP connection with the daemon. This is used by local applications to exchange bundles with the network and subscribe to events. Building on top of IBR-DTN, we extended such interface so that it also provides the possibility to interact with all the additional Æther modules described in Section 7.3. The interface exports all the commands available in the northbound of the Service Discovery module, allowing to receive notifications when the SD generates an event (i.e., discovered a new service with given properties filter) and to interact with the Virtual Service facilities.

Given the recent trend in software development that privileges REST interfaces, which provide more flexibility than textual-based sockets, our framework also provides such interface, with support to the use of WebSockets to register to events. Finally, C++ libraries support the development of native applications.

7.5 Experimental Results

To evaluate Æther, we performed both experiments on our prototype implementation and larger-scale simulations. We set up multiple use cases, which are described below, to evaluate different components of our architecture.

7.5.1 Prototype - on Physical Devices

We performed some tests deploying Æther on top of some physical end devices. Such tests aim to measure the communication performance and any overhead introduced by our system and, specifically, by its components to the baseline network properties.

Device Connection

We measured the time needed by the devices to complete the connection binding over WiFi and Bluetooth during an opportunistic connection between two devices. The test-base consists of two Raspberry Pi B+ v1.2, equipped with a WiFi dongle based on the Realtek RTL8188EU chipset, and Bluetooth 4.1. For what concerns the connection through Bluetooth, already paired devices require between 362 and 378 ms to establish the RFCOMM connection, with 99% confidence. On the other hand, unpaired devices take between 1602 and 1691 ms to establish the connection, considering a *Secure Simple Pairing 3.3* procedure based on passkeys exchange. Connection over WiFi is performed using the Ad-Hoc mode, through the *IBSS merge* procedure. In all our measurements, the communication channel was established in less than 270 ms. In both cases, the additional time needed to complete the DTN binding and start the actual communication highly depend on the timeouts of the IPND protocol, which, in our setup, sends beacons every 1 second. This suggests that our prototype may be suitable also in case of occasional pairing connections that last a few seconds, which may be the case in some challenging environments.

Bluetooth CLA Overhead

We measured the overhead introduced by our implementation of the CLA for the Bluetooth communication medium. To measure throughput, we performed some tests transferring a file of size 1 MB, varying the segment size. Results are

summarized in Table 7.3. Instead, latency has been measured through the `dtnping` tool (available in IBR-DTN), transmitting 1500 bundles with size 64 bytes. In this case, we compared Bluetooth with the TCP/IP CLA over WiFi ad-hoc (Table 7.4). Results show that performance is, overall, better on WiFi due to the higher link speed, despite such configuration also features a higher dispersion. In general, the experiments have shown acceptable results on Bluetooth, as they are proportionate to the lower speed of the physical medium.

Table 7.3: Throughput on the Bluetooth CLA.

Segment Size (bytes)	512	1K	4K	10K	100k
Throughput (Mbit/s)	283	361	669	676	675

Table 7.4: RTT (ms) over the BT-CLA compared with TCP/IP over WiFi ad-hoc.

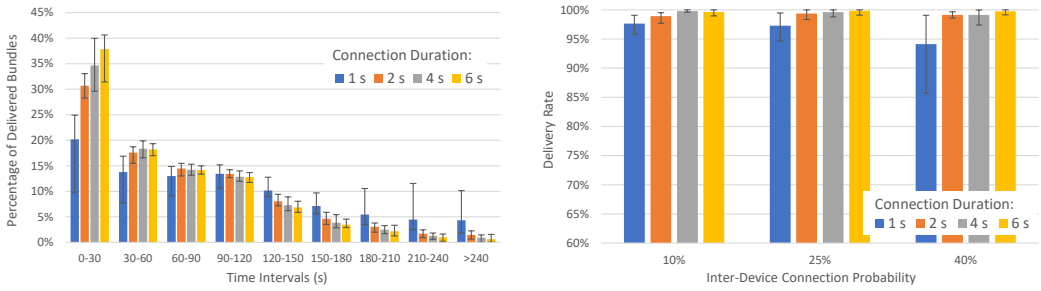
	mean	min	max
Bluetooth via RFCOMM	52.98	23.08	87.52
TCP/IP over WiFi ad-hoc	32.52	12.32	248.74

HTTP Gateway

The HTTP Gateway has been tested on board of a GNSS (Global Navigation Satellite System) device named Topcon Net-G5, which is currently used for on-the-field operations in smart agriculture scenarios. The Net-G5 features a 250 MHz processor and 256 MB of memory. The use case was to enable access to the web-based dashboard of such a device, which provides both state monitoring and configuration. When the communication delay can be tolerated, Æther may allow any device belonging to the same DTN to access such a dashboard. We measured on-device computation delays introduced by the HTTP Gateway while processing the response content for the main page of the configuration dashboard, which features any kind of resource among text, multimedia, and web socket connections, for 20 different HTTP requests. Table 7.5 compares results of the various average computation times on board of the Net-G5, with those obtained by running the HTTP Gateway on a better-equipped desktop with Intel Core i5-2410M @2.30GHz x4 and 8 GB of memory. Results show that the additional computation time introduced by the gateway is compatible with the time needed by the DTN Daemon below to process the bundles, introducing an overhead of ≈30%.

Table 7.5: Server-side computation required to process the content of the Net-G5 dashboard to convey it into the DTN.

	Gateway	DTN Daemon
Net-G5	1.706 (± 0.118) s	5.396 (± 0.789) s
Desktop	0.226 (± 0.004) s	0.574 (± 0.031) s



(a) (b)

Figure 7.10: (a) Distribution over time of delivered bundles, for different connection time intervals. (b) Bundle delivery ratio varying the probability of inter-device connections, comparing different connection time intervals (1, 2, 4 and 6 seconds). All values refer only to bundles already expired at the end of the run.

7.5.2 MQTT Gateway - on Virtualized Environment

Performance of the MQTT Gateway has been tested over a realistic telemetry use case, i.e., when a fleet of end devices exchange generated data through opportunistic connections. We run our prototype of Æther in a virtualized environment, where end devices (sensors) have been implemented within Docker containers. Communication occurs through a virtual switch whose connections are dynamically reconfigured with new OpenFlow rules injected dynamically, thus mimicking vehicle movements over time and their opportunistic connections. The communication occurs over Open vSwitch v2.6.0, while containers are deployed on Docker v17 running on a VM with 16 CPU cores, 12GB of RAM and Linux kernel 4.4.0-96 (the host machine features two octa-core Intel Xeon E5-2660 @ 2.2 GHz CPUs). These experiments have been run using the Epidemic routing algorithm.

We performed runs of 30 minutes each, with 15 devices that perform random connections over time. The duration of each opportunistic connection has been varied among 1, 2, 4, and 6 seconds, while the probability of inter-device connections among 10%, 25%, and 40%. The purpose of varying these parameters was to identify the minimum requirements in terms of inter-device connections that our system needs to work properly. All the devices act as sensor nodes, generating new measurement data every 5 seconds to be delivered to a specific *sink node*, which runs the MQTT broker.

Delivery Rate and Time

Figure 7.10a shows the distribution of bundles successfully delivered to the destination over time, for different connection duration (with a connection probability set to 25%). The plot shows that, if connections last enough, more than one-third of bundles are delivered in less than 30 s, while ≈95% of them reaches the destination within 3 minutes upon their generation. Shorter connections give a slightly smaller percentage of bundles delivered in the earlier intervals; the case with connections of just 1 second features a completely different trend, with visibly slower delivery times. Similar results were obtained for a network featuring a connection probability of 10% (graph not shown).

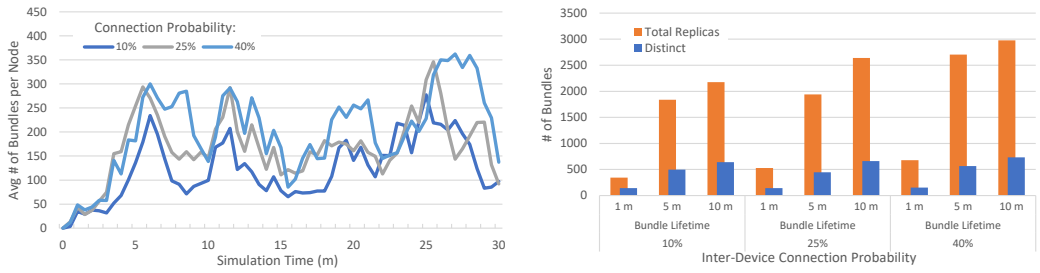
Figure 7.10b shows the percentage of bundles that, at the end of each run, have successfully been delivered to the destination. Taking into account the previous results from Figure 7.10a, The bundle lifetime has been set to 5 minutes. Results show that, when the inter-device connections last only 1 second, a high probability of meeting other peers leads to, counter-intuitively, worse performance. This is because devices do not have enough time to exchange large data, hence more connections just increase the number of duplicated bundles in the network and, thus, queue saturation. In all the other cases, almost all bundles (more than 98%) have been delivered successfully, with no significant improvements when increasing the duration or the probability of inter-devices connections.

In general, results suggest that our approach is suitable in scenarios where inter-devices opportunistic connections of at least 2 seconds can be established with a probability of 10% (or higher) over time, and applications tolerate *(i)* a latency of, at most, 3 minutes to receive the 95% of total data and *(ii)* a data loss up to 2%, which is perfectly reasonable for a telemetry system.

Storage

Since on this set of tests we used an epidemic routing algorithm (namely, a device sends a copy of every carried bundle each time an opportunistic connection is established), we performed some measurements to evaluate the storage capacity required on each device.

Figure 7.11a shows the average number of bundles stored on each node over time, for different connection probabilities and a bundle lifetime set to 5 minutes. If end devices establish connections with a probability of 10%, the number of bundles per device stabilizes between ≈100 and ≈250 after the initial transient. As expected, higher connection probabilities lead to a larger average amount of carried bundles, due to the epidemic strategy. However, in all cases the amount of bundles does not increase indefinitely over time; this behavior means that, on average, the network can deliver bundles at the same rate they are generated, thus keeping the storage occupation bounded.



(a) (b)

Figure 7.11: (c) Average number of bundles stored on each device during the simulation for different probabilities of inter-device connections, with a bundle lifetime of 5 minutes. (d) Average storage overhead varying the probability of inter-device connections and comparing different bundle lifetimes (1, 5 and 10 minutes).

We also compared the total number of bundles in the network over time with the number of distinct ones. Figure 7.11b compares the average storage overhead for different inter-device connection probabilities and different bundle lifetimes (1, 5, and 10 minutes). The graph shows that the ratio between total replicas and distinct bundles increases with the connection probability. On the other hand, we notice a significant difference in the number of bundles (both distinct and replicas) when lifetime is increased from 1 to 5 minutes, while the same does not happen to bring the lifetime to 10 minutes. Indeed, since on average only ≈55% of bundles are delivered in less than 1 minute after their generation (as seen in Figure 7.10a), such a short lifetime makes devices to discard a significant amount of still valid (i.e., not delivered) bundles. This is not the case with a lifetime of 5 and 10 minutes, because, according to the results, almost all packets (≈95%) are delivered in less than 3 minutes.

Results in Figures 7.11 confirm that the Epidemic approach is only suitable when devices with proper capacity are involved and the average size of exchanged messages is small enough, a requirement that is reasonably fulfilled in the telemetry use case here analyzed. In more generic situations, heuristics that reduce the number of replicas are preferred (see results on routing in Section 7.5.4).

7.5.3 Service Discovery - on Virtualized Environment

The above mentioned virtualized environment was used to evaluate also the performance of both Service Discovery and Virtual Service modules. In these tests, we randomly deployed 5 different types of services on top of the available nodes, giving each service a 10% of probability to be present on a given node. We performed multiple experiment instances, varying *(i)* connection probability (between 5% and 70%), *(ii)* number of nodes (between 15 and 50) *(iii)* percentage of nodes that stores service information locally (between 30% and 70%). Results are as follows.

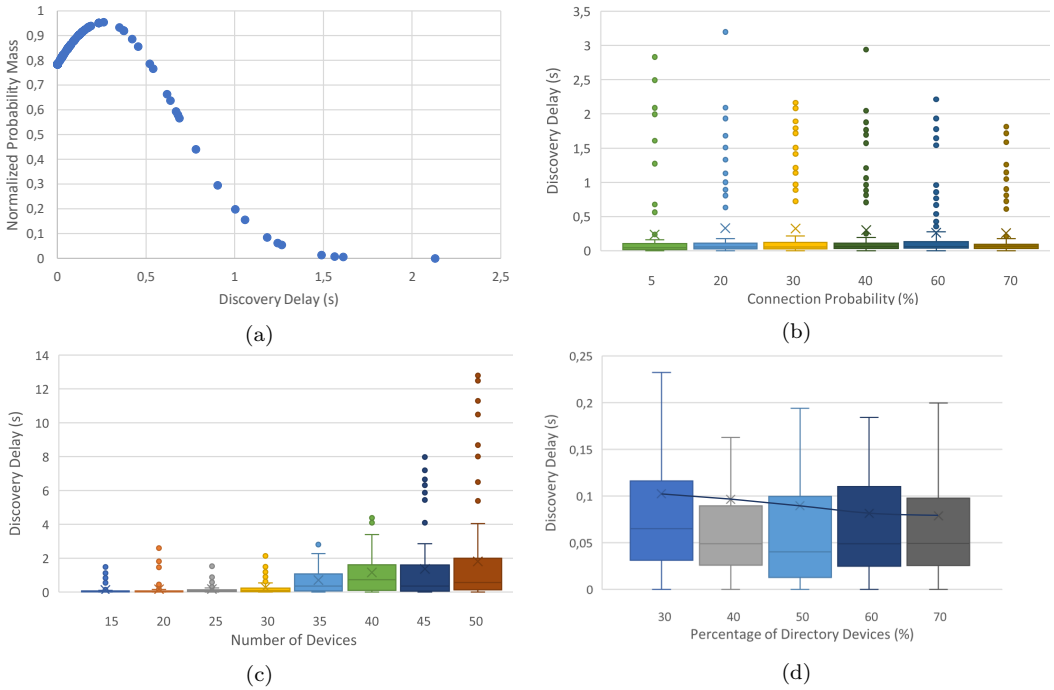
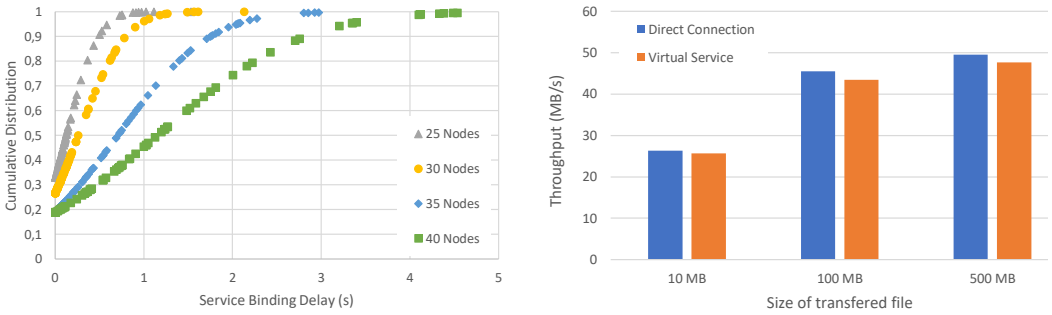


Figure 7.12: Time needed for a discovery request to get a response. (a) Normalized probability mass for the case with 30 nodes, 20% of connection probability and 50% of directories. (bcd) Comparisons in different network setup: (b) 15 nodes and 50% of directories varying connection probability; (b) 20% of connection probability and 50% of directories varying number of nodes; (b) 25 nodes and 20% of connection probability varying percentage of directories.

Discovery Delay

Figure 7.12a shows the normalized probability mass[1] of the delay needed to get a response for a service discovery request that has been propagated toward the network. In particular, it refers to a setup featuring 30 nodes, 20% of connection probability, and 50% of directory nodes (i.e., those that cache information of all known service in the local storage). The normalized mass shows that, given this setup, there is a probability between 70% and 80% to find the desired service in less than 0.5 seconds. Figures 7.12b-d show how varying the scenario setup affects service discovery delay. In particular, from Figure 7.12b it appears that varying the connection probability reduces results sparsity, but has a low impact on the average discovery delay. Figure 7.12c compares the discovery delay when increasing the number of nodes in the network: since the connection probability does not change, the network becomes sparser, with the consequent increase in the average number of traversed hops required by a request to find the desired service. This leads to

[1]In statistics, the probability mass function gives the probability that a discrete random variable is exactly equal to some value [154].



(a) (b)

Figure 7.13: (a) Virtual Service binding delay comparing different number of nodes. (b) Throughput overhead of the virtual service in a connected topology.

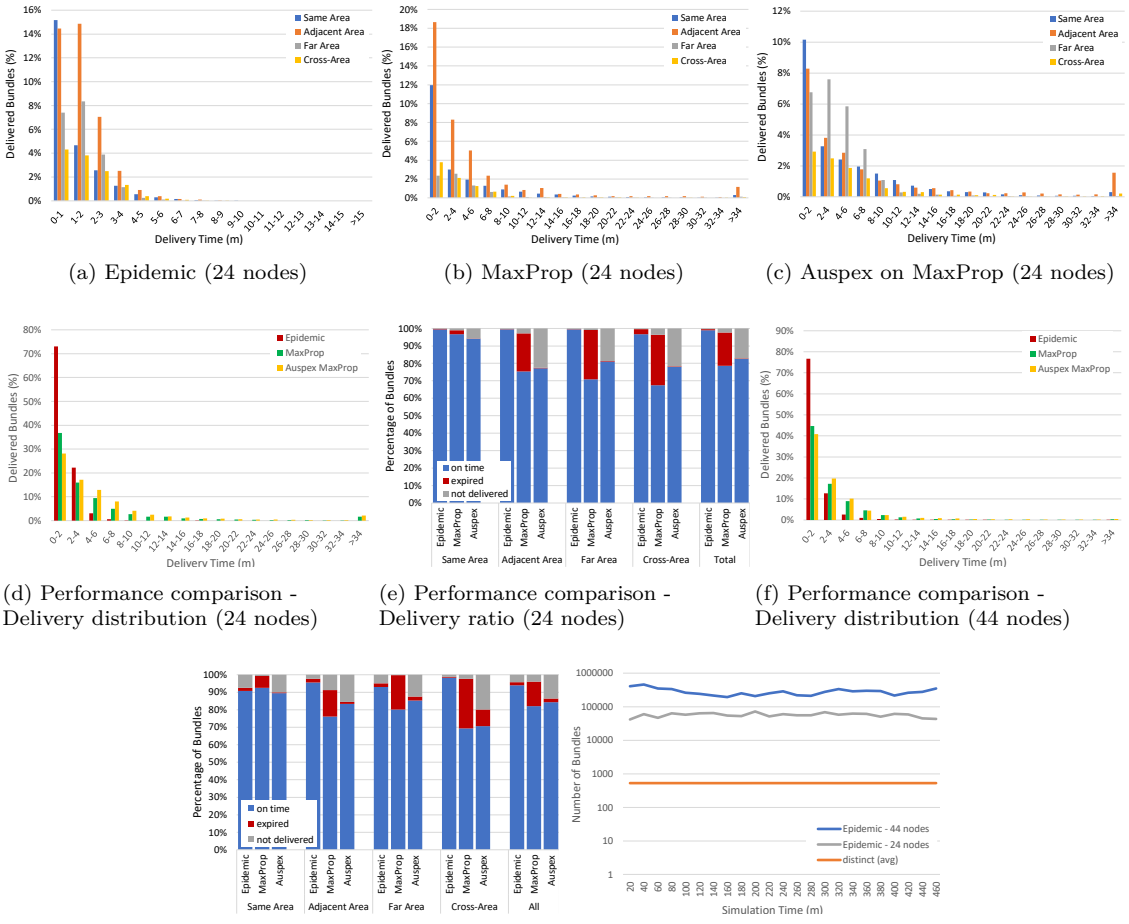
a higher average delay and sparsity of the samples. Finally, Figure 7.12d shows the impact of increasing the percentage of nodes that stores service information locally: the average discovery delay slightly decreases, but the improvements are not noticeable, even compared with the case with 30% of directory nodes.

Virtual Service

We tested the performance of the virtual service module measuring the binding time and throughput. Figure 7.13a shows the cumulative distribution of the time needed since a virtual service request is issued by the application, to the moment in which the binding is completed. This is strictly dependent on the service discovery response time and the network setup. The figure compares the results for different numbers of nodes in the network, showing that the slowest binding times remain in the order of a few seconds even for fleets of 40 nodes. Finally, we measured the overhead introduced by the virtual service module in the baseline throughput, being in between the customer and the actual remote service. These tests have been conducted in the particular case of a connected topology since in a disrupted scenario the processing time is not relevant compared to the network delay. Measurements take into account the transfer of three files of different sizes, showing a negligible overhead (Figure 7.13b).

7.5.4 Auspex Routing - on Emulated Environment

Experiments on the performance of the Auspex routing module have been realized using LEPTON [97], an emulating platform supporting a large number of nodes, each one running the desired software architecture without the necessity of deploying virtual machines or containers. This setup allowed us to simulate actual device movements and reproduce a more sophisticated scenario based on real case



(a) Epidemic (24 nodes)

(b) MaxProp (24 nodes)

(c) Auspex on MaxProp (24 nodes)

(d) Performance comparison - Delivery distribution (24 nodes)

(e) Performance comparison - Delivery ratio (24 nodes)

(f) Performance comparison - Delivery distribution (44 nodes)

(g) Performance comparison - Delivery ratio (44 nodes)

(h) Epidemic bundle overhead

Figure 7.14: Routing experiments comparing Epidemic, MaxProp Vanilla, and Auspex over MaxProp, on two setups with a different number of nodes. Plots show delivery time (a-d, f) and delivery ratio (e, g), distinguishing for destination area, and the overhead of Epidemic (h) on both the setups (scale is logarithmic).

considerations. We compare the performance of our service-oriented routing strategy with the original version of MaxProp [28], which is state-of-the-art in terms of routing algorithms in disrupted networks. Additionally, we also show the performance of the epidemic algorithm. These tests aim to assess the advantages of influencing the original routing decisions with service-related information, such as constraints specifying the tolerated end-to-end delay.

Topology setup

We deployed from 24 to 44 emulated devices on a surface of 50×100 m^2. Each simulation features 4 *stationary devices* located at the edges of the surface. The surface has been divided into six areas (partially overlapped), whose borders can only be crossed by 30% of the total devices (*cross-area* devices). Among these, 33% moves with a speed between 1-2 m/s, the remaining at 0-2 m/s. Instead, half of the *area-bounded* devices moves at 0.5-1 m/s, while the other half at 0-1 m/s. Additionally, roles have been assigned to devices as follows: 40% of the area-bounded devices are sensors, 40% of them are actuators, and the remaining 20% only act as transport devices, without receiving nor generating messages (devices with no application running, e.g., onboard of vehicles with human workers). Cross-area devices can be sensors, actuators, or only transport nodes with a uniform probability. Sensor devices generate most of the messages, sending periodically both to the assigned base station (the closest stationary device) and to at least two other assigned nodes (with arbitrary distance). Additionally, sensors may generate less frequent messages to *(i)* notify an over-threshold measurement to an actuator device or *(ii)* respond to a received request. On the other hand, actuators generate sporadic request messages for randomly selected sensors. Finally, each stationary device may send sporadic messages (e.g., software updates or work plan changes) to the assigned devices. The average number of generated messages per hour is as in Table 7.6. We run multiple simulations of 8 hours each. In each simulation, we test a different routing algorithm, while devices repeat the same movements, following a *Random Waypoint* mobility pattern [83].

Table 7.6: Average number of messages generated in one hour, divided for destination area.

same area	adjacent area	far area	cross-area	total
353	856	170	197	1576
22.4%	54.3%	10.8%	12.5%	/

Latency estimation

To perform these tests we implemented in Auspex a simple mechanism to estimate the latency with peers, which is used to compute the expected latency of a given path influencing the MaxProp decisions accordingly. Whenever a connection with neighbor j is established, in addition to update the MaxProp probabilities as described in Section 7.3.2, a value t_j, estimating the period needed to meet again such neighbor, is updated as follows:

$$t_j = (1 - \alpha)\, t_j + \alpha\, \tau_j,$$

where τ_j is the last measured sample, i.e., the period that was needed to meet again neighbor j last time it disappeared, while α is a parameter empirically set to 0.375. For the aim of these experiments, latency to a neighbor j is simply estimated as $t_j/2$. The whole latency vector is exchanged together with the probability vector at each opportunistic connection so that an estimated latency can be associated with each path, other than the MaxProp cost.

Delivery time

To have an idea of the order of magnitude of the latencies involved in our setup, we first measured the delivery times with the Epidemic algorithm. Figure 7.14a shows the times distribution distinguishing between 4 categories of traffic: *(i)* bundles with destination located within the *same area* of the source device, *(ii)* bundle with destination in an *adjacent area*, *(iii)* bundles with destination in a far area, *(iv)* bundles whose destination is a cross-area device.

Since Epidemic tries, at the same time, all the possible paths between the source and the destination, it is reasonable to consider its delivery time and ratio as near-optimal. However, such an approach is hard to adopt in real scenarios, due to its high overhead on the device storage and transmission capacity. Hence, we use Epidemic as a reference to compare performances of both MaxProp vanilla (Figure 7.14b) that is the state-of-the-art routing algorithm in DTN, and of our service-oriented extension using Auspex (Figure 7.14c).

Table 7.7 shows numerical details on the time needed to deliver 95% of bundles after they are generated, comparing Epidemic and MaxProp and distinguishing for destination areas. We used such values to properly dimension and set the latency constraints of generated messages, i.e., *(i)* the percentage of messages affected by these constraints, and *(ii)* the ranges within which the latency constraint is uniformly selected. This setup is summarized on Table 7.8.

Table 7.7: Time needed to deliver 95% of the bundles.

	same area	adj. area	far area	cross-area
Epidemic	4 m	4 m	4 m	5 m
MaxProp	18 m	24 m	16 m	12 m

Figure 7.14d summarizes delivery time distributions for the three different approaches, without area distinction. The graph shows that Auspex features a more deferred delivery distribution since it intentionally delays bundles with larger latency constraints. As shown in the remainder of this section, this compromise effectively increases the percentage of bundles that are delivered within their latency constraints. A similar trend is observed for the case with 44 nodes (Figure 7.14f).

Table 7.8: Latency constraint imposed on generated messages, based on the destination area. The table shows the percentage of generated bundles that are affected by latency constraints and the ranges within which the constraint values are uniformly selected.

	same area	adj. area	far area	cross-area
Affected	75%	75%	75%	100%
Latency	2-4 m	3-8 m	4-10 m	4-8 m

Delivery ratio

Figure 7.14e compares the percentage of bundles that have been delivered satisfying the service constraints with the three different approaches, distinguishing based on the location of the destination node. Epidemic features the best results, as it sends each message towards all the possible paths. Most bundles having source and destination within the same area are effectively delivered on time by both the heuristics, with a slight degradation (less than 3%) on Auspex. On the other hand, vanilla MaxProp performance significantly decreases when delivering to more distant destinations. Results show that Auspex effectively avoids queues saturation and distributes traffic efficiently among the available paths, improving MaxProp performances of circa 5% (on average) and up to 15% in the case of long-distance communication.

In the simulation with 44 nodes (Figure 7.14g), Epidemic noticeably deteriorates its performances, suggesting that the high number of bundle replicas starts saturating the DTN, affecting transmission capabilities. Noticeably, in this setup Auspex brings the highest improvement on shorter-range communications, with up to 9% bundles delivered on time between adjacent areas, while long-distance communications are only improved by less than 7%. This is due to the trivial algorithm currently adopted in the Auspex prototype to estimate latency between nodes, which deteriorate its accuracy as the number of nodes increases. A more sophisticated estimation technique and fine-tuning would further improve the advantages introduced by Auspex over MaxProp. However, the proposal of an effective latency estimation algorithm between DTN nodes is out of scope for this work, as dedicated works can be found in literature [20, 171, 169].

Finally, Figure 7.14h shows the overhead introduced by the usage of the Epidemic algorithm on the number of generated bundles (scale is logarithmic). The plot highlights the impracticability of using such an algorithm on real scenarios featuring a setup similar to the one reproduced in these simulations. Indeed, compared with the average number of distinct bundles, Epidemic generates replicas up to three orders of magnitude higher in the case of 24 nodes, and even four orders of magnitude higher in the simulation featuring 44 nodes.

7.6 Conclusion

In this chapter, we proposed Æther, a distributed communication system that supports the delivery and consumption of scattered services in a disrupted infrastructure scenario. Core modules of Æther enable the management of the Bundle Protocol through the store-carry-forward paradigm, a Service Discovery approach that takes into account the disrupted nature of the network, and service-oriented routing optimizations. Additionally, Æther provides virtual services that facilitate clients to transparently consume services based on semantic rather than topology considerations. Furthermore, we describe the overall framework that enables applications to operate on top of Æther, providing both APIs to directly exploit features of all the core modules, and protocol gateways that act as bridges between traditional IoT applications and the disrupted network. We deployed our prototype both on physical devices and on emulated environments. Experiments assess the performance of each Æther module, assessing the applicability of this architecture on real scenarios. Notably, results show that the service-oriented approach of Æther improves routing decisions by effectively increase the percentage of messages that are delivered satisfying service level constraints.

Chapter 8

Conclusions

This dissertation addresses the problem of service management and orchestration on new-generation infrastructures, proposing novel solutions that enable providers to benefit from the flexibility and possibilities introduced by new technologies and paradigms. Particular focus is given to the problems of distribution and interoperability in Edge Computing, a paradigm that opens new possibilities for service composition and enables new actors in the service provisioning ecosystem.

Challenges introduced by the new Edge paradigm are manifold. Scarce resources and geographical distribution leads to the need for multiple providers to inter-operate, coordinating service deployment. Each provider should be able to exploit any capability of the heterogeneous infrastructure below to achieve enough flexibility in service deployment. Due to the involvement of multiple providers, coordination in the service deployment could not be demanded to a centralized component. Additionally, different services may benefit from different strategies that cannot be pursued with traditional one-size-fits-all approaches, as application metrics are only known to the service provider of competence. Throughout this work, we investigated novel paradigms for service management and orchestration that could overcome these challenges.

We first focused on a single edge provider, proposing a capability-based orchestration approach that enables exploiting any facility offered by a multi-technological infrastructure (Chapter 2). This increases flexibility and provides better optimization possibilities in service deployment; for instance, in our tests (Section 2.5) we experienced a 21% increased throughput and 45% less end-to-end latency for a service chain delivery when all the capabilities of an SDN-enabled edge network are exploited. However, being resources at edge scarce and geographically distributed, multiple providers, even playing different roles, should be taken into account while designing an edge orchestration model. Therefore, in Chapter 3 we identified the new actors involved in service provisioning and their roles. The chapter also provides an insight into algorithms that providers may adopt in optimizing their mutual interactions. Some of these interactions have been then deepened.

In particular, Chapter 4 proposed a novel Service-Defined Orchestration approach that enables service providers to optimize on service-specific metrics even when deploying applications on third parties platforms. This new paradigm has been evaluated over some representative edge use cases, showing, for instance, how a CDN provider may improve his miss rate distribution in a situation of high user mobility, from a 20%-40% range to sub-10% values, compared to using a conventional one-size-fits-all orchestrator (Section 4.5). We then addressed the problem of coordinating multiple providers that orchestrate resources on a shared set of edge clusters. With this respect, we proposed DRAGON, a Distributed Resource AssiGnment and OrchestratioN algorithm that enables agreement on how to temporarily partition resources among a set of actors, with guarantees on convergence time and and a (1-1/e) optimal performance bound. DRAGON has been described in Chapter 5. Large scale experiments have shown that the algorithm may operate coordinating 300 concurrent applications over 400 nodes achieving convergence times in the order of 1.5 seconds (Section 5.6).

The last part of this thesis explores the interactions between services for workflow composition in the context of the highly modular and distributed edge infrastructure. In particular, Chapter 6 focuses on the challenges of service providers in exploiting the plethora of new facilities populating the infrastructure while composing their final service. We design a model-based configuration layer that enables interoperability between existing actors and facilitates service composition through monitoring and tuning of arbitrary components. The configuration layer introduces a tolerable latency of ≈2 ms for both reading and writing a configuration three with 6 levels of depth (Section 6.6). Chapter 7 ultimately overviewed the applicability of the new generation service facilities on highly scattered infrastructures. With a focus on Industrial Internet of Things scenarios, we proposed Æther, a service-oriented communication system that provides service management features and service-oriented routing optimization, thus enabling IoT services to operate on a disrupted infrastructure. As we shown in Section 7.5, the service discovery protocol proposed within Æther successfully binds ≈80% of the services in less than 1 second in scenarios with 30 sparse devices (featuring 20% of mutual-connection probability). Additionally, the service-oriented routing optimization effectively improves the state-of-the-art MaxProp algorithm in terms of on-time delivery ratio, of ≈5% (on average) and up to ≈15% in the case of long distance communication.

This thesis covered multiple aspects of the problem of service management and orchestration, providing novel solutions that may introduce benefits to the actor involved in the service provisioning ecosystem. However, several challenges still remain open, as we deepened only a subset of the existing interactions between the involved providers. In this regard, the interaction between multiple infrastructure providers deserves a particular mention. In the Edge Computing scenario, where services are tied to the physical location of the resources, infrastructure providers may encounter several opportunities in collaborating, such as being able to offer

a larger infrastructure that may include also (part of) the resources available in partnering IaaS domains. In this scenario, infrastructure providers may want to introduce further optimization strategies in service deployment that take into account agreements stipulated with peers, other than service-level constraints. As in this thesis we mainly focused on the problem from a service-oriented point of view, this problem may represent a promising topic for future investigations.

Appendix A

Author publications

Part of the research presented in this dissertation has been previously published in the following papers:

- Bonafiglia, R., Castellano, G., Cerrato, I., & Risso, F., End-to-end service orchestration across SDN and cloud computing domains. In International Conference on Network Softwarization (NetSoft), 2017 IEEE Conference on (pp. 1-6). IEEE.
- Castellano, G., Cerrato, I., Risso, F., Pezzolla, D., & Manzalini, A., Mimicking a compute domain orchestrator with the ONOS SDN controller. In International Conference on Network Softwarization (NetSoft), 2017 IEEE Conference on (pp. 1-3). IEEE.
- Castellano, G., Risso, F., & Loti, R., Fog Computing over Challenged Networks: a Real Case Evaluation. In International Conference on Cloud Networking (Cloudnet), 2018 IEEE Conference on (pp. 1-7). IEEE.
- Castellano, G., Risso, F., Enabling Fog Computing over Delay/Disruption-Tolerant Networks. In 4th Italian Conference on ICT for Smart Cities And Communities (ICities), 2018, (pp. 1-2).
- Castellano, G., Cerrato, I., Gharbaoui, M., Fichera, S., Martini, B., Risso, F., & Castoldi, P., A model-based abstraction layer for heterogeneous SDN applications. International Journal of Communication Systems 32.17 (2019), e3989.
- Castellano, G., Esposito, F., & Risso, F., A Distributed Orchestration Algorithm for Edge Computing Resources with Guarantees. In International Conference on Computer Communications (INFOCOM), 2019 IEEE Conference on (pp. 2548-2556). IEEE.

- Castellano, G., Esposito, F., & Risso, F., A Service-Defined Approach for Orchestration of Heterogeneous Applications in Cloud/Edge Platforms. IEEE Transactions on Network and Service Management 16.4 (2019), pp. 1404-1418.
- Castellano, G., Manzalini, A., & Risso, F., A Disaggregated MEC Architecture Enabling Open Services and Novel Business Models. In International Conference on Network Softwarization (NetSoft), 2019 IEEE Conference on (pp. 178-182). IEEE.

Bibliography

[1] Netgroup @polito. *Domain Information Library*. URL: https://github.com/netgroup-polito/domain-information-library (visited on 02/21/2017).

[2] Netgroup @polito. *ONOS applications*. URL: https://github.com/netgroup-polito/onos-applications (visited on 02/21/2017).

[3] Netgroup @polito. *OpenStack domain orchestrator*. URL: https://github.com/netgroup-polito/frog4-openstack-do (visited on 03/28/2020).

[4] Netgroup @polito. *SDN Application ToY Agent repository*. URL: https://github.com/netgroup-polito/sdn-app-toy-agent (visited on 04/10/2020).

[5] Netgroup @polito. *SDN domain orchestrator*. URL: https://github.com/netgroup-polito/frog4-sdn-do (visited on 02/21/2017).

[6] *Akraino Edge Stack*. URL: https://www.akraino.org (visited on 01/07/2019).

[7] Edoardo Amaldi et al. "On the computational complexity of the virtual network embedding problem". In: *Electronic Notes in Discrete Mathematics* 52 (2016), pp. 213–220.

[8] Strategy Analytics. *Global Connected and IoT Device Forecast Update*. 2019.

[9] Giuseppe Araniti et al. "Contact graph routing in DTN space networks: overview, enhancements and performance". In: *IEEE Communications Magazine* 53.3 (2015), pp. 38–46.

[10] Vladimir Atanasovski and Liljana Gavrilovska. "Efficient service discovery schemes in wireless ad hoc networks implementing cross-layer system design". In: *27th International Conference on Information Technology Interfaces, 2005.* IEEE. 2005, pp. 496–501.

[11] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[12] Maël Auzias, Yves Mahéo, and Frédéric Raimbault. "CoAP over BP for a delay-tolerant internet of things". In: *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE. 2015, pp. 118–123.

[13] Tayebeh Bahreini and Daniel Grosu. "Efficient placement of multi-component applications in edge computing systems". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM. 2017, p. 5.

[14] Debasis Bandyopadhyay and Jaydip Sen. "Internet of things: Applications and challenges in technology and standardization". In: *Wireless Personal Communications* 58.1 (2011), pp. 49–69.

[15] Paul Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177.

[16] Pankaj Berde et al. "ONOS: towards an open, distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 1–6.

[17] Maria Bermudez-Edo et al. "IoT-Lite: a lightweight semantic model for the Internet of Things". In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE. 2016, pp. 90–97.

[18] Carlos J Bernardos et al. "5GEx: realising a Europe-wide multi-domain framework for software-defined infrastructures". In: *Transactions on Emerging Tele-communications Technologies* 27.9 (2016), pp. 1271–1280.

[19] Dimitri P Bertsekas and Athena Scientific. *Convex optimization algorithms*. Athena Scientific Belmont, 2015.

[20] Nikolaos Bezirgiannidis, Scott Burleigh, and Vassilis Tsaoussidis. "Delivery time estimation for space bundles". In: *IEEE Transactions on Aerospace and Electronic Systems* 49.3 (2013), pp. 1897–1910.

[21] Giuseppe Bianchi et al. "OpenState: programming platform-independent stateful openflow applications inside the switch". In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 44–51.

[22] A. Bierman, M. Bjorklund, and K. Watsen. *RESTCONF Protocol*. RFC 8040. RFC Editor, Jan. 2017. URL: https://tools.ietf.org/html/rfc8040.

[23] Martin Bjorklund et al. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. Tech. rep.

[24] Roberto Bonafiglia. "Improving the Performance of Virtualized Network Services Based on NFV and SDN". PhD thesis. Politecnico di Torino, 2018.

[25] Roberto Bonafiglia et al. "Enabling NFV services on resource-constrained CPEs". In: *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*. IEEE. 2016, pp. 83–88.

[26] Roberto Bonafiglia et al. "End-to-end service orchestration across SDN and cloud computing domains". In: *2017 IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–6.

[27] Pat Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[28] John Burgess et al. "MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks." In: *2006 IEEE International Conference on Computer Communications (INFOCOM)*. Vol. 6. Barcelona, Spain. 2006.

[29] Celeste Campo et al. "PDP and GSDL: a new service discovery middleware to support spontaneous interactions in pervasive systems". In: *Third IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE. 2005, pp. 178–182.

[30] Yue Cao and Zhili Sun. "Routing in delay/disruption tolerant networks: A taxonomy, survey and challenges". In: *IEEE Communications surveys & tutorials* 15.2 (2012), pp. 654–677.

[31] Giuseppe Antonio Carella and Thomas Magedanz. "Open baton: a framework for virtual network function management and orchestration for emerging software-based 5g networks". In: *Newsletter* 2016 (2015).

[32] Enrico Carniani et al. "Usage control on cloud systems". In: *Future Generation Computer Systems* 63 (2016), pp. 37–55.

[33] Gabriele Castellano. *DRAGON Prototype*. 2018. URL: https://github.com/gabrielecastellano/dragon (visited on 04/10/2020).

[34] Gabriele Castellano. *SDO Use Cases*. URL: https://github.com/netgroup-polito/dragon/tree/master/use_cases_simulation.

[35] Gabriele Castellano, Flavio Esposito, and Fulvio Risso. "A distributed orchestration algorithm for edge computing resources with guarantees". In: *2019 IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2019, pp. 2548–2556.

[36] Gabriele Castellano, Flavio Esposito, and Fulvio Risso. "A Service-Defined Approach for Orchestration of Heterogeneous Applications in Cloud/Edge Platforms". In: *IEEE Transactions on Network and Service Management* 16.4 (2019), pp. 1404–1418.

[37] Gabriele Castellano, Antonio Manzalini, and Fulvio Risso. "A Disaggregated MEC Architecture Enabling Open Services and Novel Business Models". In: *2019 IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 178–182.

[38] Gabriele Castellano and Fulvio Risso. "Enabling Fog Computing over Delay/Disruption-Tolerant Networks". In: *4th Italian Conference on ICT for Smart Cities And Communities (ICities)*. 2018, pp. 1–2.

[39] Gabriele Castellano, Fulvio Risso, and Riccardo Loti. "Fog Computing over Challenged Networks: a Real Case Evaluation". In: *2018 IEEE International Conference on Cloud Networking (CloudNet)*. IEEE. 2018, pp. 1–7.

[40] Gabriele Castellano et al. "A model-based abstraction layer for heterogeneous SDN applications". In: *International Journal of Communication Systems* 32.17 (2019), e3989.

[41] Gabriele Castellano et al. "Mimicking a compute domain orchestrator with the ONOS SDN controller". In: *2017 IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–3.

[42] Ivano Cerrato et al. "Toward dynamic virtualized network services in telecom operator networks". In: *Computer Networks* 92 (2015), pp. 380–395.

[43] Han-Lim Choi, Luc Brunet, and Jonathan P How. "Consensus-based decentralized auctions for robust task allocation". In: *IEEE transactions on robotics* 25.4 (2009), pp. 912–926.

[44] *Cisco Network Services Orchestrator (NSO) Solutions*. URL: https://www.cisco.com/c/en/us/solutions/service-provider/solutions-cloud-providers/network-services-orchestrator-solutions.html (visited on 04/15/2018).

[45] A Clemm et al. "Subscribing to YANG datastore push updates". In: *draft-ietf-netconf-yang-push-05* 28 (2017).

[46] András Császár et al. "Unifying cloud and carrier network: Eu fp7 project unify". In: *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE. 2013, pp. 452–457.

[47] Li Da Xu, Wu He, and Shancang Li. "Internet of things in industries: A survey". In: *IEEE Transactions on industrial informatics* 10.4 (2014), pp. 2233–2243.

[48] George Darzanos, Manos Dramitinos, and George D Stamoulis. "Coordination Models for 5G Multi-Provider Service Orchestration: Specification and Assessment". In: *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer. 2017, pp. 262–274.

[49] Michael Demmer and Kevin Fall. "DTLSR: delay tolerant routing for developing regions". In: *Proceedings of the 2007 workshop on Networked systems for developing regions*. 2007, pp. 1–6.

[50] Namiot Dmitry and Sneps-Sneppe Manfred. "On micro-services architecture". In: *International Journal of Open Information Technologies* 2.9 (2014).

[51] Sevil Dräxler, Holger Karl, and Zoltán Ádám Mann. "Jasper: Joint optimization of scaling, placement, and routing of virtual network services". In: *IEEE Transactions on Network and Service Management* 15.3 (2018), pp. 946–960.

[52] Jianbo Du et al. "Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee". In: *IEEE Transactions on Communications* 66.4 (2018), pp. 1594–1608.

[53] *EdgeX Foundry Architectural Tenets*. URL: https://docs.edgexfoundry.org (visited on 04/29/2020).

[54] Daniel Ellard and Dan Brown. "DTN IP Neighbor Discovery (IPND)". In: *IETF Draft* (2010).

[55] PE Engelstad and Yan Zheng. "Evaluation of service discovery architectures for mobile ad hoc networks". In: *Second Annual Conference on Wireless On-demand Network Systems and Services*. IEEE. 2005, pp. 2–15.

[56] Flavio Esposito et al. "Slice embedding solutions for distributed service architectures". In: *ACM Computing Surveys (CSUR)* 46 (2013), p. 6.

[57] NFVISG ETSI. "Network function virtualization; management and orchestration". In: *White Paper* 1 (2014).

[58] Jeroen Famaey et al. "A hierarchical approach to autonomic network management". In: *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*. IEEE. 2010, pp. 225–232.

[59] Emanuele Fia. "Definizione e prototipazione di servizi auto-orchestranti su infrastrutture 5G". MA thesis. Politecnico di Torino, 2017.

[60] Emanuele Fia. *SDO Compiler Prototype*. URL: https://github.com/netgroup-polito/sdo-compiler.

[61] Emanuele Fia. *SDO Prototype*. 2018. URL: https://github.com/netgroup-polito/sdo-module (visited on 04/29/2020).

[62] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004. ISBN: 9781932394184.

[63] Zhen-guo Gao et al. "RICFFP: an efficient service discovery protocol for MANETs". In: *International Conference on Embedded and Ubiquitous Computing*. Springer. 2004, pp. 786–795.

[64] J Antonio Garcia-Macias and Dante Arias Torres. "Service discovery in mobile ad-hoc networks: better at the network layer?" In: *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE. 2005, pp. 452–457.

[65] Aaron Gember-Jacobson et al. "OpenNF: Enabling innovation in network function control". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 163–174.

[66] Molka Gharbaoui et al. "Network orchestrator for QoS-enabled service function chaining in reliable NFV/SDN infrastructure". In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–5.

[67] M Gharbaoui et al. "Experimenting latency-aware and reliable service chaining in Next Generation Internet testbed facility". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–4.

[68] *GSMA*. URL: http://gsma.com (visited on 01/07/2019).

[69] Riccardo Guerzoni et al. "Analysis of end-to-end multi-domain management and orchestration frameworks for software defined infrastructures: an architectural survey". In: *Transactions on Emerging Telecommunications Technologies* 28.4 (2017), e3103.

[70] Joel Halpern, Carlos Pignataro, et al. *Service function chaining (SFC) architecture*. RFC 7665. RFC Editor, Oct. 2015. URL: https://tools.ietf.org/html/rfc7665.

[71] Wonkyu Han et al. "State-aware network access management for software-defined networks". In: *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*. 2016, pp. 1–11.

[72] Hossam Hassanein, Yu Yang, and Afzal Mawji. "A new approach to service discovery in wireless mobile ad hoc networks". In: *International Journal of Sensor Networks* 2.1-2 (2007), pp. 135–145.

[73] Sumi Helal et al. "Konark-a service discovery and delivery protocol for ad-hoc networks". In: *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*. Vol. 3. IEEE. 2003, pp. 2107–2113.

[74] Juliver Gil Herrera and Juan Felipe Botero. "Resource allocation in NFV: A comprehensive survey". In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 518–532.

[75] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[76] Thomas T Hildebrandt and Raghava Rao Mukkamala. "Declarative event-based workflow as distributed dynamic condition response graphs". In: *arXiv preprint arXiv:1110.4161* (2011).

[77] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[78] *Instagram by the Numbers: Stats, Demographics and Fun Facts*. URL: https://www.omnicoreagency.com/instagram-statistics/ (visited on 06/15/2020).

[79] China Mobile Research Institute. *C-RAN: The Road Towards Green RAN*. Oct. 2011.

[80] *Information technology — UPnP Device Architecture*. Standard. 2008.

[81] Sushant Jain, Kevin Fall, and Rabin Patra. "Routing in a delay tolerant network". In: *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. 2004, pp. 145–158.

[82] Michael Jarschel et al. "An evaluation of QoE in cloud gaming based on subjective tests". In: *Fifth conference on Innovative mobile and internet services in ubiquitous computing (imis)*. IEEE. 2011, pp. 330–335.

[83] David B Johnson and David A Maltz. "Dynamic source routing in ad hoc wireless networks". In: *Mobile computing*. Springer, 1996, pp. 153–181.

[84] *JOLNet: a geographical SDN network testbed*. Accessed on: 2017-02-06. URL: https://www.softfire.eu/jolnet.

[85] David Karger et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[86] Sami Kekki et al. *MEC in 5G networks*. Tech. rep. June 2018.

[87] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. "The ONE simulator for DTN protocol evaluation". In: *Proceedings of the 2nd international conference on simulation tools and techniques*. 2009, pp. 1–10.

[88] Samir Khuller, Anna Moss, and Joseph Seffi Naor. "The budgeted maximum coverage problem". In: *Information processing letters* 70.1 (1999), pp. 39–45.

[89] Michael Klein and Birgitta König-Ries. "Multi-layer clusters in ad-hoc networks—an approach to service discovery". In: *International Conference on Research in Networking*. Springer. 2002, pp. 187–201.

[90] Michael Klein, Birgitta Konig-Ries, and Philipp Obreiter. "Service rings-a semantic overlay for service discovery in ad hoc networks". In: *14th International Workshop on Database and Expert Systems Applications, 2003. Proceedings.* IEEE. 2003, pp. 180–185.

[91] Kireeti Kompella and Yakov Rekhter. *Virtual private LAN service (VPLS) using BGP for auto-discovery and signaling.* Tech. rep. 2007.

[92] Hadi Razzashi Kouchaksaraei et al. "Programmable and Flexible Management and Orchestration of Virtualized Network Functions". In: *2018 European Conference on Networks and Communications.* IEEE. 2018, pp. 1–9.

[93] Ulaş C Kozat and Leandros Tassiulas. "Service discovery in mobile ad hoc networks: an overall perspective on architectural choices and network layer support issues". In: *Ad Hoc Networks* 2.1 (2004), pp. 23–44.

[94] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[95] Choonhwa Lee, Sumi Helal, and Wonjun Lee. "Gossip-based service discovery in mobile ad hoc networks". In: *IEICE transactions on communications* 89.9 (2006), pp. 2621–2624.

[96] Aris Leivadeas et al. "A graph partitioning game theoretical approach for the VNF service chaining problem". In: *IEEE Transactions on Network and Service Management* 14.4 (2017), pp. 890–903.

[97] *Lepton.* Accessed: 2020-03-11. URL: http://www-casa.irisa.fr/lepton.

[98] Anders Lindgren, Avri Doria, and Olov Schelén. "Probabilistic routing in intermittently connected networks". In: *ACM SIGMOBILE mobile computing and communications review* 7.3 (2003), pp. 19–20.

[99] Marin Litoiu et al. "A business driven cloud optimization architecture". In: *Proceedings of the 2010 ACM Symposium on Applied Computing.* ACM. 2010, pp. 380–385.

[100] DR Lopez. "OpenMANO: The dataplane ready open source NFV MANO stack". In: *IETF Meeting Proceedings, Dallas, Texas, USA.* 2015.

[101] Francesco Lucrezia et al. "Introducing network-aware scheduling capabilities in openstack". In: *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft).* IEEE. 2015, pp. 1–5.

[102] Jorge E Luzuriaga et al. "A disruption tolerant architecture based on MQTT for IoT applications". In: *Consumer Communications & Networking Conference (CCNC), 2017 14th IEEE Annual.* IEEE. 2017, pp. 71–76.

[103] Nancy A Lynch. *Distributed algorithms.* Elsevier, 1996.

[104] Pavel Mach and Zdenek Becvar. "Mobile edge computing: A survey on architecture and computation offloading". In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1628–1656.

[105] Zachary MacHardy et al. "V2X access technologies: Regulation, research, and remaining challenges". In: *IEEE Communications Surveys & Tutorials* 20.3 (2018), pp. 1858–1877.

[106] Jacques Malenfant, Marco Jacques, and Franois Nicolas Demers. "A tutorial on behavioral reflection and its implementation". In: *Proceedings of the Reflection*. Vol. 96. 1996, pp. 1–20.

[107] Antonio Manzalini et al. "A unifying operating platform for 5G end-to-end and multi-layer orchestration". In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–5.

[108] Jan Medved et al. "Opendaylight: Towards a model-driven sdn controller architecture". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. IEEE. 2014, pp. 1–6.

[109] Sara Mehar et al. "An optimized roadside units (rsu) placement for delay-sensitive applications in vehicular networks". In: *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*. IEEE. 2015, pp. 121–127.

[110] Daniele Miorandi et al. "Internet of things: Vision, applications and research challenges". In: *Ad hoc networks* 10.7 (2012), pp. 1497–1516.

[111] *MobileEdgeX*. URL: mobiledgex.com (visited on 01/07/2019).

[112] AA Mohammed et al. "SDN controller for network-aware adaptive orchestration in dynamic service chaining". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE. 2016, pp. 126–130.

[113] Martin Moser. "Declarative scheduling for optimally graceful QoS degradation". In: *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*. IEEE. 1996, pp. 86–94.

[114] Shinji Motegi, Kiyohito Yoshihara, and Hiroki Horiuchi. "Service discovery for wireless ad hoc networks". In: *The 5th International Symposium on Wireless Personal Multimedia Communications*. Vol. 1. IEEE. 2002, pp. 232–236.

[115] Ryo Nakamura et al. "Flowfall: A service chaining architecture with commodity technologies". In: *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*. IEEE. 2015, pp. 425–431.

[116] Pedro Neves et al. "The SELFNET approach for autonomic management in an NFV/SDN networking paradigm". In: *International Journal of Distributed Sensor Networks* 12.2 (2016), p. 2897479.

[117] Michael Nidd. "Service discovery in DEAPspace". In: *IEEE personal communications* 8.4 (2001), pp. 39–45.

[118] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 305–319.

[119] Diego Ongaro and John K Ousterhout. "In search of an understandable consensus algorithm." In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[120] *Open Edge Ccomputing*. URL: http://openedgecomputing.org (visited on 01/07/2019).

[121] *OpenConfig*. URL: http://www.openconfig.net (visited on 03/28/2020).

[122] *OpenFog Consortium*. URL: http://www.openfogconsortium.org (visited on 01/07/2019).

[123] Matteo Ottolini. "Fornitura di connettività ottimizzata in un'infrastruttura DTN service-oriented". MA thesis. Politecnico di Torino, 2020.

[124] Federica Paganelli, Mehmet Ulema, and Barbara Martini. "Context-aware service composition and delivery in NGSONs over SDN". In: *IEEE Communications Magazine* 52.8 (2014), pp. 97–105.

[125] Larry Peterson et al. "Xos: An extensible cloud operating system". In: *Proceedings of the 2nd International Workshop on Software-Defined Ecosystems*. 2015, pp. 23–30.

[126] Ben Pfaff et al. "Extending networking into the virtualization layer." In: *Hotnets*. 2009.

[127] Pham Tran Anh Quang et al. "Single and multi-domain adaptive allocation algorithms for vnf forwarding graph embedding". In: *IEEE Transactions on Network and Service Management* 16.1 (2018), pp. 98–112.

[128] Jrgen Quittek et al. "Network functions virtualisation (NFV)-management and orchestration". In: *ETSI NFV ISG, White Paper* (2014).

[129] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. "Pico replication: A high availability framework for middleboxes". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–15.

[130] Shriram Rajagopalan et al. "Split/merge: System support for elastic execution in virtual middleboxes". In: *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 227–240.

[131] Shermila Ranadheera, Setareh Maghsudi, and Ekram Hossain. "Computation offloading and activation of mobile edge computing servers: A minority game". In: *IEEE Wireless Communications Letters* (2018).

[132] Hajo A Reijers, Tijs Slaats, and Christian Stahl. "Declarative modeling–An academic dream or the future for BPM?" In: *Business Process Management*. Springer, 2013, pp. 307–322.

[133] Alex Reznik et al. "Developing software for multi-access edge computing". In: *ETSI White Paper* 20 (2017).

[134] Robert Ricci et al. "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications". In: *; login:: the magazine of USENIX & SAGE* 39.6 (2014), pp. 36–38.

[135] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. "A taxonomy and survey of cloud computing systems". In: *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*. Ieee. 2009, pp. 44–51.

[136] Ermin Sakic and Wolfgang Kellerer. "Response time and availability study of RAFT consensus in distributed SDN control plane". In: *IEEE Transactions on Network and Service Management* 15.1 (2017), pp. 304–318.

[137] Bessem Sayadi and Laurent Roullet. *5G: Platform and Not Protocol*. Jan. 2018. URL: https://sdn.ieee.org/newsletter/january-2018/5g-platform-and-not-protocol.

[138] Gregor Schiele, Christian Becker, and Kurt Rothermel. "Energy-efficient cluster-based service discovery for ubiquitous computing". In: *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. 2004, 14–es.

[139] Sebastian Schildt et al. "IBR-DTN: A lightweight, modular and highly portable Bundle Protocol implementation". In: *Electronic Communications of the EASST* 37 (2011).

[140] Vincenzo Sciancalepore et al. "A double-tier MEC-NFV architecture: Design and optimisation". In: *Standards for Communications and Networking (CSCN), 2016 IEEE Conference on*. IEEE. 2016, pp. 1–6.

[141] Karim Seada and Ahmed Helmy. "Rendezvous regions: A scalable architecture for service location and data-centric storage in large-scale wireless networks". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE. 2004, p. 218.

[142] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[143] Ali Al-Shabibi and L Peterson. "Cord: Central office re-architected as a datacenter". In: *OpenStack Summit* (2015), pp. 1–38.

[144] Lloyd S Shapley. "A value for n-person games". In: *The Shapley value* (1988), pp. 31–40.

[145] Siva Sivavakeesar, Oscar F Gonzalez, and George Pavlou. "Service discovery strategies in ubiquitous communication environments". In: *IEEE Communications Magazine* 44.9 (2006), pp. 106–113.

[146] João Soares et al. "Toward a telco cloud environment for service functions". In: *IEEE Communications Magazine* 53.2 (2015), pp. 98–106.

[147] Thomas Soenen et al. "A model to select the right infrastructure abstraction for service function chaining". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2016, pp. 233–239.

[148] Balázs Sonkoly et al. "Multi-domain service orchestration over networks and clouds: A unified approach". In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 377–378.

[149] Nathan F Saraiva de Sousa et al. "Network service orchestration: A survey". In: *Computer Communications* (2019).

[150] Bart Spinnewyn et al. "Coordinated service composition and embedding of 5g location-constrained network functions". In: *IEEE Transactions on Network and Service Management* 15.4 (2018), pp. 1488–1502.

[151] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. "Efficient routing in intermittently connected mobile networks: The multiple-copy case". In: *IEEE/ACM transactions on networking* 16.1 (2008), pp. 77–90.

[152] Margaret van Steenderen. "Universal description, discovery and integration". In: *SA Journal of Information Management* 2.4 (2000).

[153] Daniel H Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide: The Definitive Guide*. " O'Reilly Media, Inc.", 2005.

[154] William J Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton university press, 2009.

[155] Harald Sundmaeker et al. "Vision and challenges for realising the Internet of Things". In: *Cluster of European Research Projects on the Internet of Things, European Commision* 3.3 (2010), pp. 34–36.

[156] Maxim Sviridenko. "A note on maximizing a submodular set function subject to a knapsack constraint". In: *Operations Research Letters* 32.1 (2004), pp. 41–43.

[157] Tarik Taleb, Adlen Ksentini, and Pantelis Frangoudis. "Follow-me cloud: When cloud services follow mobile users". In: *IEEE Transactions on Cloud Computing* (2016).

[158] Tarik Taleb et al. "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration". In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1657–1681.

[159] *Telecom Infra Project*. 2016. URL: http://telecominfraproject.com (visited on 01/05/2019).

[160] *Tinyproxy*. Accessed: 2020-02-04. URL: https://tinyproxy.github.io.

[161] Gina C Tjhai et al. "Investigating the problem of IDS false alarms: An experimental study using Snort". In: *IFIP International Information Security Conference*. Springer. 2008, pp. 253–267.

[162] Amin Vahdat, David Becker, et al. *Epidemic routing for partially connected ad hoc networks*. 2000.

[163] Ishan Vaishnavi, Riccardo Guerzoni, and Riccardo Trivisonno. "Recursive, hierarchical embedding of virtual infrastructure in multi-domain substrates". In: *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2015, pp. 1–9.

[164] Christopher N Ververidis and George C Polyzos. "Service discovery for mobile ad hoc networks: a survey of issues and techniques". In: *IEEE Communications Surveys & Tutorials* 10.3 (2008).

[165] Nguyen-Son Vo et al. "Optimal video streaming in dense 5g networks with d2d communications". In: *IEEE Access* 6 (2018), pp. 209–223.

[166] Heinrich Von Stackelberg. *Market structure and equilibrium*. Springer Science & Business Media, 2010.

[167] Chih-Chiang Wang et al. "Toward Optimal Resource Allocation of Virtualized Network Functions for Hierarchical Datacenters". In: *IEEE Transactions on Network and Service Management* 15.4 (2018), pp. 1532–1544.

[168] *Wowza streaming engine*. URL: https://www.wowza.com (visited on 04/29/2020).

[169] Hongcheng Yan, Qingjun Zhang, and Yong Sun. "Local information-based congestion control scheme for space delay/disruption tolerant networks". In: *Wireless Networks* 21.6 (2015), pp. 2087–2099.

[170] *YouTube for Press*. URL: https://www.youtube.com/about/press/ (visited on 06/15/2020).

[171] Qian Yu et al. "Modeling RTT for DTN protocol over asymmetric cislunar space channels". In: *IEEE Systems Journal* 10.2 (2014), pp. 556–567.

[172] Faheem Zafari et al. "A Game-Theoretic Approach to Multi-Objective Resource Sharing and Allocation in Mobile Edge Clouds". In: *arXiv preprint arXiv:1808.06937* (2018).

[173] Ying Zhang et al. “Steering: A software-defined networking for inline service chaining”. In: *2013 21st IEEE international conference on network protocols (ICNP)*. IEEE. 2013, pp. 1–10.

[174] Yuan Zhang et al. “To offload or not to offload: An efficient code partition algorithm for mobile cloud computing”. In: *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*. IEEE. 2012, pp. 80–86.

[175] Zhensheng Zhang. “Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: overview and challenges”. In: *IEEE Communications Surveys & Tutorials* 8.1 (2006), pp. 24–37.