Università degli Studi di Messina

Department of Engineering

doctoral programme in
"cyber physical systems"
xxxiv cycle
ing-inf/05

A service-oriented architecture for IoT infrastructure and Fog-minded DevOps

Doctoral Thesis by:

Zakaria Benomar

Supervisor:

Prof. Antonio Puliafito

Co-Supervisor:

Prof. Giovanni Merlino

The Chair of Doctoral Program:

Prof. Antonio Puliafito

Academic Year 2020 - 2021

Declaration of Authorship

A thesis submitted in fulfillment of the requirements for the Doctor of Philosophy

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Zakaria Benomar

Messina, 23 November 2021

Acknowledgment

Academia

A special thanks to Prof. *Antonio Puliafito* as a great mentor and supervisor. His advice, guidance, patience, and insight throughout this project was undeniable. Besides, I would like to thank him also for providing me the wonderful opportunity in doing research at the University of Messina. I wish to express my deepest gratitude to Prof. *Giovanni Merlino* and Prof. *Francesco Longo* that have dedicated their time and effort in providing countless assistance.

Extending the gratitude to all the professors and team members at MDSLab, Department of Engineering, University of Messina.

Outside the Univeristy of Messina, I would like to thank Prof. *Khalid El Baamrani* and Prof. *Noureddine Idboufker* for their help and valuable advices.

Family and friends

I am indebted to both of my parents *Omar Benomar* and *Aziza Bousseta* to have always been there to support and help me during my studies and life in general. I thank also my brother, *Oussama Benomar* for his undeniable advices and help. I would like to recognize the invaluable assistance of *Fatima Ezzahra Houzaly* who is a big support for me. I will be always grateful for your help, assistance,and encouragement.
This project would not have been possible without the help of all of you. Thank you.

I acknowledge all my friends from MDSLab *Nachiket Tapas*, *Arif Sahbudin*, *Fabrizio De Vita*, *Islem Bejaoui*, *Giovanni Cicceri*, and *Giuseppe Tricomi*.

I also thank all the engineers from SmartMe.IO that I had the opportunity to collaborate with, in particular *Fabio Verboso*, *Nicola Peditto*, *Carmelo Romeo* and *Alfonso Panarello*.

Abstract

The huge and steady growth in terms of the number of distributed devices connected to the Internet, the so-called Internet of Things (IoT), calls for newly developed infrastructure management techniques to deal with the complexity of emerging IoT deployments, especially in light of the growing impact of the sharing economy. In this context, most of the management platforms tackle IoT issues from a high level where IoT data is managed using Cloud oriented solutions. In such a scenario, the approach adopted can be categorized under the data-centric approach where IoT devices are considered as simple mere data generators uploading data towards Cloud platforms that provide, afterwards, processed data to the users.

In order to challenge this mainstream consensus on the relationship between the Cloud and IoT, what is interesting to investigate is the adoption of the Cloud "as-a-Service" approach from a low level when dealing with IoT infrastructure. Indeed, the as-a-Service paradigm provides well-investigated mechanisms for infrastructure and service provisioning; thus the challenge then is to adapt this approach to fit the management of a dynamic, possibly virtualized, infrastructure of sensing and actuation resources. Cloud providers can extend then their offerings portfolios by providing access to shareable IoT resources according to the utility model using access at the lowest level where possible. Besides providing access to virtualized IoT nodes, an interesting capability to enable is related to computing at the network edge (even on the IoT nodes themselves) to meet the requirements of typical IoT applications, such low processing delays and data privacy.

The thesis presents the design and implementation of a set of mechanisms to integrate IoT within the Cloud wisdom. In particular, the approach enables the capability of offering IoT resources (e.g., sensors and actuators) as virtualized resources. Therefore, the virtual IoT instances can take benefits of the resources (e.g., storage, networking and compute) offered by the Cloud, Fog or the edge-based IoT nodes. The premise then lies in engaging the research from a device-centric perspective using the Stack4Things framework.

List of Publications

Peer reviewed journals

1. **Z. Benomar,** G. Campobello, A. Segreto, F. Battaglia, F. Longo, G. Merlino, and A. Puliafito, "A fog-based architecture for latency-sensitive monitoring applications in industrial internet of things," IEEE Internet of Things Journal, 2021
2. **Z. Benomar**, F. Longo, G. Merlino, and A. Puliafito, "A cloud-based and dynamic dns approach to enable the web of things," IEEE Transactions on Network Science and Engineering, 2021
3. **Z. Benomar**, F. Longo, G. Merlino, and A. Puliafito, "Cloud-based enabling mechanisms for container deployment and migration at the network edge," ACM Transactions on Internet Technology (TOIT), vol. 20, no. 3, pp. 1–28, 2020
4. **Z. Benomar**, F. Longo, G. Merlino, and A. Puliafito, "Cloud-based network virtualization in iot with openstack," ACM Transactions on Internet Technology (TOIT), vol. 22, no. 1, pp. 1–26, 2021

International conferences

1. L. D'Agati, **Z. Benomar**, F. Longo, G. Merlino, A. Puliafito, and G. Tricomi, "Iot/cloud-powered crowdsourced mobility services for green smart cities," in 2021 IEEE 20th International Symposium on Network Computing and Applications (NCA), IEEE, 2021, pp. 1–8
2. **Z. Benomar,** F. Longo, G. Merlino, and A. Puliafito, "Deviceless: A serverless approach for the internet of things," in 2021 ITU Kaleidoscope: Connecting Physical and Virtual Worlds (ITU K), IEEE, 2021, pp. 1–8
3. **Z. Benomar**, "Phd forum abstract: I/ocloud: Adopting the iaas paradigm in the internet of things," in 2021 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2021, pp. 412–413

4. G. Cicceri, G. Tricomi, **Z. Benomar,** F. Longo, A. Puliafito, and G. Merlino, "Dilocc: An approach for distributed incremental learning across the computing continuum," in 2021 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2021, pp. 113–120

5. **Z. Benomar**, F. Longo, G. Merlino, and A. Puliafito, "Stack4things: A cloud-based system for building software-defined cities infrastructure," in The 7th Italian Conference on ICT for Smart Cities and Communities, 2021

6. **Z. Benomar,** F. Longo, G. Merlino, and A. Puliafito, "A stack4things-based web of things architecture," in 2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics), IEEE, 2020, pp. 113–120

7. **Z. Benomar**, G. Campobello, F. Longo, G. Merlino, and A. Puliafito, "Fog-enabled industrial wsns to monitor asynchronous electric motors," in 2020 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2020, pp. 434–439

8. **Z. Benomar**, F. Longo, G. Merlino, and A. Puliafito, "Enabling secure restful web services in iot using openstack," in 2020 IEEE 17th International Conference on Mobile Ad Hoc and Sensor Systems (MASS), IEEE, 2020, pp. 410–417

9. G. Cicceri, C. Scaffidi, **Z. Benomar**, S. Distefano, A. Puliafito, G. Tricomi, and G. Merlino, "Smart healthy intelligent room: Headcount through air quality monitoring," in 2020 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2020, pp. 320–325

10. **Z. Benomar**, G. Campobello, F. Longo, G. Merlino, and A. Puliafito, "A new fog-enabled wireless sensor network architecture for industrial internet of things applications," in 24th IMEKO TC4 International Symposium, 2020, pp. 179–184

11. G. Tricomi, **Z. Benomar**, F. Aragona, G. Merlino, F. Longo, and A. Puliafito, “A nodered-based dashboard to deploy pipelines on top of iot infrastructure,” in 2020 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2020, pp. 122–129

12. G. Tricomi, **Z. Benomar**, G. Merlino, F. Longo, A. M. Longo, and A. Puliafito, “Too(l)smart: A template to make cities "smart",” in The 6th Italian Conference on ICT for Smart Cities and Communities, 2020

13. **Z. Benomar**, F. Longo, G. Merlino, and A. Puliafito, “Enabling container-based fog computing with openstack,” in 2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2019, pp. 1049–1056

14. **Z. Benomar**, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, “A mininet-based emulated testbed for the i/ocloud,” in 2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN), IEEE, 2019, pp. 277–283

15. **Z. Benomar** et al., “Extending openstack for cloud-based networking at the edge,” in 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2018, pp. 162–169

Table of Contents

Declaration of Authorship i

Acknowledgment ii

Abstract iii

List of Publications iv

List of Abbreviations xii

List of Figures xv

List of Tables xix

Introduction 6

1 IoT and Cloud computing 7

1.1 Introduction . . . 7

1.2 Cloud computing . . . 8

1.2.1 Definition . . . 8

1.2.2 Cloud services . . . 8

1.2.3 Resource provisioning in the Cloud . . . 10

1.2.4 Types of Clouds . . . 10

1.3 IoT cloudfication . . . 12

1.3.1 Motivation . . . 12

1.3.2 The device-centric approach . . . 14

1.3.3 I/Ocloud: a multi-tenant IoT solution . . . 15

1.3.3.1 Type of IoT nodes . . . 15

1.3.3.2 Virtual IoT entities . . . 16

1.3.3.3 I/Ocloud virtualization at the network edge . . . 19

1.4 Enabling technologies . . . 20

1.4.1 OpenStack . . . 20

1.4.2 Stack4Things . . . 21

1.5 I/Ocloud use case: Software Defined Cities . . . 25

2 Network Virtualization in IoT **27**

2.1 Introduction . . . 27

2.2 Background . . . 29

2.2.1 Network Virtualization . . . 29

2.2.2 The OpenStack Networking subsystem: Neutron . . . 30

2.3 Network Virtualization in IoT: use cases . . . 33

2.3.1 Partitioning IoT nodes within a same LAN . . . 34

2.3.2 Regrouping IoT nodes from different networks . . . 35

2.3.3 Extending an IoT network with powerful machines . . . 35

2.3.4 Extending an IoT network with personal devices . . . 36

2.4 Stack4Things and Neutron integration design . . . 37

2.4.1 Integration scenario . . . 37

2.4.2 Advanced functionalities of the integration . . . 42

2.5 The Stack4Things virtual networking solution . . . 43

2.5.1 Stack4Things networking APIs . . . 43

2.5.2 Workflow of attaching a board to a virtual network . . . 44

2.5.3 Creating Virtual Networks using Stack4Things . . . 46

2.6 Experimental results . . . 49

3 Containers deployment at the network edge **58**

3.1 Introduction . . . 58

3.2 Edge and Fog computing . . . 59

3.2.1 The Cloud shortcomings . . . 59

3.2.2 Computing paradigms at the network edge . . . 61

3.2.3 Virtualization techniques for Edge/Fog computing . . . 64

3.2.3.1 Virtualization approaches . . . 64

3.2.3.2 Containerizations . . . 66

3.2.4 Containers migration and Fog computing . . . 67

3.3 Zun, Kuryr and IoTronic integration . . . 67

3.3.1 Zun and Kuryr subsystems . . . 68

3.3.2 The integration scenario . . . 69

3.3.3 Container instantiation workflow . . . 74

3.3.4 Container migration workflow . . . 77
3.4 Empowered use cases . . . 79
3.4.1 Fog Computing benefits In IoT . . . 79
3.4.2 Mobility support using Stack4Things . . . 81
3.4.3 Implementing services using Stack4Things . . . 83
3.5 S4T environment emulation . . . 84
3.5.1 Motivation . . . 84
3.6 Emulation system . . . 85
3.6.1 S4T integration with Containernet . . . 86
3.6.2 Emulator use cases . . . 88
3.6.2.1 Emulator for metric estimation . . . 88
3.6.2.2 Emulator as decision making aid . . . 89
4 Enabling SOA in IoT using RESTful Web services 92
4.1 Introduction . . . 92
4.2 Service-Oriented Architecture and IoT . . . 94
4.2.1 Service-Oriented Architecture (SOA) . . . 94
4.2.2 Web services . . . 95
4.2.3 RESTful Web services and IoT . . . 95
4.2.4 Secure Web services in IoT . . . 96
4.3 Technologies background . . . 97
4.3.1 The OpenStack DNSaaS system: Designate . . . 97
4.3.2 Automatic Certificate Management Environment (ACME) . . . 98
4.4 Exposing Cloud-enabled IoT-hosted services . . . 99
4.5 S4T Dynamic DNS system . . . 102
4.5.1 Overview of the Stack4Things Dynamic DNS system . . . 102
4.5.2 Workflow of exposing a service . . . 104
4.6 Implementation and experimental results . . . 106
4.6.1 Testbed description . . . 107
4.6.2 Functional workflow . . . 108
4.6.3 Performance evaluation . . . 110

5 Deviceless: an approach extending Serverless to IoT deployments 114

5.1 introduction 114

5.2 The Serverless paradigm 117

5.2.1 Before Serverless 117

5.2.2 Serverless computing 118

5.2.3 Serverless for Edge computing 119

5.3 The OpenStack FaaS subsystem: Qinling 121

5.4 The Deviceless system description 122

5.4.1 Containers orchestration 123

5.4.2 Functions executions 124

5.4.3 Deviceless functional workflows 125

5.4.3.1 Runtime creation 125

5.4.3.2 Execution workflow 128

5.5 A FaaS-powered flow-based development tool for distributed IoT environments 129

5.5.1 Node-RED 130

5.5.2 Extended Node-RED 131

5.6 Experimental results 133

6 Industrial use case: Stack4Things as a Fog system for Industrial IoT monitoring applications 137

6.1 Introduction 137

6.2 System Architecture And Description 138

6.2.1 The sensing layer 138

6.2.2 The aggregation/compute Layer 140

6.2.3 The middleware layer 141

6.2.4 The application layer 142

6.3 Security mechanisms of the system 142

6.3.1 Sensing layer security services 143

6.3.2 Fog layer security services 147

6.3.3 Cloud layer security services 148

6.4 Case study 149

6.4.1 Analytical model . . . 150
6.4.2 Exploiting the Fog-layer for data aggregation . . . 153
6.4.3 Simulation results . . . 154
6.5 Further advantages of data aggregation . . . 158

Conclusion and future works **160**

Bibliography **163**

List of Abbreviations

6LoWPAN	*IPv6 over Low-Power Wireless Personal Area Networks*
AC	*Alternating Current*
ACME	*Automated Certificate Management Environment*
ADC	*Analog-to-Digital Converter*
AES	*Advanced Encryption Standard*
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
AR	*Augmented Reality*
CA	*Certificate Authority*
CCA	*Clear Channel Assessment*
CDF	*Cumulative Distribution Function*
CLI	*Command-Line Interface*
COE	*Container Orchestration Engine*
CORBA	*Common Object Request Broker Architecture*
CPU	*Central Processing Unit*
CRIU	*Checkpoint/Restore In Userspace*
CRM	*Customer Relationship Management*
CRUD	*Create, Read, Update, Delete*
DDNS	*Dynamic Domain Name System*
DES	*Data Encryption Standard*
DHCP	*Dynamic Host Configuration Protocol*
DNS	*Domain Name System*
DSC	*Distributed Source Coding*
DV	*Domain Validation*
EV	*Extended Validation*
FUSE	*Filesystem in Userspace*
GPIO	*General-Purpose Input/Output*
GRE	*Generic Routing Encapsulation*
GUI	*Graphical User Interface*
HSM	*Hardware Security Module*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
HVAC	*Heating, Ventilation, and Air Conditioning*
IaaS	*Infrastructure as a Service*
ICMP	*Internet Control Message Protocol*
ICT	*Information and Communication Technology*
IoT	*Internet of Things*
IP	*Internet Protocol*
ISM	*Industrial, Scientific and Medical*
ISP	*Internet Service Provider*
IT	*Information Technology*

IWSN	*Industrial Wireless Sensor Network*
JSON	*JavaScript Object Notation*
LAN	*Local Area Network*
LSB	*Least Significant Bit*
LUT	*Lookup Table*
M2M	*Machine to Machine*
MAC	*Medium Access Control*
MCU	*Microcontroller Unit*
MEC	*Multi-access Edge Computing*
MPU	*Microprocessing Unit*
MSB	*Most Significant Bit*
MTU	*Maximum Transmission Unit*
NaaS	*Networking as a Service*
NAT	*Network Address Translator*
NFV	*Network Function Virtualization*
NIST	*National Institute of Standards and Technology*
NTP	*Network Time Protocol*
NV	*Network Virtualization*
OS	*Operating System*
OV	*Organization Validation*
OVS	*Open vSwitch*
PaaS	*Platform as a Service*
PDR	*Packet Delivery Ratio*
PKI	*Public Key Infrastructure*
QoE	*Quality of Experience*
QoS	*Quality of Service*
QUIC	*Quick UDP Internet Connections*
RAM	*Random Access Memory*
RBAC	*Role-based Access Control*
ROM	*Read-only Memory*
RPC	*Remote Procedure Call*
RPL	*Routing Protocol for Low-Power and Lossy Networks*
RSSI	*Received Signal Strength Indicator*
RTT	*Round-Trip Time*
S4T	*Stack4Things*
SaaS	*Software as a Service*
SBC	*Single-Board Computer*
SDC	*Software Defined City*
SDN	*Software Defined Networking*
SINR	*Signal-to-Interference-plus-Noise Ratio*
SOA	*Service-Oriented Architecture*

SOAP	*Simple Object Access Protocol*
SSH	*Secure Shell*
SVD	*Singular Value Decomposition*
TC	*Traffic Control*
TCP	*Transmission Control Protocol*
TPM	*Trusted Platform Module*
UDP	*User Datagram Protocol*
UI	*User Interface*
URL	*Uniform Resource Locator*
URLLC	*Ultra-Reliable Low-Latency Communication*
UUID	*Universally Unique Identifier*
VETH	*Virtual Ethernet*
VIF	*Virtual InterFace*
VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
VMM	*Virtual Machine Monitor*
VN	*Virtual Node*
VNC	*Virtual Network Computing*
VNF	*Virtual Network Function*
VNI	*Virtual Network Identifier*
vNIC	*virtual Network Interface Card*
VPN	*Virtual Private Network*
VXLAN	*Virtual Extensible Local Area Network*
WAMP	*Web Application Messaging Protocol*
WAN	*Wide Area Network*
WoT	*Web of Things*
WS	*WebSocket*
WSDL	*Web Services Description Language*
WSN	*Wireless Sensor Network*
WSS	*WebSocket Secure*
XaaS	*Everything as a Service*
XML	*Extensible Markup Language*

List of Figures

1 Management solutions. . . . 9
2 Cloud-based IoT architectures: (a) IoT devices as a data source for the Cloud, (b) IoT as remote interface to Cloud-based applications, and (c) IoT as an extension of the Cloud resources. . . . 12
3 I/Ocloud instances with attached I/O resources. . . . 17
4 A layered architecture of the I/Ocloud virtualization approach. . . . 18
5 OpenStack architecture. . . . 21
6 Stack4Things subsystems. . . . 22
7 Stack4Things architecture overview. . . . 23
8 Stack4Things Cloud side architecture. . . . 24
9 Lightning-Rod architecture. . . . 24
10 The Software Defined City paradigm. . . . 26
11 Architecture of Neutron subsystem. . . . 31
12 Nova compute node/binding-host design. . . . 32
13 The MENO end-to-end communication concept. . . . 33
14 Regrouping IoT nodes in different virtual networks. . . . 34
15 Overlays extending IoT networks with Cloud-based VMs and personal devices. . . . 36
16 A hybrid overlay composed of IoT nodes, Cloud-based VMs and a personal device. . . . 37
17 The Cloud-side architecture of the Stack4Things network virtualization system. . . . 38
18 The node-side architecture of the Stack4Things network virtualization system. . . . 39
19 Low-level functional diagram of bridged tunneling over WS (IoT nodes managed by the same WS tunnel agent). . . . 40
20 Low-level functional diagram of bridged tunneling over WS (IoT nodes managed by different WS tunnel agent). . . . 40

21 Low-level functional diagram of bridged interconection between an IoT node and a Nova VM. . . . 42
22 The workflow of attaching an edge IoT node to a specific logical network. 45
23 Throughput variation when the edge node is uploading/receiving TCP traffic to/from a Cloud-based instance. . . . 51
24 Comparison of the latency experienced using a Public-IP based deployment and the S4T overlay. . . . 52
25 Comparison of the virtual networking performance with a UDP traffic using the two versions of Socat. . . . 53
26 CPU usage of the WS tunnel server and Socat when managing UDP packets with 80 bytes payload length sent by different instances. . . . 54
27 CPU usage of the WS tunnel server and Socat processes varying traffic sources and the payload length. . . . 55
28 Overview of Cloud, Fog, Mist and Edge Computing in an IoT smart environment context. . . . 62
29 Platform/Hypervisor-based virtualization and Containerization comparison. . . . 65
30 Stack4Things Cloud-side containerization subsystem architecture. . . . 68
31 Stack4Things board-side containerization subsystem architecture. . . . 70
32 Low-level functional diagram of bridged tunneling over WS for edge-based containers (boards managed by the same WS tunnel agent). . . . 72
33 Low-level functional diagram of bridged tunneling over WS for edge-based containers (boards managed by different WS tunnel agent). . . . 72
34 A container instantiation workflow using the S4T system. . . . 75
35 A container migration workflow using the S4T system. . . . 78
36 Use cases for containers migration 80
37 S4T Fog containerization use case. Two different services are depicted in this scenario, i) a Traffic Control (TC) service that collects data based on the smart cameras and traffic lights deployed across different sites ii) an Augmented Reality (AR) service following the user during his/her mobility. . . . 82

38 Integration design of S4T with Containernet. 86
39 S4T Cloud-based overlay in the emulation deployment. 87
40 Designate subsystem architecture. 98
41 The WebSocket tunneling system. 101
42 DDNS Cloud-side system architecture. 103
43 The device side Web services system. 104
44 A detailed workflow description of exposing a service (a Web server in this case) hosted on an edge IoT node. 105
45 The Stack4Things-based routing mechanism. 109
46 Latency experienced with the WS tunnel when using packets with 40 bytes of payload. 110
47 Latency experienced with the WS tunnel when using packets with 400 bytes of payload. 111
48 CPU usage on the Raspberry Pi 3 Model B+. 113
49 A Software Defined City as closed-loop system. 116
50 Management responsibilities in PaaS and FaaS 118
51 The Cloud-side Deviceless system architecture. 121
52 The device-side FaaS system architecture. 125
53 A runtime instantiation workflow. 126
54 A function execution workflow. 128
55 Node-RED browser-based flow editor. 130
56 Integration of the flow-based Node-RED development tool with the Deviceless approach. 132
57 Node-RED node for Qinling: configuration editor 132
58 Qinling node-based example flow. 133
59 Graph and gauge generated by the Qinling functions. 133
60 Monitoring system layers. 139
61 Block diagram of the sensor board used in our experiments for interfacing 3-phase motors. The board is equipped with all the necessary sensors for measuring voltages, currents, speed, temperature and mechanical vibrations. 140

62 Block diagram of the encryption algorithm used for the *Sensing layer* . 144
63 Voltage samples corresponding to a single packet transmission, with and without encryption. . . . 146
64 Proposed Architecture . . . 150
65 Packet Delivery Ratio (PDR) as a function of the traffic generation rate (λ) for 20-node (red line) and 5-node (black line) star-topology WSNs. 152
66 Omnet++ simulation environment . . . 154
67 Packet Delivery Ratio (PDR) for different traffic generation rates λ in the case of: 1) a cluster with 5 nodes (black line); 2) a cluster-tree network with 4 clusters and without data aggregation (blue line); 3) a cluster tree-network with 4 clusters with data aggregation (red line). . . 155
68 End-to-end latency for different traffic generation rates λ in the case of: 1) a cluster with 5 nodes (black line); 2) a cluster-tree network with 4 clusters and without data aggregation (blue line); 3) a cluster tree-network with 4 clusters with data aggregation (red line). . . . 155
69 Cumulative Distribution Function (CDF) of packet delay for λ =10 pk/s and λ =25 pk/s. . . . 156
70 Recovered currents with and without data aggregation. . . . 158
71 Mean relative error on the estimation of maximum current with and without SVD-based data aggregation for different values of SNR. . . . 159

List of Tables

1 The IoTronic RESTful networking APIs. 43
2 Comparison between different IoT network virtualization approaches. . 47
3 Statistical TCP results of throughput (Mbps) during an upload/download from/to the IoT node. 51
4 Statistical TCP results of latency (ms) during an upload/download from/to the IoT node. 51
5 WS tunnel client and Socat averaged CPU usage during 5 mins with different packets data sizes. 56
6 The Stack4Things IoTronic RESTful containerization APIs. 74
7 Throughput results (in Mbits/s) while uploading from an edge-device to a Cloud instance 88
8 Links latency (point-to-point) without involving the S4T overlay. 89
9 Links bandwidth (point-to-point) without involving the S4T overlay. . . 89
10 Emulation latency results (in ms) using the S4T network virtualization approach. 90
11 Emulation throughput results (in Mbits/s) using the S4T network virtualization approach. 90
12 Averaged results considering the different scenarios possible (based on the S4T network virtualization approach). 91
13 WS tunnel client and `Socat` CPU usage with different packets' payload lengths. 112
14 The resources usage of the Deviceless system on an IoT node 134
15 Execution time comparison between Serverless and Deviceless 135
16 802.15.4 MAC parameters used for simulation and analytical results. . . 150
17 End-to-end latency and PDR of DISCUS and proposed SVD-based aggregation scheme for two different traffic generation rates. 157

Introduction

The Internet of Things (IoT) allows us to interconnect objects, places, people, and any other physical entity to the Internet. By attaching communicating devices to them (e.g., sensors and actuators), digital representations of our physical goods can be created. The new mass of data thus generated by these devices can be gathered afterwards to monitor/control specific resources and activities. Indeed, the scope of IoT capabilities consists of diverse settings and use cases including healthcare, environmental monitoring, security/surveillance, retail, and more [1].

Recently, the IoT field attracted most of the biggest companies and organizations to determine its abilities and potential. The vast field of applications makes the IoT an interesting profit-making domain generating an immense economic value. According to McKinsey Global Institute [2], by the year 2025, IoT could spawn an annual economic impact between USD 3.9 trillion and USD 11.1 trillion. In fact, several new services can be conceived using the plethora of IoT devices taking ground. Based on the world population and number of IoT devices [3] [4], we can assume that by 2025 a person will own, on average, around nine connected devices communicating with the Internet. Due to this overwhelming expansion in terms of variety and number of distributed connected devices, dealing with such a complex environment is challenging. Newly designed mechanisms and systems are required to deal with the typical constraints of IoT deployments. Besides, the massive amount of heterogeneous data, so called Big Data [5], generated by the IoT devices needs to be properly managed and processed. In this context, the mainstream solution adopted in the IoT landscape is resorting to the (limited) data-centric [6] approach provided by the Cloud computing paradigm. In this design, the Cloud is typically leveraged as is and the IoT devices are seen as mere data producers. Indeed, although the Cloud computing providers (e.g., Amazon and Microsoft) offer IoT services, they only help with the collection, distribution and processing of IoT data. The IoT infrastructure that produces this data is obviously deployed outside the datacenters and must usually be connected and managed by the application developers themselves [7] [8]. In this regard, the need to

own the infrastructure can be a market barrier, especially for small and medium sized software companies that cannot afford the related costs. For these companies, it might be beneficial to focus only on applications development and possibly rent the necessary IoT sensors and actuators [9]. In addition, if we take into consideration the emergence of the new advanced and complex concepts, such as the smart city [10], the IoT services providers have to get authorization to install IoT nodes in public domains. Such a problem is not encountered in private domains/campuses, but it would become a significant hurdle to overcome in wide and large-scale deployments. In addition, albeit IoT applications may use the same kind of sensor data, each application provider has to deploy his/her own infrastructure. Actually, in most of the deployments, as is the case for Wireless Sensor Networks (WSNs), applications are, usually, embedded on the devices thereby, leading to redundant deployments and underutilization of the available resources [11]. The provisioning of emerging IoT-enabled services in a cost-efficient manner while decoupling applications from the underlying physical infrastructure and configurations is still an ongoing challenge.

Along with the accelerating pace of development of powerful and flexible embedded systems, characterized by reprogrammable behavior and ease of use, such things are often gaining a "smart" labeling to indicate this evolution. Ubiquity, in terms of availability of cheap resources (often coupled with free and open software tools), as well as ever higher board reconfigurability and embedded processing power may be taken for granted in specific scenarios. Thus, such a stimulating albeit challenging scenario calls for suitable approaches, technologies, and solutions to overcome the limitation of the Cloud/IoT data-centric integration approach (e.g., non real time data, no users-initiated interactions with actuators). Ideally, at the very least, (fleets of) devices bought from a diverse range of vendors, should be manageable by resorting to a unified framework, reconfigurable on-the-fly at runtime, even if already deployed (i.e., remotely), and repurposed for a variety of duties, possibly multiplexed onto the same resources concurrently when constraints allow for it.

In the interest of using the infrastructure more efficiently, the Cloud "as-a-Service" model brings convenient features enabling on-demand access to a shared pool of

resources. By integrating IoT within the Cloud ecosystem at the infrastructure level, decoupling IoT services from the infrastructure can be achieved. In fact, in the Infrastructure-as-a-Service (IaaS) Cloud computing model, the users can, on-demand, access/use virtualized instances, such as Virtual Machines (VMs) or containers through Secure Shell (SSH) protocol, Web-based virtual console, or Virtual Network Computing (VNC). Moreover, the IaaS model offers significant flexibility for the users by enabling complete control over the virtual instances, including their networking configurations. Accordingly, the idea is to provide a full integration between IoT and the Cloud by virtualizing IoT resources (i.e., sensors and actuators) and providing them to users/developers as Cloud resources thus, extending the providers' IoT portfolios by offering developers the possibility of renting tailored IoT virtual infrastructure.

To conceive a solution integrating IoT with the Cloud, an important aspect to take into account is that providing virtualized IoT instances at the datacenters level cannot always satisfy the requirements of particular IoT applications. Indeed, the Cloud computing resources are usually concentrated in a few datacenters which are considerably far from the vast majority of data producers. Such a considerable distance from the IoT devices leads to some drawbacks (first and foremost, high round-trip latencies) that are not acceptable for typical time-sensitive applications and services. For such applications, it might be possible to process the sensed data at the edge and transmit only pertinent processed information back to the central datacenter (when required). The virtualization of IoT resources has to be then managed at the network edge (on the devices themselves or in their proximity) while preserving the Cloud characteristics (e.g., multi-tenancy and transparency).

In the context of exploring new interaction methods with IoT deployments, our view aims at fully integrating IoT within the Cloud wisdom and adapt the typical as-a-Service Cloud model to IoT. An (IoT-oriented) view aligned with the DevOps [12] trend aiming to accelerate the delivery of software products. Besides, we would like to enable edge computing capabilities in certain cases to overcome the Cloud drawbacks.

Structure of the dissertation

In this work some approaches and solutions are presented to address the aforementioned challenges while exploring new ways to make use of an IoT distributed infrastructure. This dissertation is organised in six chapters, excluding the closing one and this introduction, as outlined in the following:

- **Chapter 1** discusses the different IoT/Cloud integration patterns. Besides, we introduce the basic concepts behind our device-centric, service-oriented, Cloud-mediated approach for sensing and actuation, including a high-level architecture. In particular, a description of the OpenStack-based Stack4Things (S4T) middleware is provided. Moreover, the different background concepts used throughout the thesis are introduced. This chapter is based on the following publication:
  - "PhD Forum Abstract: I/Ocloud: Adopting the IaaS Paradigm in the Internet of Things," **Benomar** et al. [13]
- **Chapter 2** describes a mechanism for Cloud-enabled network virtualization in IoT. The solution is implemented within S4T and is meant to support and simplify the management of wide-area heterogeneous sensor-/actuator- hosting nodes. A Cloud user can deploy overlay networks among distributed IoT nodes as well as Cloud-based instances to make them seem on the same physical network. Real-world examples and experimental results of the solution are outlined. This chapter is based on the following publications:
  - "Extending openstack for cloud-based networking at the edge," **Benomar** et al. [14]
  - "Cloud-based Network Virtualization in IoT with OpenStack," **Benomar** et al. [15]
- **Chapter 3** reports a solution to enable the sharing of an IoT infrastructure through containerization. For enhanced networking capabilities, the networking driver introduced in Chapter 1 is being used to provide advanced networking

services for the edge-based virtual IoT instances. The usage of the system to enable the Fog computing paradigm in an IoT context is reported as well. Furethermore, a description of an S4T emulation environment for performance evaluation is detailed. This chapter is based on the following publications:

  - "Enabling container-based fog computing with openstack," **Benomar** et al. [16]
  - "Cloud-based enabling mechanisms for container deployment and migration at the network edge," **Benomar** et al. [17]
  - "A Mininet-Based Emulated Testbed for the I/Ocloud," **Benomar** et al. [18]

- **Chapter 4** introduces a set of mechanisms to enable the use of a Service Oriented Architecture (SOA) design in IoT. In particular, the approach exposes, to the Web, the distributed IoT resources (either physical or virtual) as Representational State Transfer (REST) enabled Web services. The solution, in this context, provides new services by dynamically assigning globally resolvable domain names to identify the physical/virtual dispersed IoT nodes even when deployed behind networking middleboxes. This chapter is based on the following publications:

  - "A Stack4Things-based Web of Things Architecture," **Benomar** et al. [19]
  - "Enabling Secure RESTful Web Services in IoT using OpenStack," **Benomar** et al. [20]
  - "A Cloud-based and Dynamic DNS approach to enable the Web of Things," **Benomar** et al. [21]

- **Chapter 5** introduces a new computing paradigm in the IoT landscape named Deviceless. Specifically, the new approach aims at extending the Cloud-based Serverless computing model down to the network edge. In particular, a Cloud user (i.e., a developer) can make use of a distributed IoT infrastructure with hosted resources (e.g., sensors and actuators) in a Serverless-like fashion. A use case of exploiting the new paradigm to conceive data pipelines leveraging a

distributed IoT infrastructure is reported as well. This chapter is based on the following publication:

- "Deviceless: A Serverless Approach for the Internet of Things," **Benomar** et al. [22]

- **Chapter 6** presents an industrial use case where the S4T edge computing enabling services are exploited. Concretely, an S4T-based Fog architecture is introduced to provide edge computing capabilities by deploying a data aggregation algorithm. A set of advantages of using the system are discussed and reported based on a real industrial monitoring use case. This chapter is based on the following publications:
  - "Fog-Enabled Industrial WSNs to Monitor Asynchronous Electric Motors," **Benomar** et al. [23]
  - "A new fog-enabled wireless sensor network architecture for industrial internet of things applications," **Benomar** et al. [24]
  - "A Fog-based Architecture for Latency-sensitive Monitoring Applications in Industrial Internet of Things,"**Benomar** et al. [25].

An ending chapter, which is not numbered, wraps up this work with the conclusions and proposes further work related to the presented subjects.

Chapter 1

IoT and Cloud computing

1.1 Introduction

Taking into account the massive amounts of IoT devices, available by the billions very soon, generate means stepping into the BigData realm, usually tackled at the higher levels, i.e., in terms of centralized treatment, analytics and storage [26]. Due to the rapid adoption of IoT services, the problem of storing, processing, and accessing large amounts of data has arisen. In this regard, the Cloud computing paradigm [27] is improving the success of IoT thanks to the facilities and services it provides. The use of Cloud platforms in conjunction with IoT has become a kind of catalyst by bringing up many advantages to deal with data management and processing. IoT devices with sensing capabilities can upload the gathered information about their surrounding physical environments to the Cloud as input for intelligent monitoring/actuation systems. This IoT/Cloud integration aims at transforming IoT data into insights and consequently, driving cost-effective services and applications.

Within the trend that aims to adopt the service-oriented computing and its extension to the Everything-as-a-Service (XaaS) approach [28], several solutions opt for adapting the "as-a-Service" paradigm in IoT environments. Nonetheless, most of the approaches consider the Cloud as an extended application domain (i.e., a data sink) where data generated by IoT devices are stored and then retrieved according to a data-centric approach [29] [30] [31]. Such solutions can obviously provide enough resources to process IoT data yet, they are limited since the users cannot customize the business logic running on the IoT devices and therefore, they can only make use of the data stored on the Cloud as is. In addition, the data-centric approach does not provide user-initiated interactions with the actuators.

In this chapter, we dive into the details of the Cloud and IoT integration. Specifically, we firstly introduce the Cloud computing paradigm and its different services and implementations. Afterwards, we present our view that aims at integrating IoT within the Cloud wisdom at the deepest level possible (i.e., the infrastructure), thus enabling users to share the IoT infrastructure by virtualizing the nodes hosted resources (i.e., sensors and actuators).

1.2 Cloud computing

1.2.1 Definition

The Cloud computing paradigm has an instrumental role in expanding the benefits of computing, storage, and networking capabilities to Cloud-based applications. The National Institute of Standards and Technology (NIST) defines Cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort and without the service provider involvement [27]. The seamless reconfigurability provided by the Cloud is beneficial to offer the pay-as-you-go payment method [32]. This efficient billing model plays a relevant role in adopting Cloud services since it allows the users to conveniently access and use remote resources (e.g., compute, storage) and data management services whilst being charged for only the amount of resources being used. Today, big Information Technology (IT) companies like Google, IBM, Microsoft, and Amazon provide large datacenters to host users' applications and services.

1.2.2 Cloud services

The Cloud provides different services depending on the application developers needs. We mention here the three most known Cloud offerings, namely IaaS, Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). Cloud users can choose one of

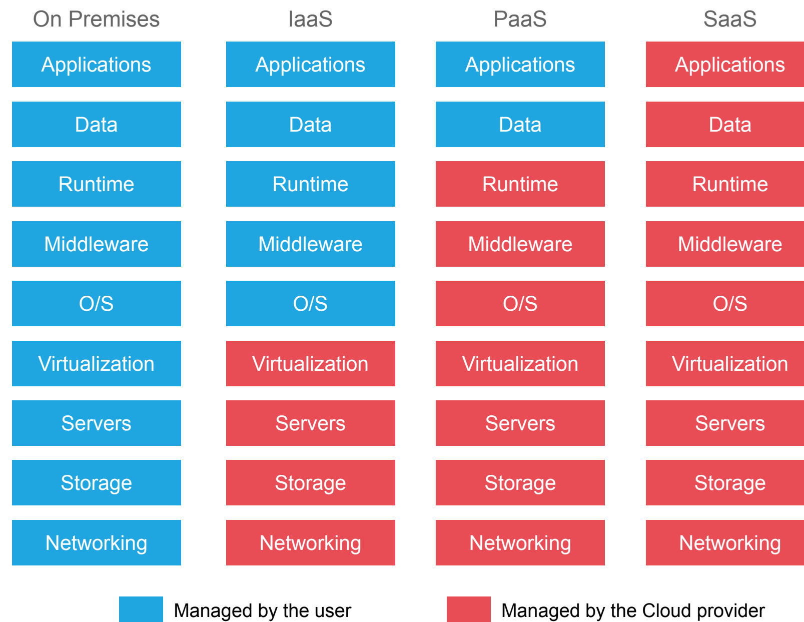


Figure 1: Management solutions.

these services based on the level of control he/she needs over the infrastructure.

In the IaaS computing model, a user has low-level access to the IT infrastructure with regards to processing, storage, and networking resources [33]. An IaaS user can configure his/her instances (often offered as standalone VMs) in terms of hardware and software. In particular, the control over the instance allows the Cloud consumer to customize its hardware configuration, such as the number of central processing unit (CPU) cores, Random Access Memory (RAM) capacity, and storage capacity. In addition, the user is also responsible for managing the system-level software [34].

In the PaaS computing model, the user does not have the same control over the infrastructure as in the IaaS. A PaaS user instead delegates the infrastructure hardware and software management duties to the Cloud provider. The provider then delivers hardware and ready-to-use software tools over the Internet. Usually, these tools are needed for applications development [35]. The user focuses only on the application business logic without worrying about the software/hardware configurations.

As for the SaaS model, the Cloud provider offers an entire application stack (i.e., the application runs completely on the provider's datacenter). The user delegate, in

this case, all configuration/management tasks to the provider. Using the SaaS model, the user needs only to log in and then use the service through a specific application/software (e.g., a Web browser). Examples of SaaS include Gmail, Dropbox, and Microsoft 365.

Based on the discussions above, the Cloud is a platform that can be utilized for distinct use cases by a variety of end-users depending on their needs. Figure 1 illustrates the three different Cloud services and their relationship with the underlying Cloud infrastructure. It also depicts the infrastructure management responsibilities in the three Cloud offerings (i.e., IaaS, PaaS, and SaaS).

1.2.3 Resource provisioning in the Cloud

Resource provisioning is a key feature for the Cloud computing paradigm [36]. Since the demands of the Cloud tenants can not be known beforehand and can change over time, setting a static allocation of resources can lead to a performance drop by either over-provisioning or under-provisioning them [37]. The core concept of Cloud computing is based on provisioning the resources in a flexible fashion on the basis of the demand/load. To optimize resource usage in a Cloud, the providers opt for virtualization technologies and efficient provisioning systems to manage the hardware and software configurations of their datacenters. Besides, as already mentioned, since it is difficult to estimate the usage of the applications/services hosted on a Cloud, the providers adopt the pay-as-you-go billing plan [38] with a demand-driven resource provisioning.

1.2.4 Types of Clouds

Cloud deployments can be categorized into four types private Clouds, community Clouds, public Clouds, and finally hybrid Clouds [27]. The private Cloud is an IT model that provides a dedicated proprietary environment for a single business entity. Like other types of Cloud computing environments, the private Cloud provides extensive and virtualized computing resources through physical components stored

on-premises or in a vendor's datacenter. One of the main benefits of deploying a private Cloud is the degree of control gained by the organization and the high privacy it ensures. As the private Cloud is only accessible to a single organization, this later has the possibility to configure the environment and manage it in a way adapted to its specific IT needs. However, in this model, users do not benefit from the pay-as-you-go billing method.

The community Cloud, more rarely used, consists of sharing a given Cloud datacenter between several companies with the same requirements in terms of security and confidentiality. It is therefore akin to a shared private Cloud.

A public Cloud is a service accessible to everyone via the Internet. This service can consist on provisioning resources, such as storage (e.g., Dropbox) or computing power (as offered by Amazon EC2), or even applications (e.g., Customer Relationship Management (CRM) tools). Public Cloud providers benefit from enormous storage and compute capacities allowing them to serve all their users at once. They generally have a global presence. However, the services provided by public Cloud platforms are adaptable up to a certain limit, so they may not quite meet all users needs. Besides, using a public Cloud can also be economical as no investment is necessary to set it up. Usually, the user only pays for what he/she consumes, which is also a good point for businesses up to a certain level of use as from a certain volume of data transfer, the economic advantage may decrease.

The fourth Cloud implementation is the hybrid Cloud which is a combination of private and public Clouds. Nowadays, the needs of companies in terms of information systems are constantly evolving and becoming more complex and specific. The hybrid Cloud makes it possible to best respond to these concerns thanks to a distribution of resources and a precise definition of the roles of each Cloud in the overall operation of the information system. With a hybrid Cloud, the user can benefit from the security of a private Cloud to store sensitive data, and at the same time, the flexibility and speed of a public Cloud to size an infrastructure. The combination of the different Cloud deployments can be achieved through standardized or proprietary technologies [39].

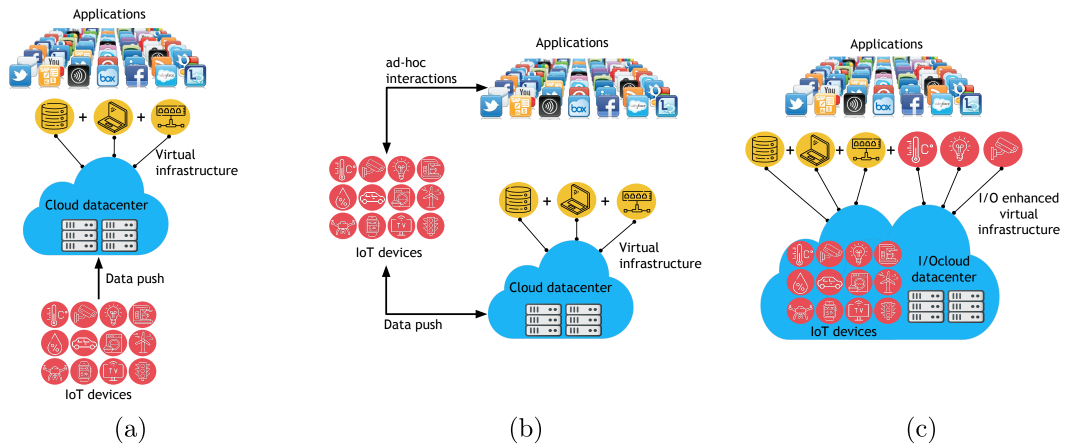


Figure 2: Cloud-based IoT architectures: (a) IoT devices as a data source for the Cloud, (b) IoT as remote interface to Cloud-based applications, and (c) IoT as an extension of the Cloud resources.

The Cloud computing paradigm is meant to provide users access to a pool of computing resources for ubiquitous computing. However, even though the Cloud has been an effective solution to develop a wide range of applications [40] [41], it induces several issues mainly related to the time required to access the services as well as privacy issues [33]. Besides, the fast expansion in the amount of data generated at the network edge by a rising number of connected devices necessitates data processing close by the connected devices in order to satisfy the applications demands (e.g., low processing delays).

1.3 IoT cloudfication

1.3.1 Motivation

The growing interest in the IoT comes from the ubiquity of devices with sensing and actuating capabilities that act as programmable gateways to the physical world. In general, most of the approaches to fully exploit the IoT ecosystem rely primarily on adopting the Cloud paradigm. However, as discussed earlier, most of the solutions adopted can be classified in the category of data-centric solutions [42] [43] [30] where the only operations permitted are data manipulation ones (see Figure 2a). In such a management design, the IoT devices are considered as mere/simple sources generating

data or, at most, a bidirectional remote interface usually non-reconfigurable [44].

To have complete control over an IoT infrastructure and enable the possibility to reprogram it, users may opt for vertical solutions to deploy and manage their infrastructure (see Figure 2b). Nevertheless, a similar solution does not enable the application developers to share an IoT infrastructure, thus each user has to set up its own infrastructure, which is a limitation for the adoption of IoT applications on a larger scale as the capital expenditure of IoT infrastructure is often nontrivial. Besides, authorizations to deploy IoT nodes in public domains for large-scale deployments can be hard to acquire. In addition to the limitation of sharing the IoT infrastructure, the data-centric-oriented solutions are based on sending all the generated data towards a datacenter. For instance, an IoT sensing deployment with sensors producing data at a high data rate from a large number of sensors can incur significant operational expenditure in terms of bandwidth [45], storage and processing cost. In such scenarios, it might be useful to process the generated data at the edge and transmit only preprocessed information to the Cloud, thus avoiding high bandwidth and storage use. On the other hand, data processing at the network edge is also useful to satisfy the requirements of typical time-sensitive applications that cannot tolerate delays introduced when relaying on a faraway Cloud.

We believe that technologies are now mature enough to challenge the mainstream consensus on the relationship between the Cloud and IoT. In the context of enabling multi-tenant IoT infrastructure, our view aims at stretching the Cloud paradigm by adapting the Cloud-enabled metaphors to the IoT infrastructure; that translates into viewing IoT as a natural extension of the datacenter as in Figure 2c. By doing so, it becomes possible then to pool a diverse range of geographically dispersed devices as infrastructure resources, together with standard Cloud facilities, such as compute, storage, and networking facilities.

Problem statement: Is it possible to build a public, multi-tenant IoT infrastructure capable of edge computing (when needed) and that can offer an IoT infrastructure as a extension of a Cloud deployment?

1.3.2 The device-centric approach

Putting sensing-related duties into context, a way to manage effectively huge collections of incoming data consists in minimizing the communication overhead by bringing computation closer to data, and not the other way around [46]. As already discussed, the solutions usually used in IoT systems adopt a data-centric approach based on the Cloud. An interesting paradigm to investigate in order to overcome the IoT/Cloud integration issues is to provide the users actual, even if virtual, sensing/actuation resources at the network edge instead of only the data they generate.

Although the Cloud data-centric oriented approach can be an effective solution to adopt in a set of scenarios, the device-centric one bring up a set of benefits [46]:

- **Decentralized control:** distributed policies can be set up on sensors and actuators through customization features allowing to deploy user-personalized software on the sensing/actuation entities.
- **Onboard data prefiltering and processing:** using edge computing, data generated by the sensors/actuators can be filtered and/or preprocessed on the IoT devices themselves, thus reducing latency to take decisions while enhancing users privacy.
- **Reduced number of data transfer:** by enabling edge computing on top of IoT devices, direct links between the users and the sensing/actuation devices can be established, thus requiring just one data transfer. Instead, in the data-centric approach, at least two transfers are required since data are first stored into a database that exposes them afterwards to the users.
- **Composition and repurposing:** a user can implement customized logic on an IoT node; thus, he/she can aggregate, compose, and/or repurpose sensing resources.
- **Enhanced security:** the device-centric approach enhances security/privacy in IoT. In fact, the approach enables shifting processing tasks from the Cloud to

the devices and vice versa, according to the required level of security and the device capabilities.

- **Information dissemination:** data are disseminated through the distributed sensing infrastructure, hence allowing the implementation of distributed data delivery algorithms to optimize data transfer.

To put into action the device-centric view together with multi-tenancy capabilities in IoT, we need to provide a set of functionalities in the areas of sensors and actuators virtualization to have (virtual) sensing resources available as endpoints, e.g., registered and enumerable, as well as actionable.

In the following, we introduce an overview of our I/Ocloud approach [44] [13] that aims at providing the capability of offering standardized and generic programming capabilities on top of IoT resources, regardless of the underlying infrastructure configurations. In addition, the approach keeps the ability to make use of the unique characteristics of an IoT-enhanced distributed datacenter, such as the availability of nodes at the edge, which may then be used as computing infrastructure to deal with data (pre)processing.

1.3.3 I/Ocloud: a multi-tenant IoT solution

1.3.3.1 Type of IoT nodes

An IoT resource is a connected entity (for instance, a sensor/actuator) that can be exported and attached while not necessarily being programmable. Among the wide range of available IoT resources, we can mention, for example, sensors attached to General-Purpose Input/Output (GPIO) digital pins of a Single-Board Computer (SBC), an accelerometer hosted on a smartphone, optical heart rate sensor within a smart watch, and a wireless sensor node.

As such, an IoT node can be defined then as any computing entity capable of hosting physical IoT resources (e.g., sensors and actuators) while running a user-defined

logic. Examples of IoT nodes include smartphones, SBCs, etc. In general, such kinds of nodes are typically commercialized with limited computing and storage capabilities. Besides, in most cases, these nodes are being used at the network edge and often, deployed behind networking middleboxes, such as Network Address Translators (NATs) and/or firewalls.

1.3.3.2 Virtual IoT entities

As discussed before, our view aims at achieving a seamless integration between the Cloud and IoT by providing the distributed IoT resources (i.e., sensors and actuators) hosted on nodes deployed at the network edge as virtualized Cloud resources. This approach might also be considered as an input/output of a Cloud datacenter, a paradigm that we refer to as I/Ocloud [13]. A Cloud user can then interact with remote IoT resources as they were Cloud-based resources without resorting to ad-hoc or application-level Application Programming Interfaces (APIs). One of the critical functions of the I/Ocloud is to ensure that IoT deployments are engaged as active elements of the Cloud infrastructure while preserving their characteristics (e.g., sensing capability). In fact, the core concept of the Cloud/IoT integration is to redefine virtualization to include IoT nodes' hosted resources. We talk then in our view about I/O virtualization (virtIO).

The I/Ocloud approach extends the virtualization concept to the IoT world by abstracting IoT resources and providing them as virtual ones. An I/O resource can be seen as an instance of a developer-friendly interface for an I/O primitive of its physical counterpart. The abstraction mechanism can concern the entire I/O resources of an IoT node or just a subset of the resources. In addition, the I/O virtualization capability can also regroup, logically, IoT resources from different IoT nodes within the same (logical) entity.

To provide low level access to IoT resources, the I/Ocloud abstraction approach is based on file system virtualization. This choice was made on purpose since many common modern IoT nodes (e.g., Raspberry Pi, Arduino) employ the GPIO pseudo-

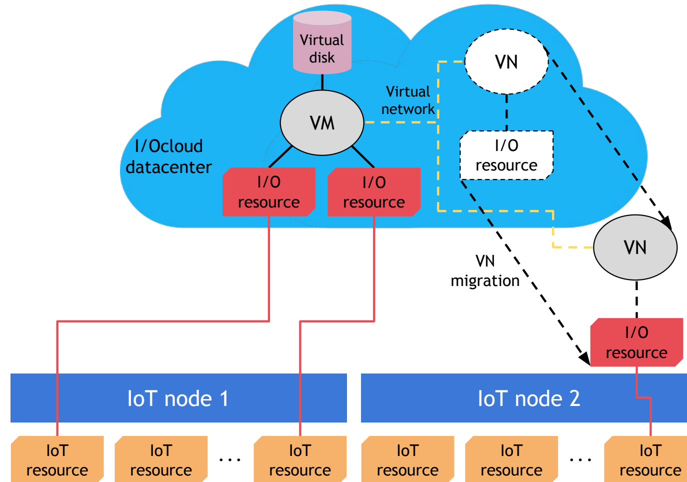


Figure 3: I/Ocloud instances with attached I/O resources.

file system[1] approach in their system software to interact with the boards physical pins interfaces. This I/O virtualization can also be extended to cover nodes virtualization in IoT. Indeed, from our perspective, we would like to provide accessible virtual entities with attached I/O resources (see Figure 3) virtualized through the file system. An I/Ocloud instance is a virtual representation of a physical IoT node, including its physical pins while being able to host user-defined logic and providing, at the same time, interactions with the remote physical IoT resources. Technically, an I/Ocloud instance is a self-contained and isolated environment with a user space-defined file systems. This later file system aims at representing a clone of the remote physical IoT resources within the file system hierarchy of the virtual IoT node.

Figure 4 depicts a layered architecture of the virtualization approach. As shown in the figure, bottom-up, we have sensors and actuators in the case of a physical IoT device (left node), whereas on the right, a compute node belonging to a Cloud datacenter is depicted, featuring no directly attached transducers. In the IoT node, on top of the Linux-based Operating System (OS), we have the pseudo file system (sysfs)-based interface enabling I/O operations to/from the physical pins. To fully virtualize and reproduce the corresponding IoT node as a virtual instance hosted on the physical IoT node, the GPIO pseudo file system of the IoT node has to be supplemented by

[1] A pseudo file system is a hierarchical interface to non-file objects that appear as if they were regular files in the tree of a file system. These non-file objects may be accessed with the same system calls as regular files and directories.

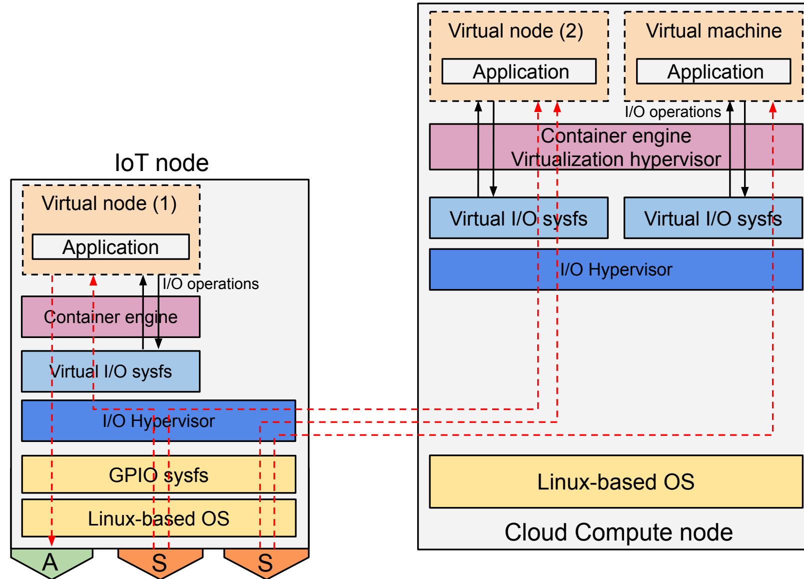


Figure 4: A layered architecture of the I/Ocloud virtualization approach.

a comprehensive clone of the user space setup. The role of the the I/O hypervisor is to expose a virtualized (user space-defined) sysfs (i.e., the /sys filesystem) for each virtual IoT node spawned. The sysfs virtualization can then shift the availability of I/O operations upwards to the corresponding virtual IoT node.

The I/Ocloud virtualization approach can be also enabled on the Cloud by exposing remote transducers on virtual IoT nodes spawned in the Cloud datacenter as shown in Figure 4 by the dashed arrow starting from the sensors on the physical IoT node (on the left), which through inter-hypervisor communication are exposed to a virtual IoT instance hosted on the Cloud compute node (on the right). Wrapping up, the red dashed arrows highlight the final result in terms of I/O virtualization. In particular, the direction of the arrows corresponds either to an input (a sensor) or output (an actuator). In our design, the user space file system is enabled using Filesystem in Userspace (FUSE) [47] over Remote Procedure Calls (RPCs) to ensure remote interactions with the physical IoT resources.

When considering the typical Cloud facilities, we can deploy the I/Ocloud approach either using plain VMs or Virtual Nodes (VNs). The difference between the two instances resides in the instance flavor as in standard Cloud deployment. Once a VM/VN gets instantiated, virtualized I/O resources can be attached to the instance

as they were physically connected regardless of the configuration of the physical nodes hosting them. An IoT application developer can therefore use his/her VM/VN as it is a physical IoT node with sensors/actuators attached to its physical pins (see Figure 3).

1.3.3.3 I/Ocloud virtualization at the network edge

The I/Ocloud virtualization at the edge perspective emerges as soon as we recognize that I/O resources represent an interface to the physical world while the distributed IoT nodes hosting the physical resources can provide a pool of distributed computing nodes at the network edge.

Pushing computing to the periphery of the network means decentralizing business logic as desired, scheduling its execution on the boards themselves, whenever conceivable. This means possibly engaging the Cloud to mainly support communication below the application level when direct interaction among nodes is impaired as a result of network-imposed constraints.

Regarding the VNs management, a VN may be instantiated as an isolated and portable environment (e.g., a lightweight container) either on the datacenter or a remote IoT node. A VN may be deployed at first on the I/Ocloud datacenter and migrated, when needed, toward other infrastructure (e.g., to the edge) to meet, for instance, latency constraints (see Figure 3). Conversely, the VN may be instantiated on an edge-based physical IoT node then offloaded to the Cloud, the case when more compute resources are required, for example. Besides the virtualization of the IoT nodes and their physical resources, the I/Ocloud view goes beyond this virtualization level to provide network virtualization as well. In fact, this is a critical aspect to fully integrate IoT with the Cloud and overcome networking barriers in IoT deployments. The I/Ocloud view aims at making the users able to instantiate personalized networking topologies among any combination of VMs and VNs spanning both, the datacenter as well as Wide Area Networks (WANs) when VNs are deployed at the network edge (see yellow dashed lines in Figure 3). Besides, the networking solution should also cover bare-metal IoT nodes the case when a user own an IoT device or

set of devices and want to include them within his/her IoT deployment.

Based on these discussions about the I/Ocloud, the approach can bring significant benefits, including:

- decoupling the IoT infrastructure and the underlying networking configurations from users business logic,
- enhancing the concept of IoT infrastructure as Code (IaC),
- enabling edge computing to meet the requirements of specific applications,
- enabling a low-level abstraction of IoT nodes and resources (this is important for applications code portability),
- interacting with IoT resources with a high level of granularity (thanks to the pseudo file system);
- overcoming networking barriers in IoT deployments.

1.4 Enabling technologies

1.4.1 OpenStack

OpenStack is a set of open-source software tools for building and managing Cloud computing platforms. OpenStack is a centerpiece of infrastructure Cloud solutions for most commercial, in-house, and hybrid deployments, as well as a fully open source ecosystem of tools and frameworks. Currently, OpenStack allows to manage virtualized computing/storage resources, according to the infrastructure Cloud paradigm. In Figure 5, a conceptual architecture of OpenStack depicting components, as boxes, and the services they provide to other components, with arrows, are shown, respectively. Nova, the compute resource management subsystem, lies at the core of OpenStack and provisions VMs, with the help of a number of subsystems that provide core (e.g., networking in the case of Neutron) and optional services (e.g., block storage, in the

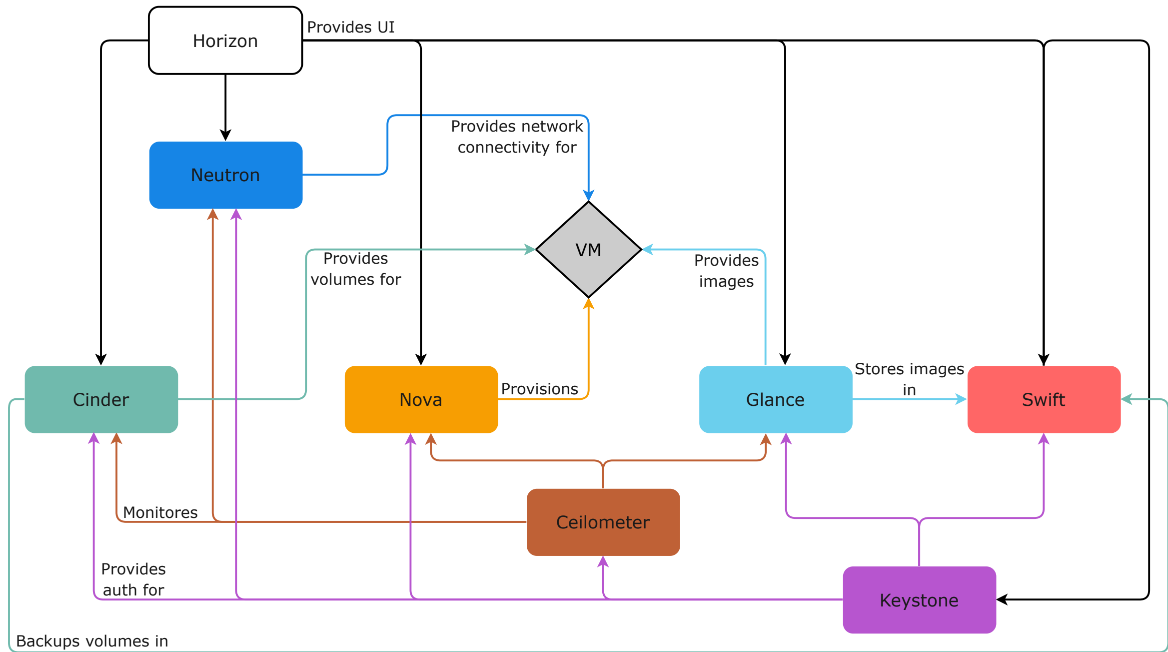


Figure 5: OpenStack architecture.

case of Cinder) to the instances. Horizon is the dashboard and as such, provides either a (web-based) User Interface (UI) or even a command-line interface to Cloud end users. Ceilometer, the metering and billing subsystem, like most other components of the middleware, cannot be fully analyzed on its own, as it needs to interface to, and support, Nova. In particular, while both Nova and any of the aforementioned subsystems exploit a common bus, the former alone dictates a hierarchy on participating devices, including their role and policies for interaction. Indeed, Nova requires a machine operating as Cloud controller, i.e., centrally managing one or more compute nodes, which are typically expected to provide component-specific services (e.g., computing) by employing a resource-sharing/workload-multiplexing facility, such as a hypervisor.

1.4.2 Stack4Things

To achieve the I/Ocloud view discussed earlier, we based our work on the OpenStack open-source project. A first effort extending the OpenStack ecosystem to support the management of IoT deployments has been made by virtue of the S4T middleware [48]. The implementation-driven approach of the project tries to implement

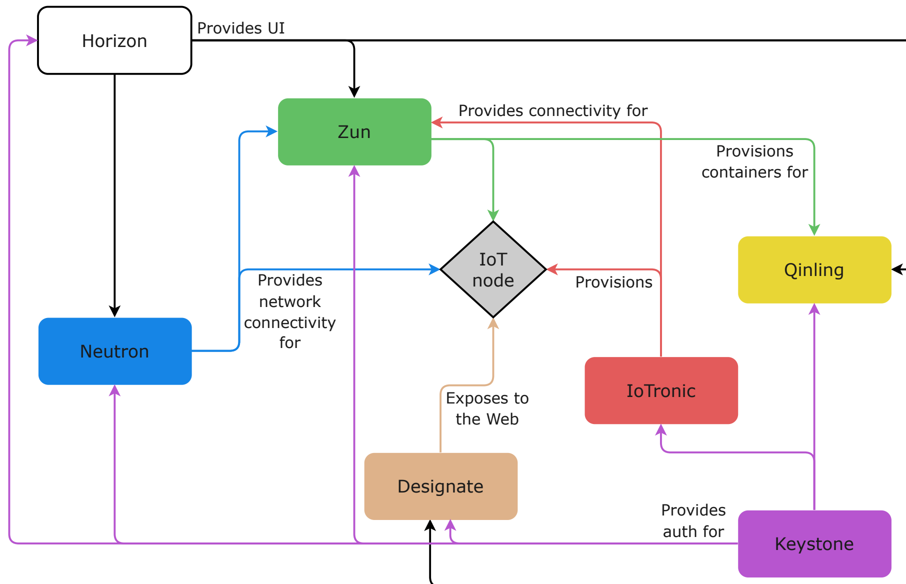


Figure 6: Stack4Things subsystems.

suitable capabilities for IoT infrastructure to join an edge-extended IaaS Cloud. The middleware provides infrastructure-enabling facilities to manage instances at the network edge.

Highlighting this focus, as can be seen in Figure 6, the same kind of conceptual architecture as in Figure 5 is used to represent the subsystems composing our S4T middleware by taking into account just core components for our approach. In particular, introducing a novel subsystem, IoTronic, devoted to the provisioning and configuration of IoT nodes with hosted sensing and actuation resources. Indeed, in place of a VM, in this case, we have a diamond-shaped box symbolizing a (transducer-hosting) IoT node, and corresponding interactions are described as text along the arrows. With regard to the rest of the OpenStack subsystem used within S4T, the networking service, Neutron, has been enhanced to provide network connectivity for both, IoT nodes deployed at the network edge (see Chapter 2) and virtual IoT nodes (i.e., I/Ocloud VNs) instantiated using the Zun subsystem (see Chapter 3). Furthermore, to make the edge-based IoT nodes resources (either virtual or physical) exposed as Web resources, we used the OpenStack Designate subsystem to associate publicly resolvable domain names with the distributed physical/virtual IoT nodes even when deployed within IPv4 masquerade networks (see Chapter 4). In addition to the capabilities of

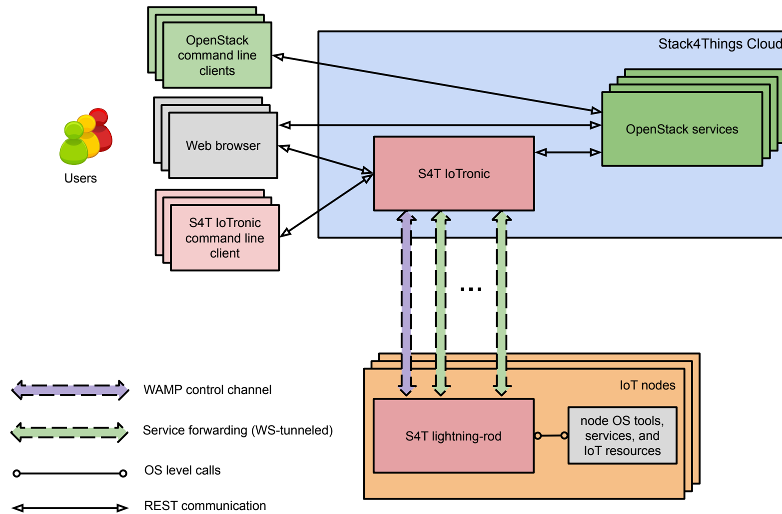


Figure 7: Stack4Things architecture overview.

the I/Ocloud view in providing virtual IoT resources (e.g., sensors and actuators), we also extended our approach to enable the developers to use, in particular situations, Serverless-like interactions as interfaces to the nodes hosted resources (i.e., sensors and actuators). We extended, in this case, the capabilities of Zun and Qinling subsystems (see Chapter 5).

Figure 7 gives a technical high-level overview of the S4T deployment which is split between a datacenter and a number of edge IoT nodes. As hardware setup of the managed nodes, we made a decision, on purpose, to make use of relatively smart (embedded) devices capable of hosting a minimal Linux distro (e.g., OpenWRT) such as SBCs (e.g., Arduino, Raspberry Pi, and Arancino[2]) that are microprocessor (MPU)-powered. Such a hardware configuration makes the IoT nodes capable of hosting Linux-based tools together with different runtime environments, specifically Python and Node.js, that are required by the S4T node agent Lightning-Rod (LR). This later agent is responsible for bridging the remote IoT nodes to the Cloud infrastructure where the S4T IoTronic service is deployed. IoTronic is modeled after the standard design of OpenStack services as exhibited in Figure 8 that depicts the Cloud side architecture of S4T (red subsystem).

The interconnection between IoTronic and LR is built on top of a full-duplex mes-

[2]https://smartme.io/projects/arancino-cc/

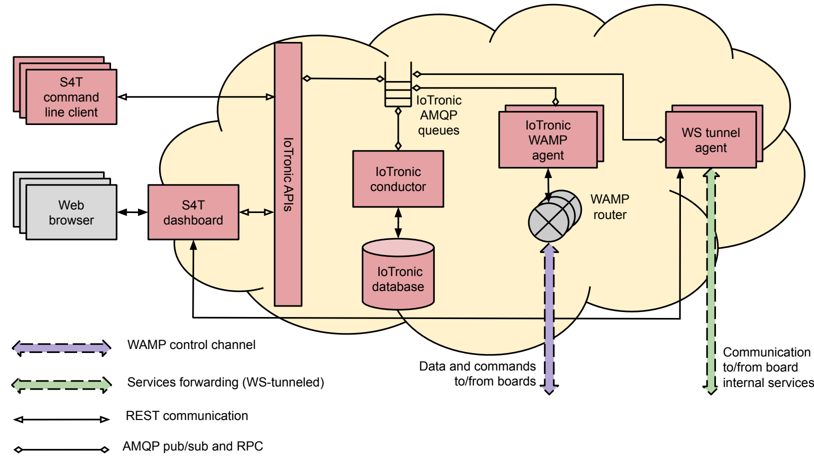


Figure 8: Stack4Things Cloud side architecture.

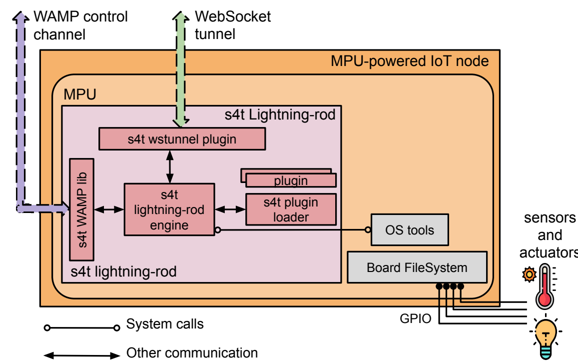


Figure 9: Lightning-Rod architecture.

sage channel used in our design to forward commands from the Cloud to the IoT nodes. Technically, the interconnection is established using a Web Application Messaging Protocol (WAMP)-based communication (violet arrow in Figures 7, 8 and 9). WAMP is an open standard WebSocket (WS) subprotocol stated to provide publish/subscribe (pub/sub) as well as RPC patterns (alongside routed RPCs). IoTronic provides services forwarding through the Cloud (green arrows in Figures 7, 8 and 9). In particular, users/administrators are able to make use of services (e.g., ssh) enabled on remote devices, regardless of their physical networking configuration. Specifically, the user connection requests are forwarded, on the Cloud side, to the S4T IoTronic WS tunnel agent that is a "wrapper" in control of the WS server to which the devices connect through the use of S4T wstunnel libraries (see Figure 9). This particular capability of services forwarding is provided based on a reverse tunneling mechanism using WebSocket [49].

As depicted in Figure 8, at the core of the IoTronic subsystem, the conductor manages the local database that stores metadata about the nodes. The S4T API server exposes a set of RESTful APIs that enable different interactions with the (remote) IoT nodes, either through the IoTronic Command-Line Interface (CLI) or Web-based Graphical User Interface (GUI). For this purpose, the OpenStack Horizon dashboard has been customized with an S4T panel exposing the services provided by IoTronic.

1.5 I/Ocloud use case: Software Defined Cities

A Smart City [10] is an ecosystem of infrastructure and services aiming to bring together society, government, and technology to produce enhanced services (smart mobility, smart environment, etc.). This holistic view calls for an all-encompassing approach to embrace technologies and services, thus providing a broader (or even a global) solution to (smart) city problems. In this light, there is the need for a scalable architecture aiming at reusing, multiplexing, and sharing technologies and services on the urban scale. The goal is to establish a homogeneous ecosystem where multiple applications can scale out to a metropolitan scope, thus underpinning an open and shared Information and Communication Technologies (ICTs) infrastructure made of sensing, actuation, network, processing, and storage resources.

In order to manage heterogeneous and complex socio-technical systems on the scale of whole cities, where both social and technological issues merge, an overarching approach able to deal with all related issues in an all-encompassing fashion is required. Specifically, on the one hand the goal is to provide a uniform representation of connected smart objects by abstracting, grouping, and managing them as a unified ecosystem of smart objects to be configured, customized and contextualized according to the high level, application, requirements. On the other hand, a management layer able to control the ecosystem dynamics, able to map such requirements into lower level ones, implementing and enforcing specific policies to satisfies such requirements is needed.

A suitable solution may therefore lie in adopting a software defined approach,

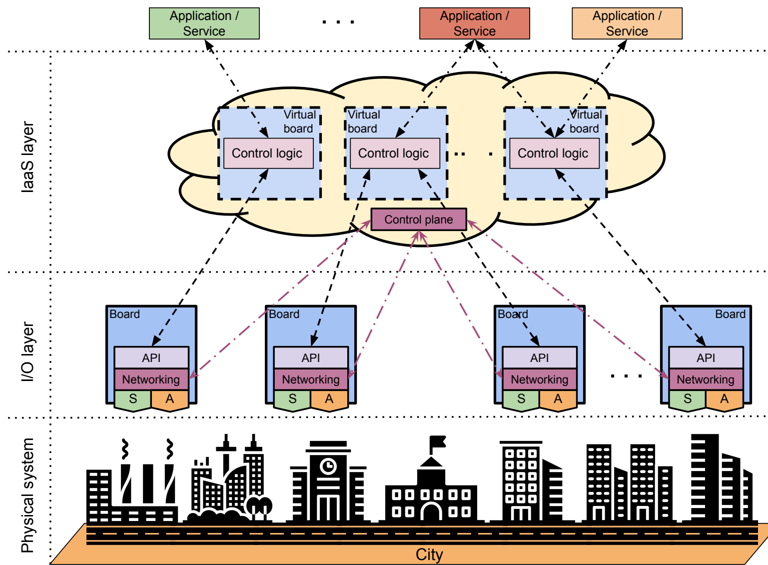


Figure 10: The Software Defined City paradigm.

where basic mechanisms provided by the smart cities objects at data plane, are used by the control plane to implement policies related to application/end user-level requirements. Thus, we talk about Software Defined Cities (SDCs) [50]. The data plane level in this approach is composed of the geographically distributed IoT devices that interact with the physical environment using the sensors and actuators. Regarding the control plane, it is composed of the Cloud-hosted (or edge-based) virtual boards (see Figure 10).

Chapter 2

Network Virtualization in IoT

2.1 Introduction

In a typical IaaS Clouds, users are able to create VMs and access them remotely. In addition, a user has full control over the networking configurations of his/her VMs as he/she can instantiate topologically complex virtual networks among them. In the context of consolidating the Cloud and IoT paradigms, deep and genuine integration between of the two environments may only be attainable at the IaaS level. In fact, an IoT deployment without networking resilience and adaptability makes it unsuitable to meet user-level demands and requirements. Such a limitation makes IoT services adopted in very specific and statically defined scenarios, thus leading to limited plurality and diversity of use cases. By integrating IoT within the Cloud ecosystem at the infrastructure level, decoupling IoT services from the infrastructure and the underlying networking configurations can be achieved.

However, when considering the particularity of IoT environments, they cannot be managed similarly as standard IaaS Cloud-based deployments. Several features and properties significantly diverge from what a typical IaaS environment is characterized by and what it is expected to provide. On the one hand, an IoT deployment is a fully distributed ecosystem with IoT nodes being geographically scattered; accordingly, their reachability is a critical factor that should not be taken for granted. Indeed, the actual IoT landscape is based on nodes deployed, most of the time, behind firewalls with particular security policies/restrictions and/or NATs, especially when the IPv6 addressing scheme could not be an option. On the other hand, contrary to IaaS-oriented datacenters where the provider has complete control over the equipment's physical level and logical layouts (e.g., cabling, networking configurations), IoT de-

ployments do not provide such a level of control. An IoT infrastructure owner, in general, expects not to grant an administrator similarly complete management abilities and privileges over his/her physical infrastructure. For instance, the infrastructure may be assembled in an opportunistic way (i.e., volunteer-contributed) from multiple (IoT) owners, and the requirement to decouple ownership from administrative capabilities, in this case, becomes then even more urgent. Yet, any (standard) network virtualization [51] [52] mechanism will require at least some form of reconfiguration capabilities on the IoT node-side networking facilities.

Considering the aforementioned dictated limitations, decoupling IoT applications logic from the underlying networking configuration of the infrastructure cannot be enabled through mechanisms and protocols which are standard in the realm of datacenter-level deployments. As such, IoT poses unique challenges including always-on reachability of IoT nodes, or at least suitable signaling, diagnostic and recovery mechanisms to cope with connectivity disruptions.

This chapter describes a rationale and some mechanisms in order to enable such functionalities when dealing with the unique requirements and challenges of IoT environments, e.g., embedded boards and other constrained devices. In particular, network virtualization is addressed here on top of a Cloud platform whislt taking into account the limitations of the smart devices. Furthermore, the approach can at the same time be mapped onto an IaaS-focused IoT solution based on the I/Ocloud approach when integrated with containerization at the network edge (see Chapter 3).

The contribution here is thus three-fold: a Cloud-based framework for the setup of virtual networks among geographically distributed IoT nodes as well as Cloud-based instances (e.g., VMs) whichever the deployment scenario; description of a set of scenarios that can be enabled through network virtualization in IoT; a flexible and lightweight network virtualization solution for IoT based on universally available and minimal tools, according to the Unix philosophy of composability and modular design.

2.2 Background

2.2.1 Network Virtualization

The networking field has known undeniable progress thanks to Network Virtualization (NV) [51] techniques by introducing advanced and innovative functionalities. The NV approach aims at providing the ability to set up logical/virtual networks decoupled from the underlying network hardware and configurations. This approach was introduced through several virtualization techniques that are widely adopted nowadays such as, Virtual Local Area Networks (VLANs) [53] and Virtual Private Networks (VPNs) [54]. Mainly, these techniques were used to provide functional services (e.g., security, traffic engineering) [52].

Over the past decade, organizations have been adopting virtualization technologies at an accelerated rate considering the emergence of Cloud offerings. In a Cloud infrastructure, the NV approach abstracts networking services and connectivities that were previously delivered by dedicated hardware into logical virtual networks using virtual instances hosted by off-the-shelf hardware resources decoupled from the physical underlying networks. These virtual networking instances can operate just like the standard hardware solutions would. Instead of using dedicated equipment only to deliver standard layer 2/3 services such as, switching and routing, the NV techniques can also incorporate advanced virtualized layer 4-7 functionalities, such as load-balancing and firewalling. Such a software-defined approach provides a variety of benefits in solving typical challenges in today's datacenter-based deployments. Indeed, it grants a significant level of flexibility and scalability for both the service providers and users who can self provision their networks without modifying or configuring the underlying physical links/infrastructure. Therefore, customization of networking topologies has become more efficient by dint of the agility provided by software-based compute resources. Recently, the NV approach has known a significant boost thanks to the genuine decoupling of the control and forwarding planes supported by Software-Defined Networking (SDN) [55] and Network Functions Virtualization (NFV) [56]. These latter paradigms introduced advanced programmability capabilities with a meaningful

level of network resilience not achievable before.

2.2.2 The OpenStack Networking subsystem: Neutron

Neutron is an OpenStack project providing Networking-as-a-Service (NaaS) for devices, such as virtual Network Interface Cards (vNICs) used by other OpenStack services such as Nova[3]. Neutron exposes a set of APIs and provides several choices for datacenters administrators to manage the networks traffic. Indeed, rather than using only the default open source switching solutions provided by OpenStack (e.g., Linux bridge and Open vSwitch (OVS)), administrators can also use third-party developed plugins. Therefore, the Cloud users can design customized networking topologies with extended features. In a typical Cloud deployment, the Neutron subsystem provides:

- The capability to handle, in a flexible way, OpenStack networking objects, such as *ports*, *networks* and *subnets* that other OpenStack services can make use of through a set of APIs.
- The ability to instantiate scalable Cloud deployments by supporting a large number of isolated users. Cloud tenants can instantiate desired networking topologies using different agents, drivers, and plugins.

The OpenStack networking subsystem (i.e., Neutron) is meant to be a self-service system, i.e., a Cloud user can configure and customize his/her networking topologies without involving or opening a support ticket from the Cloud administrator. For instance, Neutron can deal with the provisioning of the infrastructure by keeping users' traffic isolated even when they use the same IP addressing ranges (i.e., overlapping addresses).

In Neutron terminology, a set of terms has been introduced such as *networks*, *subnets* and *ports*. To have a complete grasp of the integration and interactions between IoTronic and Neutron, understanding these (Neutron) concepts is recommended for the rest of this chapter.

[3]https://docs.openstack.org/nova/latest/

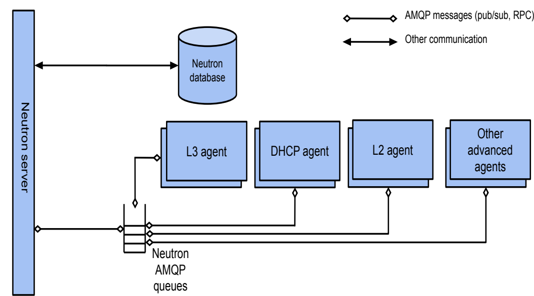


Figure 11: Architecture of Neutron subsystem.

To keep tenants' deployments isolated, Neutron uses the concept of *networks*. From the datacenter perspective, a *network* is similar to the VLAN concept. In particular, each *network* is associated with a user and has a unique identifier that can be recognized with. The *network* is totally under the control of the user who can handle it depending on his/her needs. A user can instantiate several *subnets* within his/her *network* and create customized topologies with particular networking policies. A *subnet* in simple terms, is a set of IP addresses associated with a particular configuration state. Last and not least, a *port* in OpenStack refers to the logical connection of a device (e.g., a vNIC) to a *subnet* (i.e., a virtual switch). It also describes the corresponding networking configuration of the device, such as its IP and Media Access Control (MAC) addresses.

The typical architecture of Neutron allows the support of core functionalities (e.g., switching) along with the ability to include extended ones (e.g., routing, load balancing) using extra agents. Figure 11 shows the internal components of Neutron. The subsystem consists of five elements, a RESTful server exposing a set of APIs to interact with the different Neutron services. A database that stores the metadata associated with the Neutron objects (e.g., *networks*, *ports*, and *subnets*). Then we have the L2 and L3 agents that afford the networking services. More specifically, the L2 agent deals with switching tasks within the same logical network while the L3 facilities provide (virtual) routing between different *subnets* or the external world (e.g., Internet). To make use of other advanced services, such as load balancing and firewalling, a user can deploy extra dedicated agents. We mention here that all the different Neutron components communicate using Advanced Message Queuing Protocol (AMQP).

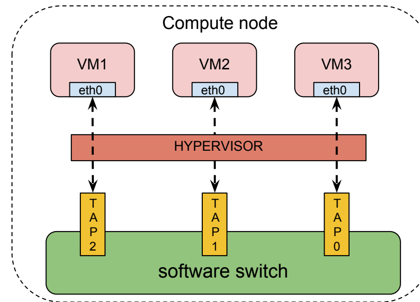


Figure 12: Nova compute node/binding-host design.

In a native Neutron implementation (see Figure 12), the L2 agent runs on physical servers where VMs are provisioned. These servers also host the virtual switching facilities on which the Neutron *ports* get instantiated (see Figure 12). In Neutron terminology, we refer to these machines (i.e., on which *ports* get created) as binding-hosts. Technically, a Neutron *port* is a TAP device/interface. In this architecture, the L2 agent manages the software bridges to provide multi-tenancy capabilities by keeping the users' *networks* isolated. The L2 agent keeps watching for new devices (i.e., TAP-class interfaces) and once a new one gets instantiated, the L2 agent queries Neutron for metadata associated with it (e.g., the *subnet*/*network* it belongs to). Based on the information the L2 agent receives, it attaches the TAP interface to the software bridge in charge of the virtual network involved. Afterwards, the hypervisor connects the TAP device to the VM using a Virtual Ethernet (VETH) pair as the VM and the virtual switch are hosted on the same physical machine (i.e., the binding-host, see Figure 12).

In a Cloud environment, since the infrastructure is being shared among several tenants, isolating their workloads is fundamental. To keep users' traffic segregated from a networking perspective, several overlay technologies, specified in terms of OpenStack options as type drivers, can be used. For instance, VLAN, Generic Routing Encapsulation (GRE) [57], and Virtual Extensible Local Area Network (VXLAN) [58]. These technologies keep users isolated by associating for each *network* (i.e., user) a unique identifier to be recognized with. The unique identifiers are stored in the Neutron database and shared among all the physical hosts (i.e., compute nodes) where VMs are provisioned. Therefore, packets switching among VMs, belonging to the same

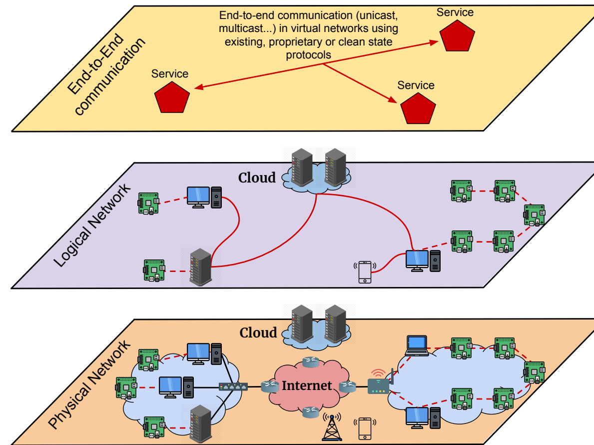


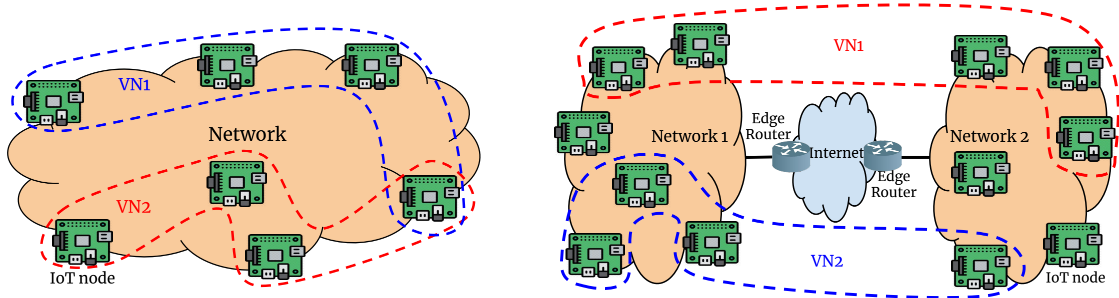
Figure 13: The MENO end-to-end communication concept.

network and *subnet*, but running on different compute nodes can be achieved by encapsulating Ethernet frames within packets that contain the *network* identifier in their header.

For layer 3 capabilities, such as virtual routing and floating IPs, an L3 agent (located in OpenStack network node) is required. Such nodes can host Dynamic Host Configuration Protocol (DHCP) agents as well. A DHCP agent is used in particular deployments where tenants/users delegate to Neutron the liability to take care of the IP stack of the VMs (i.e., configuring dynamic IP addresses).

2.3 Network Virtualization in IoT: use cases

In several IoT deployments, application developers do not need to expose or store the data generated by the IoT nodes in a server with a public IP address. They can, for example, store the data internally or process it before exposing it publicly. In such a case, only a limited number of IoT nodes, machines (either physical or virtual), and personal devices (e.g., smartphones, tablets, and wearables) need to cooperate for the sake of providing a specific service. The concept of such deployments where different smart objects can communicate in an end-to-end fashion was discussed in [59] under the name of "Managed Ecosystem of Networked Objects" (MENO) (see Figure 13). The MENO concept highlights the importance of network virtualization techniques in enabling scenarios where applications running on top of a virtual network see only



(a) Segregating IoT nodes within the same physical network into two distinct virtual networks.

(b) Two virtual networks accommodating IoT nodes from two distant physical networks within the same overlays.

Figure 14: Regrouping IoT nodes in different virtual networks.

the logical layer.

To accommodate geographically distributed IoT nodes within the same virtual network, virtual links connecting them should be created on top of underlying layer 2/3 physical networks. Therefore, IoT nodes belonging to the same overlay can reach each other even when they are not part of the same physical network (e.g., Local Area Network (LAN)) regardless of their networking configurations. The NV approach is extremely important in the IoT landscape to overcome reachability issues considering that neither a public IPv4 addressing scheme can be used given the proliferation in terms of the number of IoT devices, nor the IPv6 one which is not a suitable solution under all circumstances. In the following, we describe a number of generic use cases achievable by adopting NV techniques in IoT.

2.3.1 Partitioning IoT nodes within a same LAN

The most straightforward application the NV approach may enable in an IoT context is the partitioning of IoT nodes within a physical network into two or more virtual networks. Figure 14a depicts two virtual networks within each of them a subset of the available IoT nodes are grouped. This scenario can be considered, for example, in buildings management where virtual networks belonging to different administrators, owners, departments, etc., should be deployed. In such a situation where the IoT

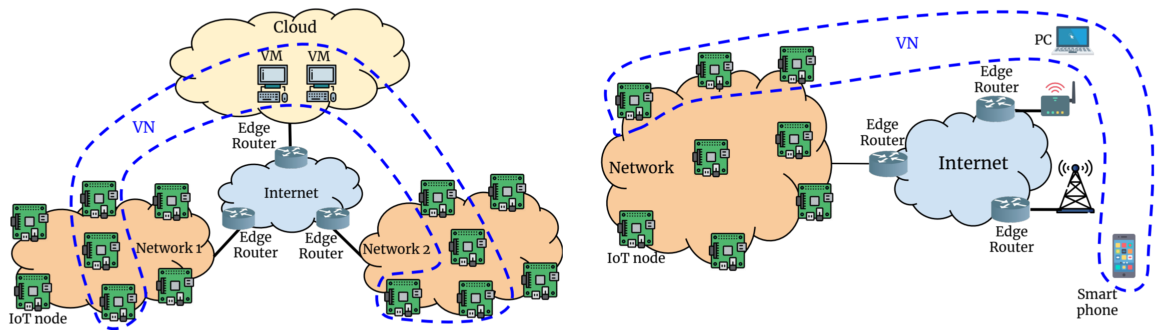
infrastructure may be used by multiple collaborators, we may need a flexible solution to segregate the networks based on their users or services they may provide. To deploy an application, the developer may arrange the IoT nodes within a customized topology to accommodate the application requirements. We can also mention the smart home use cases. Indeed, homes are increasingly becoming equipped with sensor nodes (e.g., HVAC, fire detection, cameras, burglar detection...). By integrating all the sensors and the actuators into a coherent ecosystem, they can afford a self-reliant system able to react, automatically, to specific events (e.g., sending notifications or actuation commands).

2.3.2 Regrouping IoT nodes from different networks

This use case may be required to connect IoT nodes belonging to two or more geographically distributed physical networks. In Figure 14b, two different virtual networks grouping multiple IoT nodes belonging to different (distant) physical networks are depicted. In this case, IoT nodes within an overlay seem to be, from a networking perspective, sharing the same physical network regardless of their locations and underlying networking configurations (e.g., being behind NATs). Such a scenario can be realized, for example, by creating L3-based VPNs between the edge routers. This use case can be envisaged, for instance, when IoT nodes from a physical network should act upon measurements or data collected by other IoT nodes belonging to a different physical network.

2.3.3 Extending an IoT network with powerful machines

In many cases, it is required to have one or more machines with enough computational/storage capabilities to deal with data management or perform calculations beyond the capabilities of the IoT nodes. Figure 15a depicts the case when the machines are located in a Cloud datacenter. The virtual network, in this scenario, accommodates within the same overlay not only scattered IoT nodes, but machines running on the Cloud as well. Such a deployment can be considered, for example, to



(a) Regrouping IoT nodes from different physical networks with Cloud-based instances within the same overlay.

(b) A virtual network composed of IoT nodes and mobile personal devices (e.g., smart phones).

Figure 15: Overlays extending IoT networks with Cloud-based VMs and personal devices.

store data gathered from the IoT nodes or to perform advanced data processing in the Cloud. This virtualization capability exempts the user from assigning to the Cloud machine a public IP address. A VM on the Cloud can collect data generated by a set of IoT nodes and use it afterwards to provide higher-level applications/services.

2.3.4 Extending an IoT network with personal devices

This use case presents a scenario where a number of smart personal devices (e.g., smartphones, PCs, tablets, watches, connected cars) can be part of an IoT network. Figure 15b depicts a virtual network regrouping a set of IoT nodes together with personal devices.

Obviously, all of the above scenarios may be combined when needed to fit the users demands. In Figure 16 a hybrid virtual network is shown. The virtual network is created by combining partitions from two separate networks composed of several IoT nodes while extending it with Cloud-based VMs and personal devices as well (e.g., a smart phone in this case). It is clear that the possible configurations are countless and the networking topology will strongly depend on the use case to be implemented.

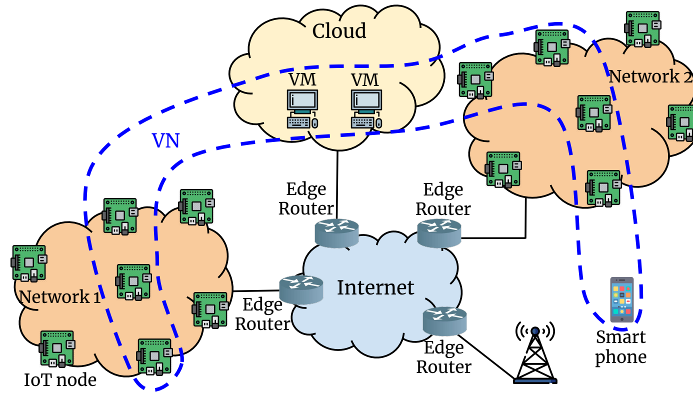


Figure 16: A hybrid overlay composed of IoT nodes, Cloud-based VMs and a personal device.

2.4 Stack4Things and Neutron integration design

In this section, we report a detailed description of our NV approach for IoT deployments. In particular, we highlight, from a technical implementation perspective, the differences between a standard Cloud-based Neutron deployment and our IoT-oriented solution.

2.4.1 Integration scenario

Instead of provisioning networking services only for instances hosted on the Cloud (e.g., VMs), our view aims at extending the NV usage scope by enabling its capabilities in the IoT landscape as well. The approach we are introducing provides NV services through REST interactions with the IoTronic subsystem presented in Chapter 1.

In a typical native OpenStack deployment, Neutron provides networking services for VMs running (locally) on the same hosts where the networking facilities (i.e., software switches) are hosted: a VM is attached to a virtual switch using a VETH pair (see Subsection 2.2.2). On the other side of the spectrum, considering the kind of IoT scenarios we are targeting, there is a significant difference compared to Cloud-based deployments as the switching facilities and instances (i.e., IoT nodes) can not be co-located on the Cloud: due to the nature of IoT deployments, IoT nodes can not obviously be deployed inside datacenters where the virtual switches are running;

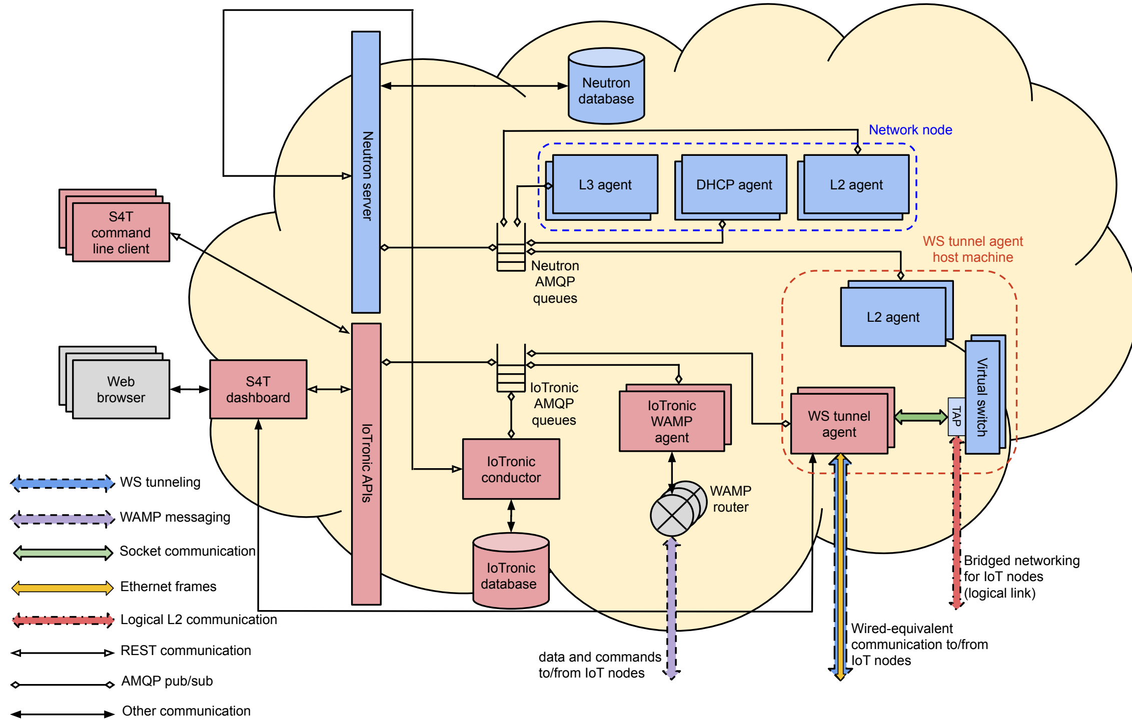


Figure 17: The Cloud-side architecture of the Stack4Things network virtualization system.

accordingly, we can not use (virtual) Ethernet pairs as the case for VMs. In our approach, the binding-hosts (where Neutron *ports* get created) are deployed inside the datacenter precisely, we chose the machines hosting the WS tunnel agents as binding-hosts (see red dashed rectangle in Figure 17). This choice was made on purpose as the WS agents can set up WS tunnels in order to attach Neutron *ports* instantiated on those machines to the IoT nodes deployed at the network edge using WebSocket. In simple terms, *ports* are created/managed on the Cloud then attached to the IoT nodes deployed at the network edge using WebSocket in a decoupled two-step pattern.

The system design takes into duly consideration the typical resource constraints of IoT environments. In fact, the architecture used makes the IoT nodes not involved in most of the NV duties by keeping them totally unaware of Neutron involvement. Besides, since L2 agents and switching facilities are running on the Cloud, our approach provides availability for mission-critical Neutron services and scalability for particular hefty configuration requirements. As shown in Figure 18 that represents the S4T node-side architecture, attaching an IoT node to a virtual user-defined network

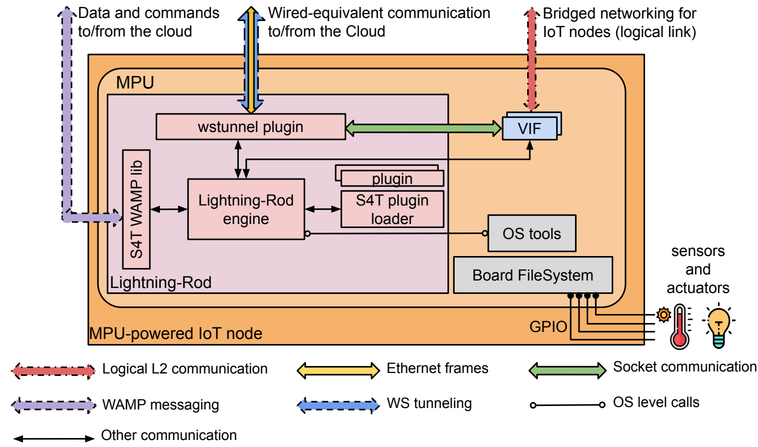


Figure 18: The node-side architecture of the Stack4Things network virtualization system.

translates into a relatively straightforward process, i.e., a Virtual InterFace (VIF) gets instantiated on the IoT node and then attached (using a reverse WebSocket tunnel) to the OpenStack networking platform managed by Neutron on the Cloud.

We focus, in the following, on a deployment involving the use of the Neutron Modular Layer 2 (ML2) plugin with Linux bridge and VXLAN as mechanism driver and type driver, respectively. Deployments using other mechanism drivers (e.g., OVS, Cisco Nexus 1000v) or type drivers (e.g., GRE), can be achieved as well, if needed.

In Figure 19 a sketch is modeled after the low-level reverse tunnel, yet focused on the creation of a switching platform using a virtual switch (i.e., a Linux bridge) between two IoT nodes controlled by the same WS tunnel agent. Still sticking to the setup of a control WS based on WAMP, as a preliminary step in this workflow, in this case, a reverse WS tunnel (rtunnel) gets activated for each IoT node to be virtually bridged. As a simplified scenario, the figure depicts only two such nodes, yet no limitation is in place regarding the number of remote nodes to be virtualized in terms of networking. As any IoT node in our approach, from now on referred to as a client, needs to go through the same set of operations, we will describe the flow just for a single instance for the sake of brevity.

Taking into consideration the uppermost node (i.e., a single-board computer) in the diagram (see Figure 19), an initial step lies in setting up a TCP connection based

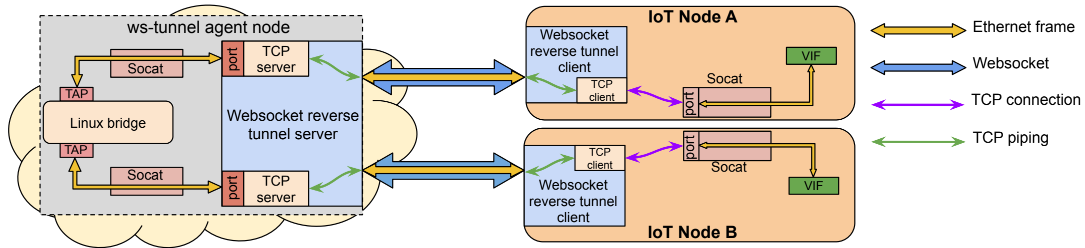


Figure 19: Low-level functional diagram of bridged tunneling over WS (IoT nodes managed by the same WS tunnel agent).

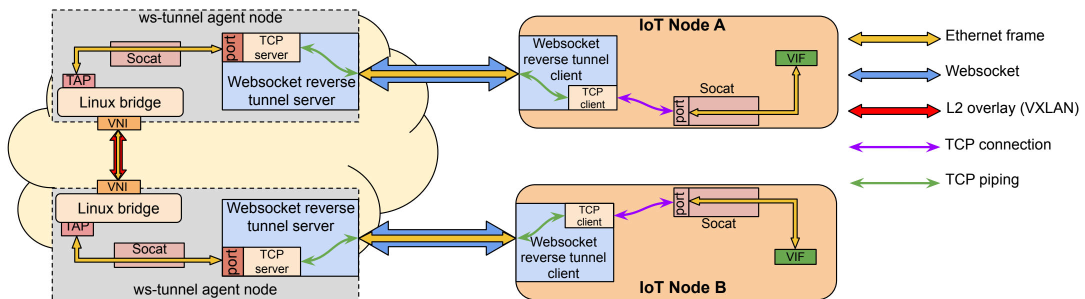


Figure 20: Low-level functional diagram of bridged tunneling over WS (IoT nodes managed by different WS tunnel agent).

on a WS rtunnel, which consists of exposing, on the server-side, a listening socket on a local port, as soon as the rtunnel server accepts a request for a new rtunnel. The TCP connection just established gets piped to the rtunnel that encapsulates TCP segments in a WS-based stream. On the WS rtunnel client-side, as soon as the rtunnel is established, a new TCP client is brought up connecting to a local (`Socat`-provided listening) port, and such TCP connection gets piped to the rtunnel. By employing the executable called `Socat`, which operates in listening mode on both sides of the chain and, on connection, starts exposing a virtual TAP device (depicted in Figure 19 as VIF for the boards) on either side, then flows to/from these TAP interfaces get piped through `Socat` to the rtunnel. Regarding virtual devices setup and binding, `Socat` is considered a networking "swiss army knife" tool for Unix-like systems. Alike its counterpart, Netcat, the more full-featured `Socat` offers a set of networking features and quick shortcuts, such as socket piping/tuning, setup of virtual TUN/TAP devices, and process control as well. The above-reported steps are time-insensitive logical operations that can easily tolerate network unavailability. When a user requests to attach an IoT node to a specific network through the dashboard or

command line, IoTronic manages the process by:

i) requesting the creation of a Neutron *port* (TAP-class device) on the machine hosting the WS tunnel agent in charge of managing the involved IoT node.

ii) establishing a (reverse) WS tunnel between the WS tunnel agent and the IoT node.

iii) creating a VIF (i.e., a TAP class device) on the IoT node then piping connections from the TAPs on both sides to the WS tunnel already created.

iv) assigning the IP and MAC addresses specified by Neutron to the VIF instantiated previously in the IoT node.

Figure 19 describes a scenario with two IoT nodes managed by the same WS tunnel agent and belonging to the same virtual/overlay network. As already mentioned, when attaching an IoT node to a network, IoTronic manages the process. Firstly, the IoTronic conductor interacts with the Neutron server to create a *port* on the machine where the WS tunnel agent in charge of the IoT node is running. Afterwards, the L2 agent hosted on that machine handles this event (i.e., a new TAP class device is created) by attaching it to the appropriate Linux bridge managing the traffic of the concerned logical network. IoTronic then pipes the connections to the tunnel client/server, respectively, and sets up the (reverse) WebSocket tunnel. In order to attach another IoT node to the same logical network, an identical workflow is triggered. As a result, another WS tunnel is then created, and the connection piped to the same Linux bridge as the two IoT nodes are expected to be on the same logical network when the setup is over (see Figure 19). From a security perspective, a deployment of this kind is secure as, for each instance, a WS tunnel is established between the IoT node and the Cloud; thus, tunnels can be encrypted to ensure confidentiality, if needed, such as when traversing public networks.

The reachability among the distributed IoT nodes belonging to the same virtual network but managed by different WS tunnel agents is ensured by default. Indeed,

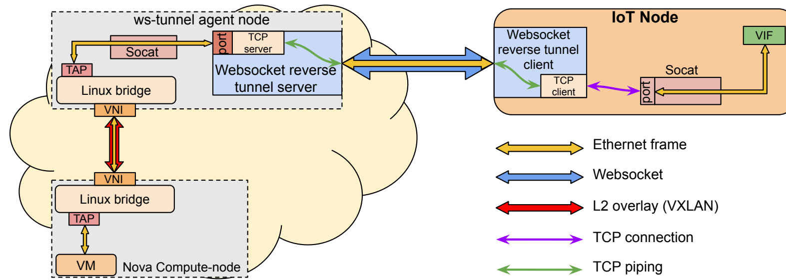


Figure 21: Low-level functional diagram of bridged interconection between an IoT node and a Nova VM.

as discussed previously, Neutron uses unique identifiers shared among all the binding-hosts (i.e., machines hosting the WS agents in our case) to segregate users traffic and enable transparent packets transport among them. Figure 20 shows the case when the two IoT nodes are managed by different WS tunnel agents. In this case, an overlay network is established by Neutron. In addition, reaching OpenStack instances (e.g., Nova VMs) from IoT nodes is also insured as the *networks'* unique identifiers are shared between all the binding-hosts including Nova compute nodes (see Figure 21). Communications among all the binding-hosts (i.e., compute nodes and WS tunnel agents machines) are based on overlay technologies, such as VXLAN and GRE. In Figures 20 and 21 depicting the datapaths, we choose VXLAN and its corresponding interfaces, represented as VXLAN Network Identifiers (VNIs).

2.4.2 Advanced functionalities of the integration

Complex and more advanced networking topologies involving IoT nodes over a wide-area network are feasible in this light. Switching L2 traffic between IoT nodes deployed at the network edge is not the only use case: advanced services for node-hosted overlays may include, for example, routing, firewalling, load balancing as well as advanced software-defined and Virtual Network Functions (VNFs). For layer 3 features, the L3 agent leverages the Linux IP stack and iptables. Besides, the agent uses Linux network namespaces to provide isolated forwarding contexts; thus multiple routers with potentially overlapping IP addresses are supported. Similarly to DHCP servers that exist for each virtual network managed via Neutron and controlled by its DHCP agent, every router lives in its own namespace.

#	Method	URL	Semantics	Parameters	Return type
1	***PUT***	/v1/boards/{board_uuid or name}/ports	Create and attach a port to a specific board	Network_uuid (uuid) Subnet_uuid (uuid)	Port
2	***GET***	/v1/ports	Retrieve a list of ports	–	PortCollection
3	***GET***	/v1/boards/{board_uuid or name}/ports	Retrieve a list of ports attached to a board	Board_uuid (uuid) or name (string)	PortCollection
4	***GET***	/v1/ports/{port_uuid}	Retrieve details about a port	Port_uuid (uuid)	Port
5	***GET***	/v1/boards/{board_uuid or name}/ports/{port_uuid}	Retrieve details about a port attached to a board	Board_uuid (uuid) or name (string) Port_uuid (uuid)	Port
6	***DELETE***	/v1/boards/{board_uuid or name}/ports/{port_uuid}	Detach a port from a board	Board_uuid (uuid) or name (string) Port_uuid (uuid)	–
7	***DELETE***	/v1/ports/{port_uuid}	Delete a port (and detach it)	Port_uuid (uuid)	–

Table 1: The IoTronic RESTful networking APIs.

2.5 The Stack4Things virtual networking solution

This section gives details about our virtual networking solution from the development point of view. We first report the RESTful APIs exposed by the system, then a detailed description of the functional workflow when attaching an IoT node to an OpenStack defined virtual network is reported. We do not report other workflows due to space constraints. At the end of the section, we highlight the advantages of our IoT-focused NV approach.

2.5.1 Stack4Things networking APIs

To expose the networking capabilities of S4T, a set of RESTful APIs were built[4]. The choice of REST is in line with the design of OpenStack services APIs[5]; thus we have a seamless integration of our system with other OpenStack services. We report in Table 1 the S4T networking APIs built and tested. The table reports the exploited Hypertext Transfer Protocol (HTTP) methods, Uniform Resource Locators (URLs), semantics, input parameters, and return types. A Create, Read, Update, and Delete (CRUD) architectural design is used to build the APIs. Regarding the authentication, we delegate this aspect to the OpenStack identity service, Keystone, as is the case for all the other OpenStack services. In particular, Keystone generates authentication tokens that grant access to our networking REST APIs. Clients then

[4] A Swagger APIs description: https://mdslab.github.io/iotronic-api/swagger-ui/#/
[5] OpenStack API documentation: https://docs.openstack.org/api-quick-start/

obtain the token as well as the URL endpoint of the IoTronic service API by supplying their valid credentials to the authentication service. Whenever an S4T client makes a REST API request, the authentication token is included in the X-Auth-Token request header. Regarding the return type of the APIs, we used JavaScript Object Notation (JSON). In particular, a Port JSON data type is provided in response to requests #1, #4, and #5. While a Portcollection JSON response (i.e., a list) is due for requests #2 and #3.

2.5.2 Workflow of attaching a board to a virtual network

In this subsection, we report a detailed workflow description when attaching an IoT node to a user defined virtual network. This use case corresponds to request #1 in Table 1. As a use case prerequisite, we assume that the IoT node is already registered to the Cloud. The following operations are then performed (see Figure 22):

1. The user sends a request using either the dashboard or the CLI in order to attach an IoT node to a specific virtual network. The request contains in its body the Universally Unique Identifiers (UUIDs) of the virtual network (*network* and *subnet*) the board has to be attached to.

2. The dashboard/CLI performs the correspondant IoTronic REST API call (i.e., request #1 in Table 1).

3. The IoTronic API server sends a RPC (using RabbitMQ) to the IoTronic conductor in order to create a new Neutron *port* and attach it to the IoT node.

4. The IoTronic conductor receives the RPC by extracting it from the IoTronic AMQP queue and sends a query to the IoTronic database. In particular, it checks if the IoT node involved is already registered to the Cloud and decides the WS tunnel agent to which the IoT node has to be connected to.

5. The IoTronic conductor performs a REST call to the Neutron server to create, on its database, an entry about a new *port* having as binding-host the machine where the WS tunnel agent in charge of the IoT node is running.

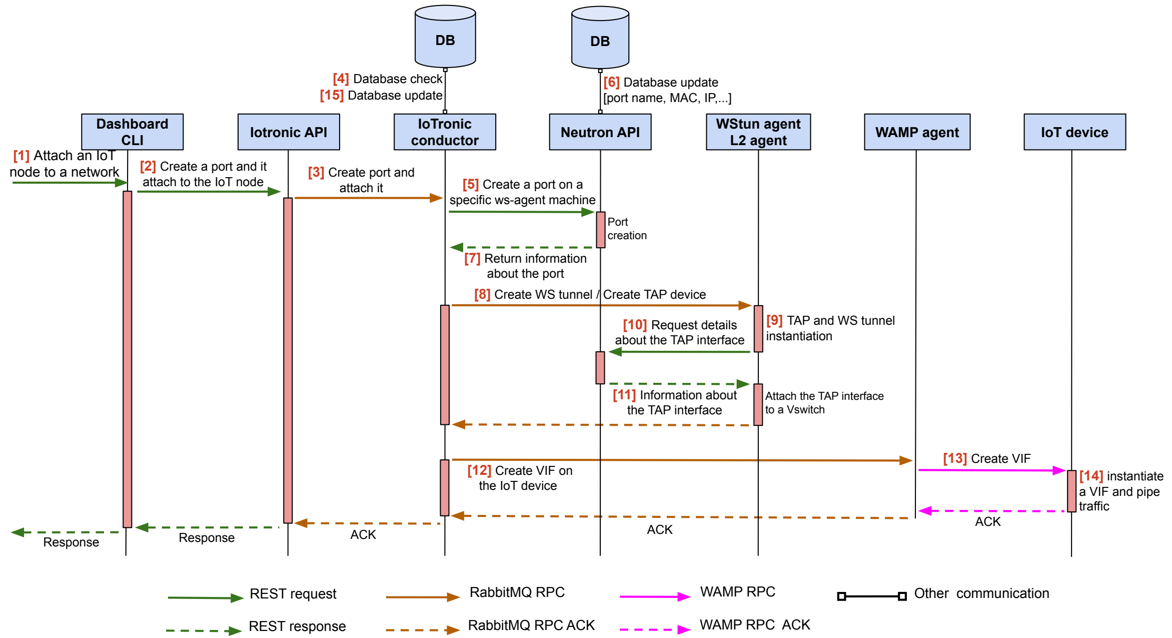


Figure 22: The workflow of attaching an edge IoT node to a specific logical network.

6. The Neutron server interacts with its database to create the *port* and associate to it a UUID. Besides, Neutron allocates for the new port both the MAC and IP addresses. Afterwards, the Neutron server sends a notification about the port, via RPC, to the DHCP agent to update the lease file.

7. The S4T IoTronic conductor receives the response for the REST call about the Neutron *port* creation. Then it (i.e., the conductor) generates a random port number that will be used subsequently for the `Socat` tunneling.

8. The IoTronic conductor sends an RPC (using RabbitMQ) to the WS tunnel agent in charge of the IoT node in order to create and configure, using `Socat`, a TAP class device associated with the Neutron *port* created in step 5 (the UUID of the *port* is sent as a parameter in the RPC to assign for the TAP device a name using it).

9. The machine where the WS tunnel agent involved (i.e., in charge of the IoT node) is hosted receives the RPC and then follows up with its own set of server-side network virtualization duties. Specifically, it creates and configures a TAP interface corresponding to the *port* already created by Neutron (the name of the interface is based on the port UUID) and pipes the WS tunnel to it. When the

L2 agent running on the machine hosting the WS tunnel agent (i.e., where the TAP interface got created) detects the new TAP interface, it triggers the next step.

10. The L2 agent requests from Neutron the details about the new TAP interface, such as the *network* and the *subnet* it belongs to.

11. The L2 agent receives the Neutron server response and then attaches the TAP interface to the appropriate Linux bridge managing the virtual network concerned when using the Linux bridge L2 agent, or it (i.e., the L2 agent) matches the interface with the proper VLAN ID when OVS is in use.

12. The IoTronic conductor sends a RabbitMQ-based RPC to the WAMP agent managing the board in order to instantiate a VIF on the IoT node.

13. The WAMP agent to which the IoT node is registered receives the RPC from the IoTronic RabbitMQ queue. The agent then forwards a WAMP-based RPC to the IoT node.

14. Through the WAMP library, the LR engine receives the RPC from the WAMP router. The LR engine then invokes (e.g., sets as running, passing needed parameters) the WS tunnel plugin that sets up a (reverse) WS tunnel with the agent specified by the conductor (i.e., the Cloud-based host where the TAP interface was created). The engine afterwards creates a VIF using `Socat` and pipes traffic to/from it to the WS tunnel.

15. The IoTronic conductor performs a query to its database to store the new details about the IoT node (e.g., *network* and *subnet* associated with the IoT node).

2.5.3 Creating Virtual Networks using Stack4Things

The S4T network virtualization approach enables users to create and manage overlays composed of IoT nodes deployed at the network edge together with Cloud-based instances. The solution is efficient and can enable all the scenarios mentioned previously in Section 2.3. In particular, the approach provides:

Aspects	S4T overlay	VPN-based overlay	Legacy apps (e.g., AllJoyn)
Security	TLS-based communications (using secure WebSocket) between the devices and the Cloud	Encryption between clients and VPN server possible (e.g., IPsec)	Depends on the protocols/platforms used [60] [61]
Ability to span distant networks	Supported	Supported	Not supported [62]
Interoperability with personal devices	Possible (requires only a minimal Linux-based environment on the devices)	Possible	Strictly linked to the platform/technologies used
Multi tenancy	Supported	Could be supported but hard to manage (management of the association users/VPNs)	Not supported

Table 2: Comparison between different IoT network virtualization approaches.

- **A seamless integration with OpenStack:** as discussed before, the introduced NV approach is fully compatible with OpenStack, in particular, its networking subsystem, Neutron. Therefore, tenants' traffic segregation can be considered a strong built-in capability S4T makes use of. Users can use the IoT NV approach the same way as in a native Cloud deployment without worrying about their networks isolation. Besides, Neutron provides a set of advanced networking functionalities (e.g., load balancing) that may enable value-added services in an IoT context. On the other hand, using a unified system to manage virtual networks accomodating distributed IoT nodes together with Cloud-based instances (e.g., VMs, containers) within the same overlay networks enables the two environments to collaborate.

- **Zero-configuration networking:** the NV solution we introduced is able to group IoT nodes belonging to different distant physical networks within the same broadcast domains. It is worth mentioning here that the approach does not require any specific configuration on the edge routers or networking equipment. In particular, we do not use any layer 2/3-based VPNs over the Internet; hence avoiding manual configurations that may inflict scalability issues (e.g., when multiple sites/networks should be connected) or involving the Internet Service Provider (ISP). In this context, the approach overcomes the limitations of other legacy applications that provide network virtualization functionalities in the IoT landscape, such as the AllJoyn platform developed originally by the AllSeen Alliance and now merged with IoTivity under the Open Connectivity

Foundation (OCF)[6]. In particular, the AllJoyn framework comprises a DBus-derived application protocol useful for messaging, advertisement, and discovery of services, working via selected mechanisms on available transports. A limitation of the protocol is being currently designed to work only as long as the communicating devices are on the same broadcast domain [62]. Therefore, it is unable to cross boundaries imposed by the network layer protocol. A single subnet might be sufficient for small-scale deployments but not for buildings and smart cities applications (see Table 2).

- **Mobility support:** being an agnostic approach vis-à-vis the physical networking configurations of the IoT nodes and edge routers makes the solution stable and does not get impacted by users/nodes mobility. Even with a mobile IoT node experiencing a vertical handover (e.g., switching from 3/4G to WiFi) or a connectivity interruption and, thus, an underlay IP address modification, our network virtualization solution for creating overlays does not get affected by such networking reconfigurations. Indeed, the overlay IP address assigned by Neutron to the IoT node will always remain the same since it is not associated with the underlay networking configuration. During a connectivity disruption, the LR daemon will keep trying to establish a connection to the Cloud. Once the Internet connectivity is reestablished, the VIF on the IoT node (created when attaching the IoT node to the overlay for the first time) gets attached (using a WebSocket tunnel) to the same Neutron *port* on the Cloud as before the connectivity interruption. We mention that the Cloud *port* has been already configured with an overlay IP and MAC addresses when attaching the IoT node to the overlay for the first time (before the connectivity interruption). In other terms, the association we have, in this case, is between the Neutron *port* (which is created on the Cloud and does not get impacted by the mobility) and the overlay IP address. Therefore, even if the underlay networking configuration of an IoT node changes over time, the same overlay IP address will be assigned to the VIF on the IoT node as the Neutron *port* remained intact.

[6]https://openconnectivity.org/allseen-alliance-merges-open-connectivity-foundation-accelerate-internet-things/

- **Extended IoT networks:** the network virtualization approach we introduced can also integrate personal devices such as, smartphones, tablets, PCs, etc., within the users defined overlays. Indeed, the device-side agent (i.e., LR) that runs on the IoT nodes typically requires only a minimal Linux-based operating system with Python and Node.js as runtime environments, or even just a JavaScript-enabled browser, for the most constrained environments, e.g., mobiles. We mention here that we have an app-based, browser engine-powered LR agent designed for Android-based devices. Generally speaking, the stability of the overlays discussed previously, even during devices mobility phases, may enable interesting value-added services when combined with personal devices.

- **Efficient applications deployment:** by using the plugins injection capability provided by S4T together with the network virtualization capability, developers can have an efficient platform for applications deployements. Using the network virtualization solution, a developer can accommodate IoT nodes to meet the application's networking topology requirements and then he/she can program the business logic running on those IoT nodes. In fact, S4T enables the (remote) programmability, even at runtime, of IoT nodes. A developer can upload, to the Cloud, his/her code defining the business logic and inject it in an IoT node (or group of nodes). As runtime environments, S4T provides the choice between Python and Node.js, as these both provide suitable execution environments and higher-level programming facilities with an extensive selection of libraries and frameworks.

2.6 Experimental results

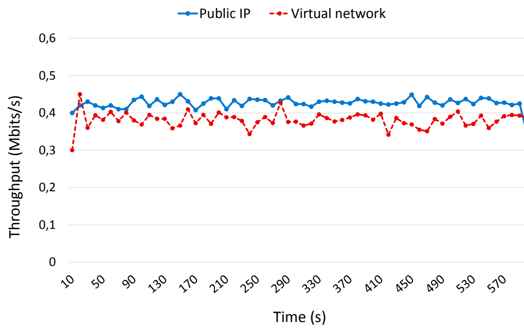
An evaluation of the proposed system is provided in the following, highlighting specific key performance metrics namely throughput and latency, according to the aforementioned virtual networking mechanisms established using a wrapped `Socat`-based tunneling. We conducted a set of experiments based on the Flent [63] tool for measuring throughput using Transmission Control Protocol (TCP) flows and Internet Control Message Protocol (ICMP) echo requests to gauge latency (i.e., Round-Trip

Time (RTT)). The test setup is based on a server collecting data and presenting statistics and a client generating TCP traffic and ICMP requests. When running multiple tests in parallel, the output of each test is parsed, and the output is stored in a common JSON-based format for further processing.

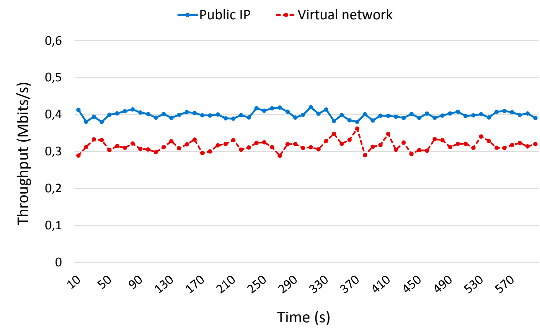
To assess the performance, we compared the results obtained by our NV approach with the ones from a native connection without involving OpenStack (i.e., public IP-based). Each experiment conducted in this part has a total duration of 10 minutes. During such time intervals, we simultaneously measured throughput and latency variation over time. The S4T Cloud infrastructure is deployed at the Department of Engineering of the University of Messina, Italy. The IoT node, instead, was configured and deployed outside the university network to get realistic results over a WAN interconnection. As an IoT node, we used an Orange Pi zero with ARMBIAN 5.60 and Linux kernel 3.4.113-sun8i. The board is powered by an Allwinner H2+ CPU (1200Mhz ARM Cortex -A7 Quad-core) and 512 Mbytes of RAM. On the Cloud side, we are implementing the WS tunnel agent on a virtual machine with 4 Gbytes of RAM and 2vCPUs (the host machine is powered by an Intel Core i7 9xx CPU 2.13GHz).

Figure 23a depicts the TCP throughput variation while uploading from the edge node to a Cloud instance considering two scenarios. In the first one, the Cloud instance was deployed using a public IP address. In that case, the communication is restricted only between the Cloud instance and the board without providing any further interactions with other IoT nodes (i.e., the switching platform is not involved). In the second one, the board has been attached to a virtual network created using OpenStack; hence the board is considered as a Cloud instance that can interact with VMs and/or other IoT nodes on the same logical network.

It may be noticed from Figure 23b that the throughput when the IoT node is downloading from a Cloud instance using the virtual network decreases compared to the native network (see Table 3). Similarly, when the IoT node is uploading to the Cloud instance, we noticed a decrease of the throughput performance (see Figure 23a and Table 3). Regarding the system latency, the approach introduces an additional



(a) Throughput variation when the IoT node is uploading, using TCP, to the Cloud instance (averaged every 10 seconds for better visibility).



(b) Throughput variation when the IoT node is downloading, using TCP, from the Cloud instance (averaged every 10 seconds for better visibility).

Figure 23: Throughput variation when the edge node is uploading/receiving TCP traffic to/from a Cloud-based instance.

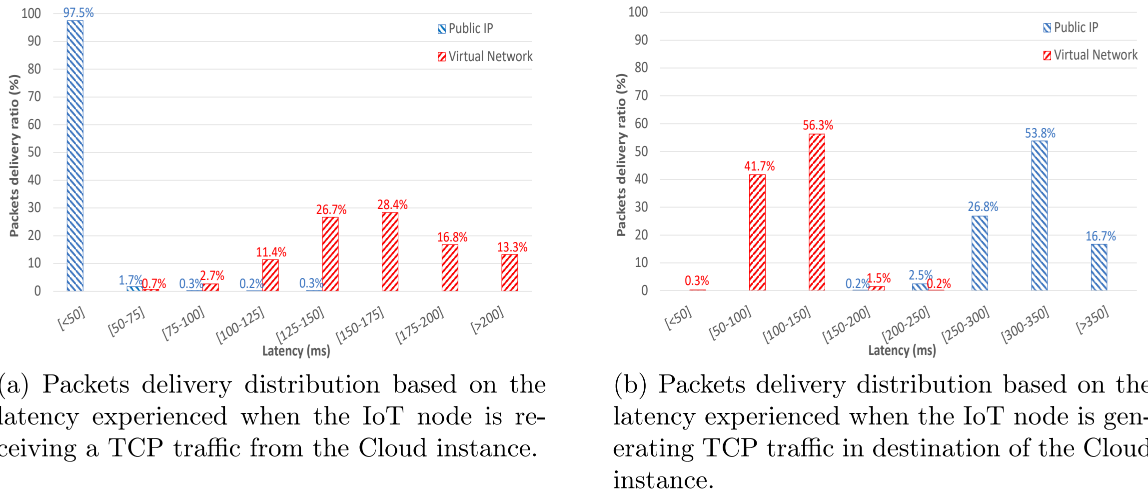
	Upload		***Download***	
Statistics	*S4T*	*Native*	*S4T*	*Native*
Min	0.16	0.25	0.02	0.27
Max	0.57	0.99	0.4	0.5
Mean	0.38	0.43	0.31	0.39
Stdev	0.06	0.07	0.05	0.04

Table 3: Statistical TCP results of throughput (Mbps) during an upload-/download from/to the IoT node.

	Upload		***Download***	
Statistics	*S4T*	*Native*	*S4T*	*Native*
Min	48.6	200	45.47	38.86
Max	205	414	323.5	148.1
Mean	103.91	317.88	160	40.49
Stdev	19.77	31.99	36	6.82

Table 4: Statistical TCP results of latency (ms) during an upload/download from/to the IoT node.

processing delay due to the different (TCP) traffic pipes used in the design. Figure 24a and Table 4 show in detail the effect experienced. Of course, such a design based on the Cloud is not for highly time-sensitive applications. For such particular latency constraints, S4T is being extended to provide Fog solutions [16]. Nevertheless, the S4T virtual network takes advantage of using WebSocket tunnels when traversing Firewalls that, in this case, do not do further packet inspection. As reported in Figure 24b and Table 4, our approach reduces the latency from the board to the Cloud while traversing the university firewall. Concretely, the average latency increase from 103.91 ms when using the virtual network to 317.88 ms when the native network is in use (see Table 4). In order to further investigate the cause of the throughput performance decrease, we performed additional tests using User Datagram Protocol (UDP) instead of TCP. Indeed, the use of UDP brings more flexibility and control over the testbed as the



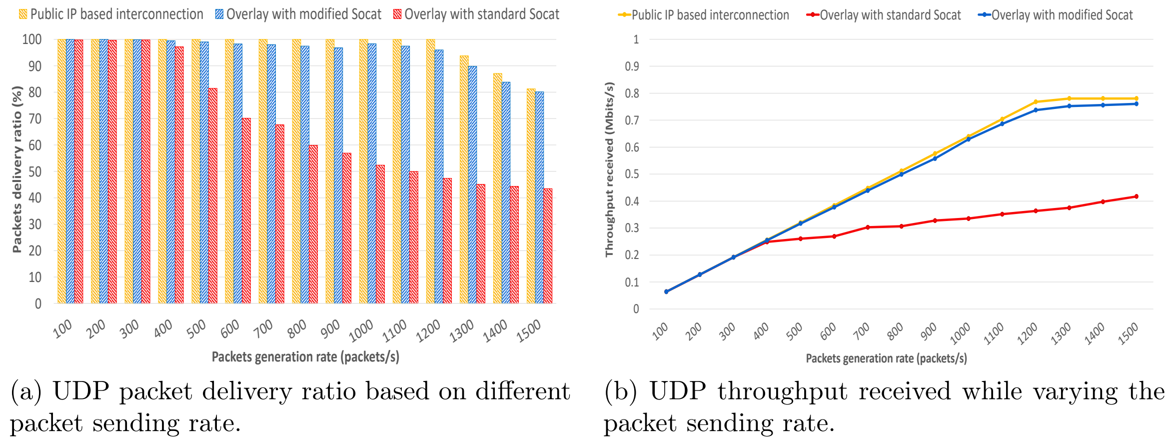
(a) Packets delivery distribution based on the latency experienced when the IoT node is receiving a TCP traffic from the Cloud instance.

(b) Packets delivery distribution based on the latency experienced when the IoT node is generating TCP traffic in destination of the Cloud instance.

Figure 24: Comparison of the latency experienced using a Public-IP based deployment and the S4T overlay.

packet sending rate can be controlled with higher granularity, whereas for TCP, the communication between the client and server is handled by the protocol itself that determines the suitable packet generation rate (i.e., based on the 3-way handshake and the windowing mechanism). UDP does not employ packet acknowledgment or retransmission mechanisms; therefore, the throughput does not get impacted by the protocol's functionalities. For this purpose, we deployed a Python-based UDP client (i.e., traffic source) and server (i.e., traffic destination/sink) on the IoT node and the Cloud instance (i.e., the VM), respectively. We used several sending rate values for the evaluation starting from 100 packets per second (pps) up to 1500 pps at 100 pps step increments, with 80 bytes of payload size. Each UDP experiment reported in this part has a duration of 5 minutes.

By using UDP and collecting statistics at the server-side, we noticed a considerable packet loss when using the `Socat`-based overlay (see red dashed rectangles in Figure 25a). In fact, increasing the packet sending rate affects negatively the percentage of the received packets. The packet loss was around 50% when we generated 1500 pps. This considerable loss impacted significantly the throughput received by the server, as depicted in Figure 25b. These results are mainly due to the `Socat`-dependent sustained packet processing rate. Actually, `Socat` starts discarding packets above a



(a) UDP packet delivery ratio based on different packet sending rate.

(b) UDP throughput received while varying the packet sending rate.

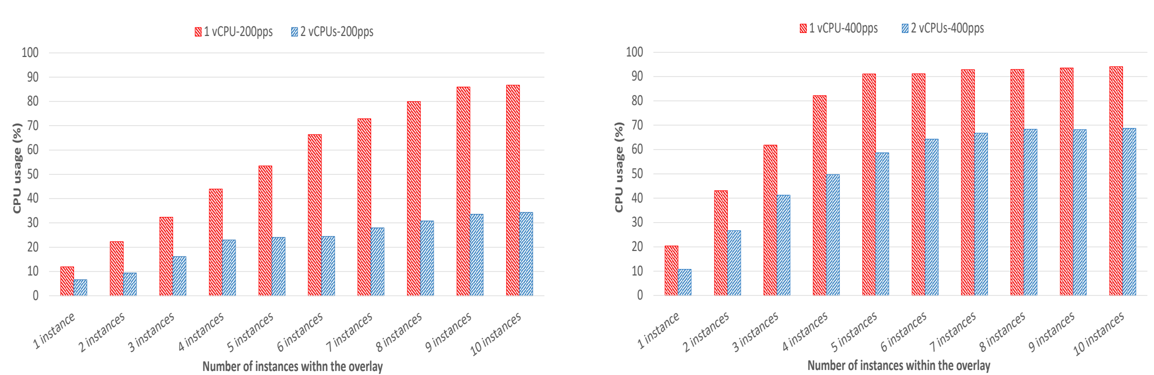
Figure 25: Comparison of the virtual networking performance with a UDP traffic using the two versions of Socat.

given threshold.

To evaluate the system scalability from the Cloud-side perspective, we deployed, on the same LAN as where S4T is running, a number of Docker containers running LR agents. By setting up the testbed within the same physical network, we cancel the impact of external network delays and packet loss: the whole testbed runs within the same LAN with links up to GB/s. We grouped all the containers on the same overlay on which an OpenStack VM is running too. An issue we experienced, in this case, is the packet loss even when the packet generation rate by each container is low. Precisely, when we generated 100 pps by one container (i.e., the IoT node) in the destination of the OpenStack VM, the packet loss was 0%. However, when we attached 15 containers to the overlay and made each of them generate 100 pps, we had an average of packet loss, on the Cloud, equal to 26%. This considerable amount of packet loss was induced by only 100 pps. By increasing the rate of packets generation, the packets ratio dropped significantly.

In order to deal with the aforementioned issues and make the system more performant and scalable, the C-based Socat tool was modified[7] to meet the requirements of our use case. Specifically, the `Socat` packet buffer size was increased to avoid packets overwriting, then the rest of the code was adapted accordingly. We report

[7]Modified Socat https://github.com/Zakaria-Ben/Socat



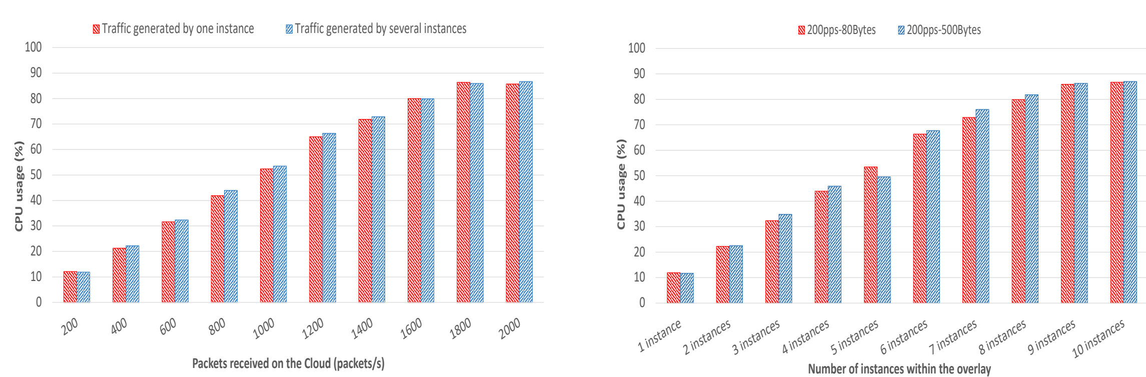
(a) Cloud CPU usage average of the WS tunnel server and Socat processes during 2 minutes with instances generating 200 pps in destination of the OpenStack VM.

(b) Cloud CPU usage average of the WS tunnel server and Socat processes during 2 minutes with instances generating 400 pps each in destination of the OpenStack VM.

Figure 26: CPU usage of the WS tunnel server and Socat when managing UDP packets with 80 bytes payload length sent by different instances.

in Figures 25a and 25b a performance comparison of the two `Socat` versions (i.e., the default and modified one). The gain in terms of throughput was significant by solving the buffer size issue that caused the considerable packet loss. For the rest of this experimental part, we present the system scalability from the Cloud-side CPU usage (i.e., WS tunnel server and `Socat`) point of view based on our modified version of `Socat`. We did not report the RAM usage as it was minor. The WS tunnel server uses around 3.1% from the available 4 GB of RAM, while each `Socat` instance uses exactly 0.1%.

As we are increasing the number of instances generating traffic in the destination of the OpenStack VM, we had an increase of the CPU usage of WS tunnel server and `Socat` as reported in Figures 26a and 26b. In particular, when the Cloud (i.e., the machine where the WS tunnel agent is hosted) is powered by one vCPU, and each instance generates 200 pps (see Figure 26a), the CPU usage of the two processes involved (i.e., WS tunnel server and `Socat`) reached around 90% with nine instances (a total of 1800 pps packet is being handled on the Cloud) while noticing a slight packet drop due to the CPU overload (other processes are running on the host, such as L2 agent and Linux bridge). We mention that the system, when overloaded, keeps functioning, yet some packet loss occurs; thus, leading to throughput degradation



(a) Comparison of CPU usage (WS tunnel server and Socat processes) when generating traffic by one and several instances.

(b) WS tunnel agent machine CPU usage (WS tunnel server and Socat processes) with different payload sizes.

Figure 27: CPU usage of the WS tunnel server and Socat processes varying traffic sources and the payload length.

and higher latency when using TCP (as the lost packets should be retransmitted). In general, the system performance is a tradeoff between the Quality of Service (QoS) required and the resources allocated on the Cloud. By increasing the number of CPUs allocated to the WS tunnel agent machine, we avoided the CPU overload and, therefore the packet loss (see Figure 26a). With instances generating more UDP packets, the CPU usage increased rapidly than the first case with 200 pps (Figure 26b). From those experiments, we show that the CPU usage of the Cloud depends on the number of packets received. Indeed, if an instance is attached to an overlay without generating packets, it does not impact the CPU usage of the Cloud (neither `Socat` nor the WS server consumes CPU resources).

To highlight more this aspect and outline the fact that the CPU usage depends on the number of packets received and not the number of instances within a virtual network, we compared the Cloud CPU usage when several instances generate a certain number of packets with one instance generating the same amount of packets. We report in Figure 27a the results when each instance generates 200 pps. As we can see from the graph, the Cloud CPU usage is transparent regarding the packets sources. Receiving a certain number of packets from one or several instances makes the Cloud behave in the same manner. The limitation of the system indeed lies in the number of packets it can manage (e.g., line rate) and not the number of instances within

	Packets with 80 bytes of payload CPU usage (%)			Packets with 500 bytes of payload CPU usage (%)		
Packets generation rate (pps)	*WS tunnel client*	*Socat*	*Total*	*WS tunnel client*	*Socat*	*Total*
100	3.26	0.74	4	3.42	0.74	4.16
200	6.05	1.34	7.39	6.06	1.35	7.41
300	8.98	1.97	10.95	8.98	2	10.98
400	12.17	2.67	14.84	11.62	2.62	14.24
500	14.72	3.26	17.98	14.57	3.18	17.75
600	17.1	3.84	20.94	16.98	3.79	20.77
700	20.34	4.54	24.88	19.67	4.2	23.87
800	22.61	5	27.61	22.52	4.97	27.49
900	24.54	5.51	30.05	24.62	5.75	30.37
1000	27.08	6.15	33.23	27.01	6.22	33.23

Table 5: WS tunnel client and Socat averaged CPU usage during 5 mins with different packets data sizes.

an overlay. To further investigate the system behavior vis-à-vis the packet size, we conducted other experiments with different payload sizes. We report in Figure 27b a comparison between 80 bytes data size packets and 500 bytes ones. As reported in this case (see Figure 27b), the CPU resource usage are quite aligned when changing the payload length.

As reported previously, the system performance depends on the number of packets generated within an overlay which is an aspect that depends strictly on the use cases. For example, in a typical IoT scenario where nodes generate continuously 50 pps in the destination of a Cloud VM, a machine hosting the WS tunnel agent with one vCPU can handle up to 36 IoT nodes without any packet loss due to the overlay (the packet loss can appear instead because of the transition network: the WAN). The number of managed IoT nodes can be increased, of course, by using a powerful hardware setup. However, we have to consider that Neutron and its virtual switches have a limitation regarding the number of packets it can deal with within a single hosting machine as reported in [64]. For this purpose, taking into consideration this later limitation while dimensioning the system is essential. It is worth mentioning here that our approach can support the use of several WS tunnel agents within different machines to ensure the horizontal scalability of the system. Therefore, issues that can be inflicted by

Neutron and its virtual switches can be bypassed. In this case, as discussed before (see Section 2.4), the reachability among IoT nodes belonging to the same overlay but connected to different WS tunnel agents is ensured through the use of VXLAN/GRE tunnels between the machines where the WS tunnel agents are deployed.

After evaluating the system scalability from the Cloud perspective, we also measured the CPU and RAM usage on an IoT node within a Neutron overlay. We considered as an IoT node a Docker container with 1 vCPU and 1 GB of RAM. The RAM usage for the solution was around 4.56% (`Socat` and WS tunnel client use 0.32% and 4.24% respectively) during all the tests. We noticed that the RAM usage does not get impacted by increasing/decreasing the packet generation rate. For the CPU usage, we report in Table 5 the variation of the CPU usage for the WS tunnel client and `Socat` (the two processes involved when using the overlay). As shown in the table and Figure 2.4, the packet size does not impact the CPU usage. In fact, the results of the CPU usage when using packets with 80 bytes and 500 bytes of payload are aligned.

Chapter 3

Containers deployment at the network edge

3.1 Introduction

The current ICT scenario is dominated by large and complex systems, paving the way to a ZettaBytes (BigData) landscape made up of a plethora of connected objects and devices. Devices usually equipped by a wide range of sensing resources and relatively advanced computing, storage and communication capabilities, often referred to as smart, populate this scenario, thus highlighting the need for facilities for their management. As discussed in the previous chapters, most of the efforts categorized under the IoT umbrella term are mainly focused on managing IoT data using Cloud platforms (i.e., the data-centric approach [30] [31]). However, other important aspects related to data pre-processing and resource management at the network edge have to be addressed.

With regard to sensing, basically, a sensor periodically checks, probes or queries the observed system to provide updated information on its status. This information may be gathered, processed or also stored for further handling. In some cases, the system may require multiple phenomena observations from different sources to be properly sensed using sensor networks, which in turn may require complex algorithms for their processing and, in particular, specific techniques for managing the (often huge) datasets thus generated. Besides the amount of data IoT deployments may generate, a new set of time-sensitive applications is taken ground, therefore pushing for quicker data processing delays. What is currently to be investigated more thoroughly in IoT is data processing at the lowest level, i.e., closer to the source.

According to the above considerations, here follows a set of developed mechanisms for processing duties to be pushed as close as possible to data sources, thus, leading

to perform computation either on the IoT devices themselves when they are powerful enough [46] or in their proximity (i.e., Fog computing [65]). In particular, the idea here is to enable Cloud users to inject intelligence on remote nodes in order to collect, preprocess, aggregate and mine sensed data at the network edge. The virtualization approach we are introducing here is meant to enable the IaaS computing paradigm in IoT environments when combined with I/Ocloud mechanisms (i.e., file system virtualization detailed in Chapter 1). Besides, with regard to Fog computing, the approach we propose introduce multitenant PaaS environments on top of shared edge-based Fog nodes.

3.2 Edge and Fog computing

3.2.1 The Cloud shortcomings

Thanks to the ample amount of resources (e.g., compute and storage) the Cloud provides, most of the IoT applications rely on it to deal with data management and processing. However, the centralized nature of the datacenters can induce a considerable topological distance between the resources hosted on the Cloud and the vast majority of IoT devices due to the geographical location. Accordingly, private Clouds might be the less affected model unless it is used to cover a wide area (e.g., an organization that deploys the datacenter in a city while having branches in other cities). On the opposite side of the spectrum, services/applications hosted on public Clouds platforms (e.g., Google, Amazon, and Microsoft) can be severely affected by the distance. Indeed, public Clouds are meant to serve a wide range of users by providing global coverage [66]. We can then have the Cloud datacenter located in another country or even continent from where a user is located. We discuss in the following the shortcoming of the typical data-centric Cloud/IoT integration:

- **Latency**: recently, we have seen the rise of a new set of applications that goes under the umbrella of Ultra-Reliable Low-Latency Communications (URLLC) category [67] where extremely low latency and response times are of utmost importance. Example of such applications include autonomous driving, smart

grids and industry 4.0 that require latencies less than 50 ms, 20 ms and 10 ms, respectively [68]. Relying on a distant Cloud to deal data processing of such applications may often induce high communication latencies, making it difficult to satisfy the imposed time constraints. In fact, from a statistical perspective, the average RTT between an Amazon Cloud server located in North California, USA and a client in Northern Virginia, USA is 62 ms; the RTT value then increases to reach 160 ms when the client is located in Milan, Italy; the RTT keeps the increase to reach around 218 ms if the client is in the Middle East [69].

- **Privacy and security**: with the adoption of IoT devices in different fields (e.g., health care, domotic, manufacturing), concerns about data security and privacy become more urgent. Transmitting personal and critical data to Cloud platforms can induce high privacy risks [70]. Since some IoT devices are resource constrained, they might be unable to deal with data encryption/decryption, therefore dealing in such a case with data confidentiality and integrity is important. Besides, legal implecations can arise since IoT data may be generated in a country then transmitted to a Cloud datacenter in another country where regulations about data privacy are different.

- **Bandwidth consumption**: with the proliferation in terms of the number of connected devices, the amount of data generated by IoT deployments is expected to exponentially increase within the next few years. By 2025, the number of IoT devices will reach more than 30 billion [71]. This massive number of IoT devices will undoubtedly generate an immense amount of data. Back in 2019, IoT devices generated around 13.6 zettabytes (i.e., 3352 billion gigabytes) of data. This volume of data is expected to keep the increase to reach around 79.4 zettabytes by 2025 [72]. Delivering such a high volume of data up to the Cloud may negatively impact users Quality of Experience (QoE) whislt imposing several challenges for backbone networks.

- **Context awareness**: context-awareness is the ability of a system to collect information about its environment at a given time and to adapt the behavior of some entities accordingly. Contextual or context-aware computing [73] uses

software and hardware resources to automatically collect and analyze data and then make adequate decisions. Due to the low coupling between the IoT devices and the Cloud due to the geographical location and lack of proximity, limited context is shared between the two entities.

3.2.2 Computing paradigms at the network edge

One of the biggest challenges that the Cloud is facing is to meet the increasing demands of the new services in terms of QoS, such as computational speed and lower latency [74]. For IoT services, the Fog computing paradigm is considered a crucial paradigm to meet their demand [75]. The fact that Fog computing nodes are topologically in proximity of end devices is a key enabler of advanced services/applications that were not feasible before while relying on the (remote) Cloud. Most of the solutions introduced to meet the requirements of IoT services are based on the use of small Cloud deployments, as the case of StarlingX[8] project. Such kind of deployments make use of powerful machines to build "cloudlets" close by end-users/devices. Another example of such deployments is the OpenStack-based platform called OpenVolcano [76] that was introduced for Fog-powered personal services deployment. However, the integration between the two platforms (i.e., OpenStack and Openvolcano) is achieved at the highest layer of the architecture named "data collection and configuration layer". OpenStack is used only to describe the requirements of the services while further configurations, such as networking and resources allocation/provisioning are leveraged to external (OpenStack) subsystems.

Recently, incumbent Cloud players are showing immense interest in the edge computing paradigm using relatively powerful nodes, such as IoT gateways and devices. For instance, Microsoft has released a platform for IoT called Microsoft Azure IoT Edge [77] that extends the Cloud concept toward the network edge using containerization technologies. The same can be said of Amazon that enhanced the pre-existing proprietary Cloud solutions using AWS Greengrass [78]; hence, they provide their consumers the capability to deploy customized applications on IoT devices and gate-

[8] https://www.starlingx.io

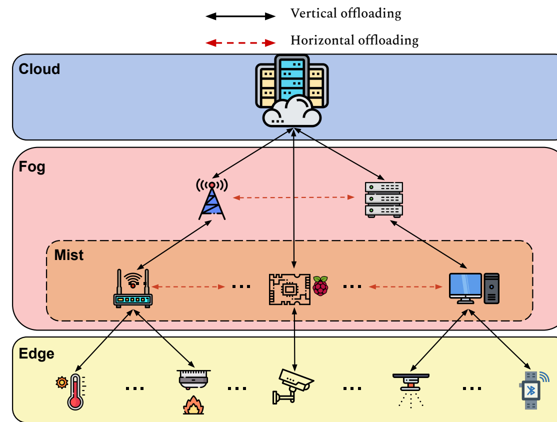


Figure 28: Overview of Cloud, Fog, Mist and Edge Computing in an IoT smart environment context.

ways. However, such solutions (a) are proprietary solutions; thus, they make the customers dependent to the vendors' solutions (i.e., vendor lock-in issues), (b) do not provide the ability to share the IoT infrastructure among different tenants the way Cloud platforms can, and (c) requires users to deploy and invest in their own IoT devices/gateways infrastructure.

Very few approaches have dealt with constrained nodes at the network edge akin to IoT gateways, SBCs and access points to make use of them and exploit their resources to satisfy the demands of specific services. Such nodes can be even shared among several users, thus extending the paradigm of PaaS towards the network edge [79]. Pushing Cloud resources to such kind of (relatively) powerful nodes is also known as Mist computing [80]. This paradigm can be considered a light-weight form of Fog computing that keeps computation even closer to the edge devices by using nodes (e.g., SBCs, access points) in the proximity of the edge IoT devices (e.g., resides directly within the network fabric). In this scenario, the edge devices could execute (relatively) simple data aggregation and filtering (or even no processing at all), then rely on Fog/Mist nodes for advanced processing tasks through vertical offloading [81]. We depict in Figure 28 the processing architecture involving the Edge, Fog/Mist, and Cloud layers according to the NIST definition [80]:

- **Edge Computing:** expects that data processing is held immediately on the

edge IoT devices accordingly, reducing network latency and overhead.

- **Mist Computing:** considers that the processing resources are close to data sources (e.g., SBCs) in order to provide support for the relatively constrained IoT devices. If Mist nodes are powerful enough, they can cooperate using clustering based on peer-to-peer communications [82].

- **Fog Computing:** presumes that resources such as compute and storage are relatively close to data sources. Fog nodes could be part of the access networks (e.g., servers, cloudlets, routers). Similarly to Mist nodes, Fog nodes can collaborate through clustering.

- **Cloud Computing:** is the highest layer of the stack that provides huge computational/storage resources for the lower levels (i.e., Fog, Mist, and Edge) when their capabilities are not sufficient.

In the literature, a number of works promote the adoption of the Fog computing paradigm with relatively powerful nodes. Paradrop [83] leverages the virtualization technology, specifically containerization, to provide users isolated and controlled containers at the network edge. Users then can make use of these containers to deploy their applications. In [84], an approach proposing multi-tenant IoT infrastructures using Raspberry Pis and LXC/LXD containers [85] is introduced. However, the implementation is at an early stage where only an architectural design based on OpenStack is presented without any further details about the services involved. In [86], the authors present a platform named PiCasso that uses SBCs (e.g., Raspberry Pi) as Fog nodes. Deploying services using PiCasso takes into consideration the specifications/requirements of the services and the available resources on the Fog nodes.

A management system that provides computing capabilities at the network edge using IoT gateways and devices should come up with a set of well-developed features capable of operating and managing, through WAN interconnections, a geo-distributed infrastructure. Accordingly, akin systems should be able to cope with unexpected aspects, such as NATs and firewalls (i.e., security policies and restricted settings) traversal.

3.2.3 Virtualization techniques for Edge/Fog computing

In the following, we give an overview of the virtualization approaches with a particular focus on the containerization technique. This later is a critical building block in realizing the I/Ocloud view described in Chapter 1 by instantiating edge-based VNs with user-space-defined file systems. On the other hand, the containerization at the Fog level can preserve multi-tenancy and users isolation as in Cloud platforms.

3.2.3.1 Virtualization approaches

The virtualization paradigm is not new as it dates back to the 1960s with the IBM 360 system that introduced this concept [87]. The hardware virtualization, also commonly named platform/hypervisor-based virtualization [88], is a set of features and techniques that make users able to isolate their runtime environments in the form of so-called VMs on top of a shared compute resource (i.e., a host machine). The VMs thus created act like real full-deployed computers as from the one hand, each of them runs its own operating system kernel and on the other hand, they do not have any access to, or visibility into, the underlying resources provided by the host machine [89]. In the virtualization terminology, we refer to VMs as guests, whereas the software that handles the virtualization process was called at its origins control program yet, the terms hypervisor and Virtual Machine Monitor (VMM) have been preferred over time [90].

Although the hardware virtualization technique has a set of advantages, this classical approach of virtualization is weak and not good enough as high density deployments are problematic to deploy and resource-consuming. In fact, using the hardware virtualization, most of the host machine memory resources are devoured by the multiple copies of the OS kernels of the VMs provisioned. For that purpose, OS-level virtualization [91], more commonly known as containerization, arises as a solution for such a problem. This virtualization approach, rather than dedicating a whole OS for each guest VM, it enables multiple isolated user-space environments to run on a single host machine while sharing a unique kernel provided by the host. Instances provided

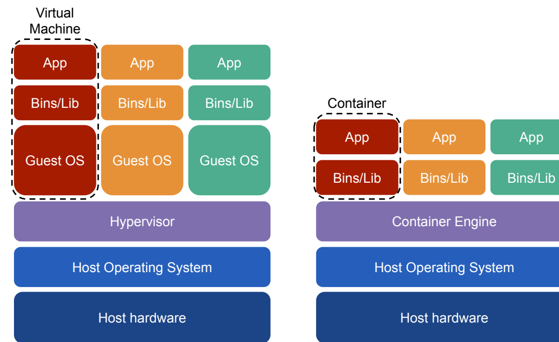


Figure 29: Platform/Hypervisor-based virtualization and Containerization comparison.

in this case namely, containers, still, from the perspective of users and the processes running on them, similar to real computers by making use of dedicated and isolated file systems, TCP/IP stacks, etc. Indeed, from inside a container, a user/process can only make use of a subset of resources made available through the kernel and intentionally assigned to that container.

The difference between the two virtualization approaches, hardware virtualization and containerization resides at the level where the resource partitioning and abstraction intervene (see Figure 29). In particular, in the case of containers, the abstraction of resources occurs at the host kernel whereas for the hardware virtualization it is below that level. The later virtualization paradigm (i.e., hardware virtualization) is characterized by higher complexity compared to the former as it makes use of a fully abstracted environment including the emulation of low-level (software-based) hardware management. Furthermore, it requires more storage capacity considering the multiple copies of OS kernels used. To make the virtualization more efficient, this approach calls for the hardware-assisted virtualization [92] that exploits the hardware capabilities of the host machine as the case for x86 processors with Intel VT-x and AMD-V.

Considering, on the one hand, the significant usage of resources with reference to computing/storage requirements in the case of hardware/platform virtualization, and on the other hand, the IoT devices constraints as the case of relatively smart embedded devices where the hardware-assisted virtualization requirements could not

be always satisfied, hardware virtualization can not be suitable for IoT environments. Such constraints make the containerization most suitable for the IoT ecosystem given its simplicity and reduced overall footprint [93] that enable, in their turn, advanced functionalities (e.g., containers migration) [94].

3.2.3.2 Containerizations

The emergence of containers revolutionized the IT field. This technology has become a key (open source) solution for application packaging and delivery, combining lightweight application isolation with a significant level of flexibility for image-based deployments. Containerization is used in several forms considering the large scope of usage and the different techniques used [95]. In the following, we will focus on Linux-based containers since they are mature and Linux fully supports containerization.

Linux Containers are enabled based on core technologies provided by the Linux kernel, such as Control Groups (Cgroups) [96] for resource management, namespaces for process isolation, and Secure computing mode (Seccomp) for security and sandboxing. The Linux kernel uses the former (i.e., Cgroups) in order to group a set of processes for the sake of the system's resource management. In particular, the Cgroups capability provides flexibility as it enables dynamic allocation of resources including system memory, CPU time, network bandwidth, or a combination of them within user-defined groups of processes. Namespacing [97] is a feature made available through the kernel that provides processes isolation. In container-based deployments, containers are isolated through the use of namespacing as for each container a set of dedicated namespaces gets created. Indeed, a namespace enables the creation of an abstracted instance with a particular system resource (e.g., Mount, UTS, IPC, PID, Network) and makes it behave as a fully separated instance with regards to the processes running inside it. Therefore, multiple containers can simultaneously use a shared resource without inflicting any kind of conflict. Seccomp [98] is a feature made available through the Linux kernel to make restrictions over the system calls that a process can use. For instance, when a developer uses potentially unsafe/unverified code or software, Seccomp provides an efficient approach to isolate and restrict the

code/program from using calls that have not been already permitted and declared. In the case of containers, Seccomp prevents against the miss behavior of launched applications inside them that could gain access to the host kernel and compromise it.

3.2.4 Containers migration and Fog computing

Extending the Cloud paradigm towards the network edge introduces novel aspects that may either lead to new research challenges or change the way to deal with the existing ones. As one of the main new research problem that arises with the Fog computing paradigm is users/devices mobility support. As already discussed, Cloud services are, in general, distant from the served users irrespective of the position of the latter; thus, end users mobility is not an issue in Cloud-only environments. Contrariwise, the Fog computing paradigm benefit is principally related to the proximity of Fog nodes from end devices. Consequently, users and devices mobility is a critical parameter to deal with as it may compromise the Fog benefits. In fact, when a device/user moves from an area to another one, the topological distance separating the Fog node and the device/user may increase and thus, impacting negatively the QoS. In the literature, a set of approaches are proposed to cope with this issue by making the services running on Fog nodes able to migrate across the Fog infrastructure[99] [100] [101].

3.3 Zun, Kuryr and IoTronic integration

In the following, we present our solution for managing the instantiation of containers at the network edge. We mention that the system is integrated within the S4T middleware. We present the two OpenStack subsystems, Zun and Kuryr, that we used to conceive the system. Afterwards, a description of the integration between Zun, Kuryr, and IoTronic is reported.

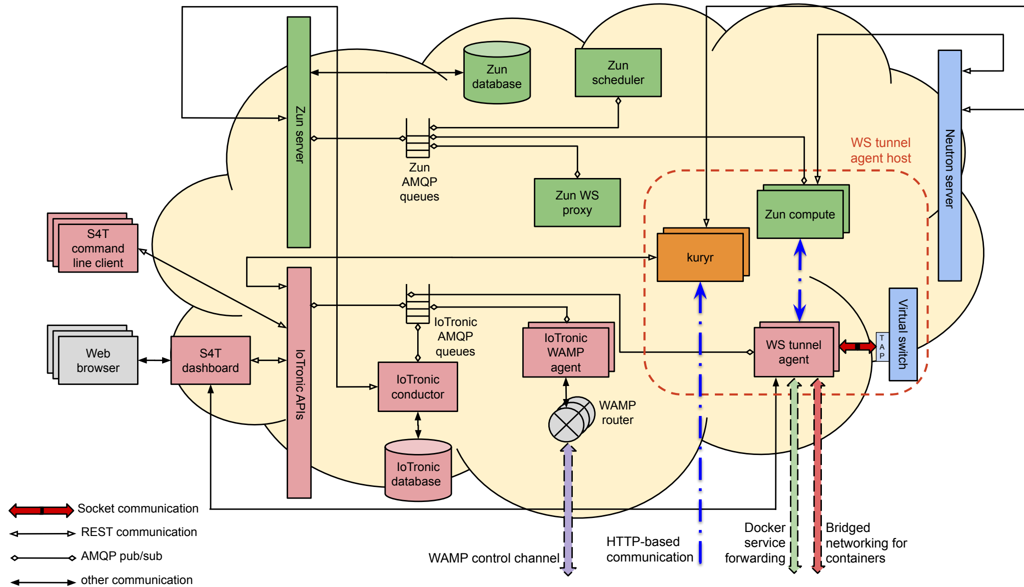


Figure 30: Stack4Things Cloud-side containerization subsystem architecture.

3.3.1 Zun and Kuryr subsystems

OpenStack is a mainstream solution used for providing Linux-based IaaS deployments. Recently, with the emergence of containers in Cloud services, it is becoming a necessity for OpenStack to fully support container-based deployments and make the infrastructure resources (i.e., storage, computing, and networking) available for them as well. Rather than setting up new vertical silos to manage containers in OpenStack-enabled Clouds, users might find efficient and useful a cross-platform API that handles all the kind of Cloud instances (i.e., VMs, bare-metal servers, as well as containers).

As the OpenStack subsystem that provides containers management, Zun[9] (the green subsystem in Figure 30) makes the users able to rapidly launch and operate containers without dealing with servers and clusters management. At the backend, Zun uses several technologies while providing well-defined APIs that manage containers in a abstracted manner; hence hiding the complexity of the workflows. Zun supports Docker [102] as a container runtime tool and can, optionally, cooperate with other OpenStack subsystems, for instance, Glance in order to manage containers images instead of pulling them from public repositories (e.g., Docker hub) and Neutron

[9]https://docs.openstack.org/zun/latest/

(through Kuryr) to provide, for the containers, advanced networking capabilities (e.g., creating overlays). At the core of Zun components, we have the Zun-compute agent that runs on the compute nodes (where the containers get provisioned). By performing (almost) all backend operations, this agent hides the workflows of the services.

Talking about VM-based Cloud computing deployments, particularly the IaaS model, networking is an essential feature as it is provided on demand for the users. In fact, IaaS deployments grant a considerable level of flexibility and manageability for the users as networking configurations of their instances are entirely under their control. To provide the same level of manageability in container-based deployments, a recent OpenStack project arises under the name of Kuryr[10]. In particular, Kuryr-libnetwork is a Docker networking plugin leveraging Neutron to provide networking services for Docker containers. This project aims to use the abstraction level and all the complex services that Neutron and its corresponding plugins maintain to provide networking capabilities, resources, and services for container-based deployments. The abstraction of networking features is mapped, using Kuryr, to standard Neutron APIs. Therefore, in a stable and effective experience, users can interconnect all Cloud instances, namely VMs, bare-metal servers as well as containers, to the same (logical) network with persistent networking capabilities (e.g., floating IPs, security groups).

3.3.2 The integration scenario

In the following, an extension of the Cloud-based management system provided by OpenStack to the network edge is presented. In particular, our approach uses Zun and Kuryr to provide management and networking services for Docker containers deployed at the edge of the network, specifically provisioned on top of geographically dispersed nodes. This is achieved through RESTful interactions with IoTronic.

A major difference from data center-based OpenStack deployments is that, in our approach, the three components Zun-compute, Kuryr, and the Docker engine are not co-located on the Cloud, specifically on the same host (i.e., compute node). In fact,

[10]https://docs.openstack.org/kuryr/latest/

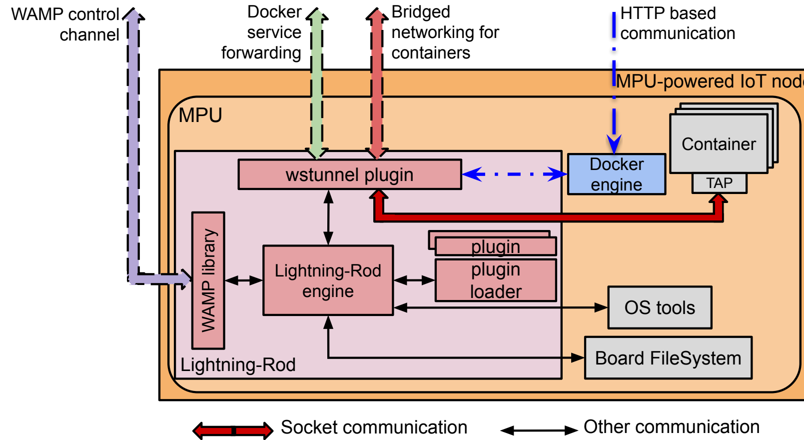


Figure 31: Stack4Things board-side containerization subsystem architecture.

to extend the management scope provided by Zun and Kuryr in Cloud deployments and make them able to deal with OS-level virtualization at the network edge, we adapted the typical Cloud compute nodes (red dashed box in Figure 30) as Cloud-based machines hosting IoTronic WS tunnel agents, Zun-compute agents as well as the OpenStack switching platforms (i.e., virtual switches) managed by Neutron. The Docker engines instead operate on the remote nodes where the containers get instantiated (see Figure 31).

Figure 31 depicts the architecture of an IoT edge node (in this case, an MPU-powered SBC). At the core of the IoT node architecture, we have the S4T LR engine that provides interactions with the Cloud using the WAMP libraries (violet arrow in Figures 30 and 31). Furthermore, the S4T WS tunnel libraries enable the engine to act as a WS reverse tunneling client in order to connect the node to a specific WS server running on the Cloud. The WS tunnels are used in our edge-based OS-level virtualization approach twice: firstly, to forward the commands from the Cloud to the (remote) Docker engines running on the IoT nodes (green arrow in Figures 30 and 31) and secondly, to provide networking facilities to the containers instantiated on top of the edge-based nodes by attaching them to the switching platforms hosted on the Cloud-side (red arrows in Figures 30 and 31).

In terms of the networking services provided by Neutron and Kuryr combined, in our approach, the Neutron ports will get instantiated on the Cloud-side, exactly on

the compute nodes (i.e., machines hosting the WS tunnel agents). Afterwards, these Neutron ports get attached to the containers provisioned on the remote nodes using a newly introduced Kuryr networking driver that interacts with IoTronic and uses Websocket as a communication channel (the mechanism detailed in Chapter 1).

In the following paragraphs, to describe our approach technical workflows, we focus on the case when using the Neutron Modular Layer 2 (ML2) plugin based on Linux bridge and VXLAN technologies as *mechanism driver* and *type driver*, respectively, as depicted in Figure 33. We mention that deployments based on other technologies using, for instance, OpenVSwitch as *mechanism driver* and GRE as *type driver* are feasible as well.

When a user wants to deploy a container on a specific node at the network edge, he/she sends a REST request to IoTronic (through the dashboard or CLI) that handles the process by creating the container then attaching it to the desired Neutron overlay. Indeed, IoTronic manages the workflow by:

- Setting up the environment by creating a forwarding service (the service is based on a port forwarding approach on the Cloud) for Zun by establishing a WS tunnel from the Cloud to the involved remote node (green arrow in Figures 30 and 31). This communication channel is used then to forward the container instantiation requests from the Cloud-based Zun-compute agent to the remote node.
- Interacting with the Zun-API server using REST to instantiate a container on the specified host (i.e., the WS tunnel agent node in charge of the involved node). This request will then be forwarded to the Docker engine on the remote node using the WS tunnel created previously.
- After the instantiation of the container on the remote IoT node, the two subsystems, Kuryr and IoTronic, cooperate in providing networking services for the container (i.e., create a TAP class device on the container, pipe traffic to/from it to the WS tunnel used for networking duties, finally attach the TAP interface to the container).

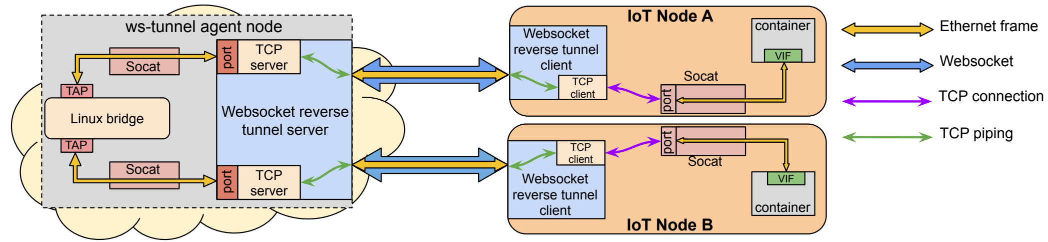


Figure 32: Low-level functional diagram of bridged tunneling over WS for edge-based containers (boards managed by the same WS tunnel agent).

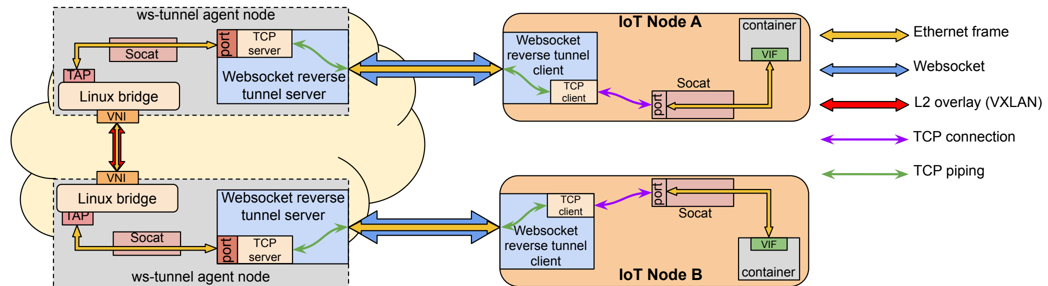


Figure 33: Low-level functional diagram of bridged tunneling over WS for edge-based containers (boards managed by different WS tunnel agent).

We mention that the edge-based containers that belong to the same logical network but, managed by different WS tunnel agents is taken for granted as Neutron, by default, set up a full-mesh topology among all the compute nodes (i.e., WS tunnel agents nodes). This feature is based on overlay technologies, such as VXLAN and GRE that enable network isolation.

A high-level overview of the approach design is depicted in Figures 33 and 32. The scenario shows the case when two containers provisioned on top of two remote IoT nodes yet, the containers belong to the same logical network. As indicated previously, when the user solicits the instantiation of a container on a specific remote node, IoTronic deals with the process. In particular, as an initial step, IoTronic establish the WebSocket tunnel bridging the Cloud and the involved node. This particular tunnel is used to forward the requests to the Docker engine. Specifically, IoTronic sends the container instantiation request to the Zun-API server that forwards the request to the Zun-compute running on the WS tunnel agent host in charge of the remote node. The request is then forwarded (using the service forwarding WS tunnel) to the Docker engine on the remote node. As a next step, Kuryr and IoTronic together

deal with the rest of the workflow by connecting the already instantiated container to the requested network. The newly introduced Kuryr driver attaches the Neutron port once created on the WS tunnel agent host in charge of the node to the (remote) container. The interconnection is made by exploiting a reverse WS tunnel (rtunnel) and `Socat`-based piping. Specifically, the link is created by leveraging, on the Cloud-side (i.e., the WS tunnel server), using `Socat`, a listening socket (determined by IoTronic) connected to the TAP interface associated with the Neutron *port*. Once the Cloud-side rtunnel server gets the request to create the tunnel and accept it, the `Socat`-based TCP connection is then piped to the WebSocket-based tunnel established (blue arrow in Figures 32 and 33). On the rtunnel client-side (i.e., the edge node), an identical workflow is initiated by connecting the TCP client (of the rtunnel) to a TAP device that is created based on the metadata specified by Neutron (e.g., MAC and IP addresses), then this `Socat`-based TCP connection is piped to the (rtunnel) WebSocket communication channel. Finally, IoTronic attaches the (latter) TAP interface to the container.

This proposed approach of S4T edge-based containerization service has been conceived taking into consideration the constraints and limitations of edge nodes to make the system efficient and scalable. In fact, the IoT nodes are not involved in the complex workflows and duties, specifically the network virtualization ones since they are completely not aware of Neutron/Kuryr subsystems that are running on the Cloud consequently, keeping essentially the footprint of the approach lightweight for the relatively constrained nodes. Besides, the fact that Kuryr, Neutron, and their corresponding switching platforms are running on the Cloud side provides robustness and availability for particular complex configuration requirements and mission-critical Neutron services (see Figures 30, 31, 32, and 33). In the architecture we propose, the edge-based nodes are responsible only of provisioning the containers. In this context, the relatively constrained IoT nodes (e.g., SBCs) offer good scalability to provision multiple containers [103].

#	Method	URL	Semantics	Parameters	Return type
1	PUT	/v1/boards/{board_uuid or name}/containers	Create a container on a specific board and attach it to an overlay	Image (string) Network_uuid (uuid) Subnet_uuid (uuid)	Container
2	GET	/v1/containers	Retrieve a list of containers	–	ContainerCollection
3	GET	/v1/containers/{board_uuid or name}/containers	Retrieve a list of containers on a board	Board_uuid (uuid) or name (string)	ContainerCollection
4	GET	/v1/containers/{container_uuid}	Retrieve details about a container	Container_uuid (uuid)	Container
5	GET	/v1/boards/{board_uuid or name}/containers/{container_uuid}	Retrieve details about a container on a board	Board_uuid (uuid) or name (string) Container_uuid (uuid)	Container
6	DELETE	/v1/boards/{board_uuid or name}/containers/{container_uuid}	Delete a container from a board	Board_uuid (uuid) or name (string) Container_uuid (uuid)	–
7	DELETE	/v1/ports/{container_uuid}	Delete a container (force delete)	Container_uuid (uuid)	–
8	PUT	/v1/boards/{board_uuid or name}/containers/{container_uuid}	Migrate a container from a board to another one	Board_uuid (uuid) or name (string) Container_uuid (uuid) Board_uuid (uuid) or name (string)	Container

Table 6: The Stack4Things IoTronic RESTful containerization APIs.

3.3.3 Container instantiation workflow

In this subsection, the workflow of creating an edge-based container attached to a specific network using IoTronic is described. The request corresponds to the call #1 in Table 6. We highlight the different interactions among the subsystems involved (i.e., IoTronic, Zun, Kuryr, and Neutron). As a use case essential requirement, we presume that the hosting node (a SBC in this case) is already registered to the Cloud. We do not target the application-level use case considering that it is impacted by the logic of the deployed application. The following steps describe the workflow when instantiating a container (see Figure 34):

1. The user requests to instantiate, on a specific remote node, a container attached to an OpenStack/Neutron network using either the dashboard or the CLI.

2. The dashboard performs one of the available IoTronic APIs calls via REST.

3. The IoTronic conductor pulls the message from the IoTronic AMQP queue and then performs a query on the IoTronic database. In particular, it checks if the board is already registered to the Cloud and decides the WS tunnel and the WAMP agents to which the node has to be connected to. Finally, it generates a free TCP port to be used for forwarding socket-based requests from/to Zun compute to/from the Docker engine running on the board (from now on, we refer to this mechanism as Docker forwarding service).

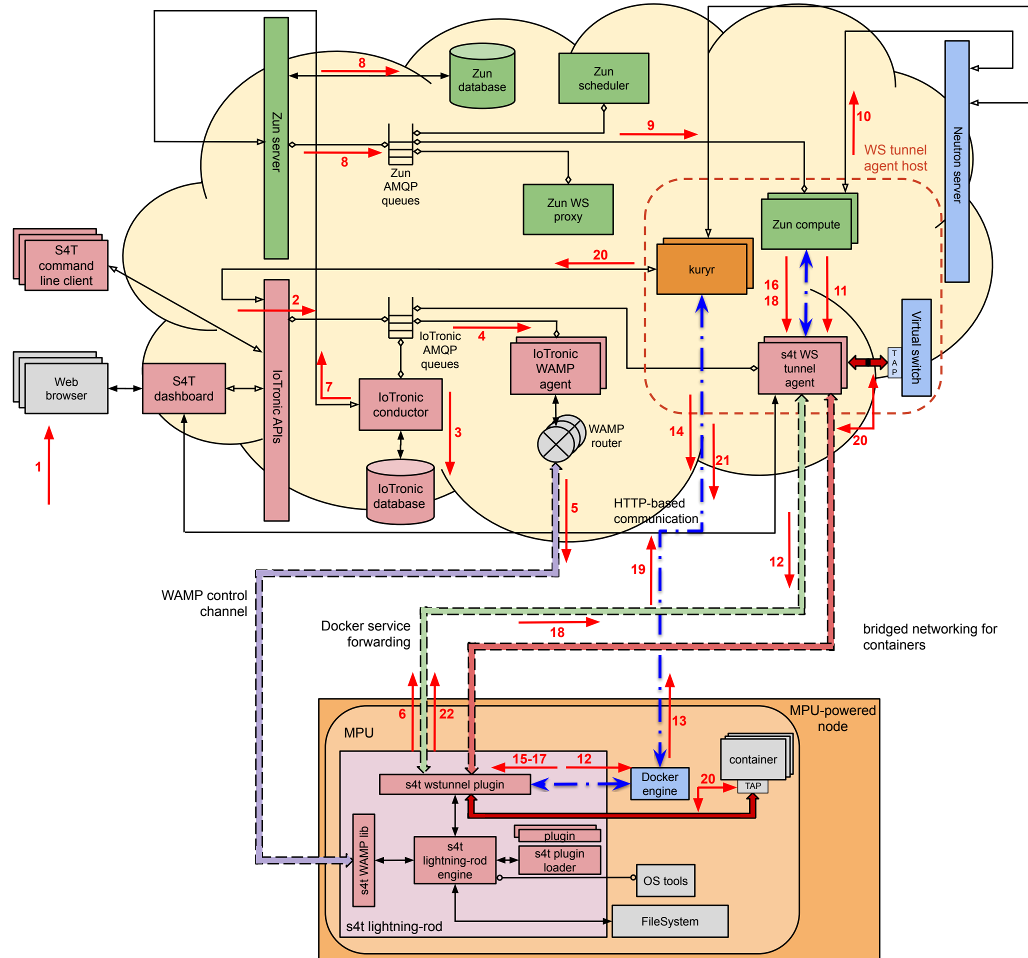


Figure 34: A container instantiation workflow using the S4T system.

4. The S4T IoTronic conductor pushes a new message into a specific AMQP IoTronic queue, then the S4T IoTronic WAMP agent, to which the board is registered, pulls the message from the queue and publishes a new message into a specific topic on the corresponding WAMP router.

5. Through the S4T WAMP lib, the S4T LR engine receives the message by the WAMP router.

6. The LR engine opens a reverse WebSocket tunnel to the S4T IoTronic WS tunnel agent specified by IoTronic also providing the TCP port on which the Docker deamon is listening (by default, it is 2375).

7. The S4T IoTronic conductor performs a REST call to the Zun server to create a container on a specific host (i.e, WS tunnel agent host).

8. The Zun server creates, on his local database, an instance associated with the container then, it pushes an RPC into the AMQP to create the container on the specified host (i.e, WS tunnel agent host).

9. The Zun compute agent pulls the call from the AMQP queue.

10. The Zun agent performs a REST call to the Neutron server in order to get the available networks.

11. After receiving the response of the REST message, the Zun agent sends a REST request using the port provided for Docker service forwarding in order to create the network in Kuryr environment.

12. the request is forwarded to the board using the Websocket based tunnel already created for Docker service forwarding. Once received, the request is piped to the port on which the Docker engine is listening.

13. Docker engine sends a REST request to Kuryr in order to create the network to which the container should be attached.

14. Kuryr creates the network on his environment, then it sends back the response of the request to the Docker engine.

15. The Docker engine forwards to the Zun compute agent, through the Docker service tunnel, the response of the received request.

16. The Zun compute agent sends to the Docker engine a REST request to create an isolated container (i.e., not attached to any network).

17. The Docker engine creates a container on the board then sends back a response to the Zun compute agent about the status of the creation of the container

18. The Zun compute agent forwards a REST-based request to the Docker engine in order to attach the container already created to the specified network.

19. The Docker engine performs a REST request to Kuryr in order to attach the container to the specified network.

20. Kuryr using a newly introduced driver, it performs a REST call to the IoTronic server that triggers a set of operations to configure the container on the board side (i.e., creates a tap interface, sets up a WebSocket based reverse tunnel with the WS tunnel agent specified by the conductor). Then, it configures the Cloud side by creating a TAP interface associated with a Neutron port. The driver then connects the TAP devices on both sides (Cloud and remote node) to the rtunnel using a socat-based piping mechanism. Finally, IoTronic attaches the TAP on the board to the container.

21. Kuryr sends a response for the request to the Docker engine.

22. The Docker engine sends back a response to the Zun compute.

3.3.4 Container migration workflow

We describe in this part the workflow of a container migration process between two remote boards. The request corresponds to the call #8 in Table 6. The different interactions between the involved subsystems are highlighted. As the previous use case, we assume that the two boards used in this part are already registered to the Cloud. Concerning the migration, we make use of the Checkpoint and Restore in Userspace [11] (CRIU) Linux-based tool that can freeze a running application and checkpoint it to the memory as a set of files. In order to make the workflow uncomplicated and easy to be understood, we focus on the case where the two boards are managed by the same WS tunnel agent [12]. The steps listed bellow describe the workflow of the migration process (see Figure 35):

1. The user sends a request (using either the dashboard or the CLI) to migrate a container from a specific (source) board where it is actually running to another one (destination board).

2. The dashboard performs one of the available IoTronic APIs calls via REST.

[11] https://criu.org

[12] In case the two boards are not managed by the same agent, an additional step is required. Specifically, IoTronic do a check on its database to select the agent in charge of the destination board before starting the migration process

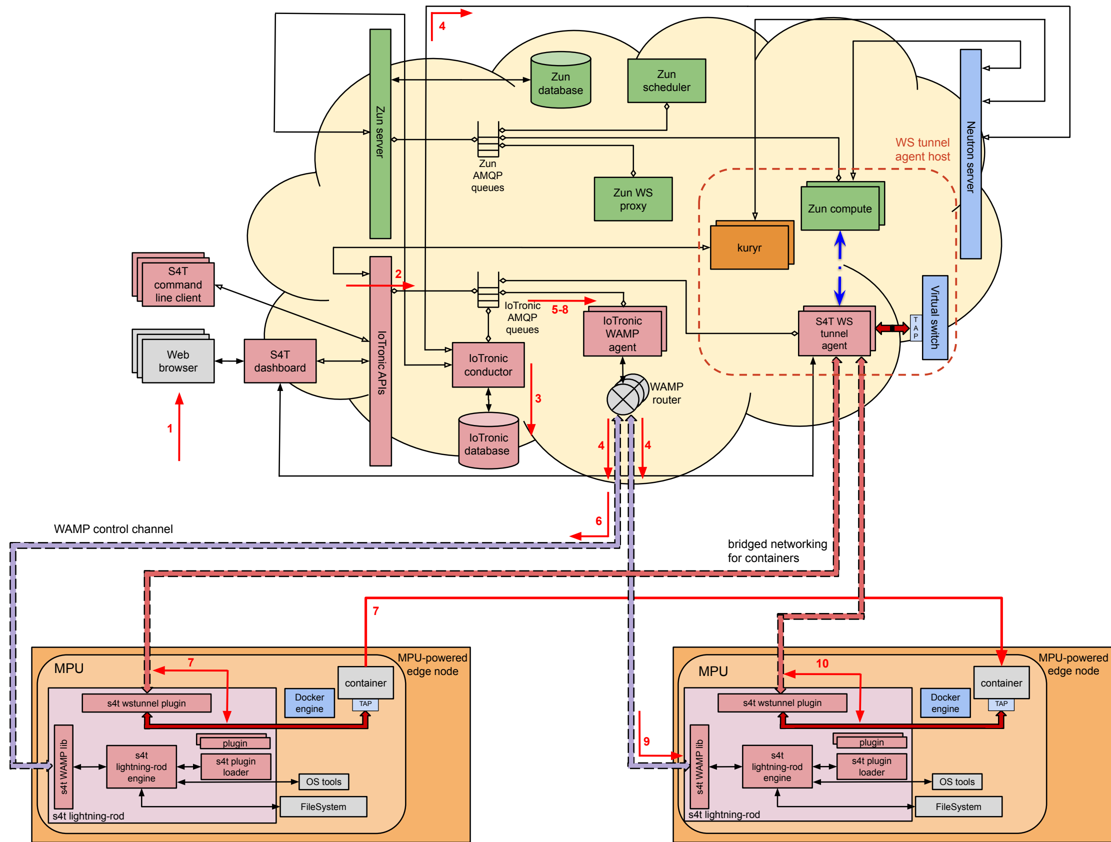


Figure 35: A container migration workflow using the S4T system.

3. The IoTronic conductor pulls the message from the IoTronic AMQP queue and then performs a query on the IoTronic database. In particular, it checks if the two boards are already registered to the Cloud and decides the WS tunnel and the WAMP agents to which the boards have to be connected to. Furthermore, it checks if the specified container is actually running on the source board.

4. IoTronic through a set of interactions using HTTP with Neutron and the two involved boards using RPC messages, attaches the two boards to the same overlay used for the migration service [13].

5. The IoTronic conductor pushes a new message into a specific AMQP IoTronic queue, then the S4T WAMP agent to which the source board is registered pulls the message from the queue and publishes a new message into a specific topic on the corresponding WAMP router.

[13]Note that the 4th step for the creation of the overlay between the two boards and the 11th step that removes the overlay have been detailed in Chapter 2

6. Through the S4T WAMP lib, the LR engine receives the message sent from the WAMP router.

7. The LR agent proceed with the migration process by killing the processes involved (i.e., `socat` and wstun), then it creates a snapshot of the container using CRIU. Finally, LR uses of the migration overlay created in step 4 by specifying for CRIU as a destination the (overlay) IP address of the destination board.

8. After the migration, to attach again the container to the network, the S4T IoTronic conductor pushes a new message into a specific AMQP IoTronic queue to create the Websocket tunnel, then the WAMP agent to which the board is registered pulls the message from the queue and publishes a new message into a specific topic on the corresponding WAMP router.

9. On the destination board, the LR agent receives the message from the WAMP router.

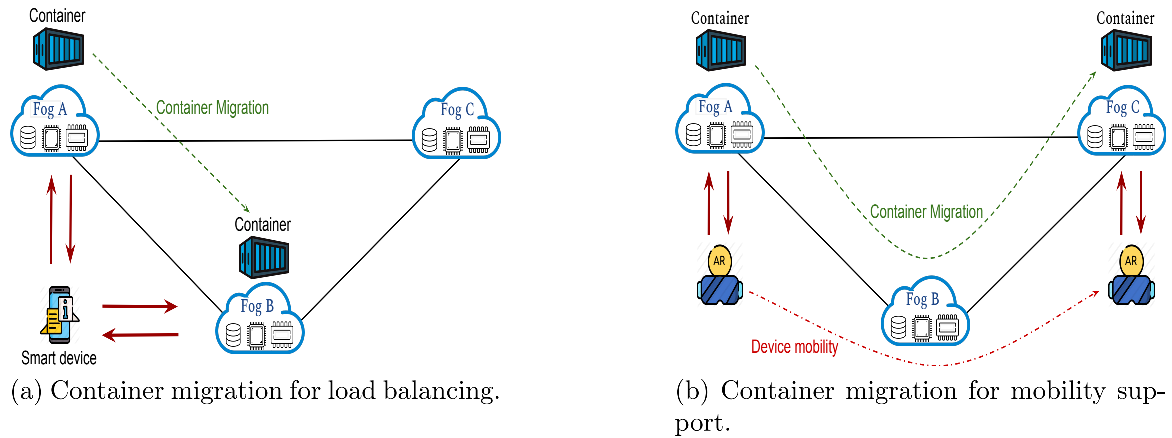
10. The LR engine opens a reverse WebSocket tunnel to the S4T IoTronic WS tunnel agent specified by the S4T IoTronic conductor (using the same Cloud-side port number as before the migration since the TAP interface on the Cloud-side remains the same: the two boards are managed by the same WS tunnel agent). And finally, LR pipes the traffic to/from the container TAP interface.

11. IoTronic interacts with Neutron and the two boards in order to remove the bare metal overlay created in step 4 (HTTP requests are sent to Neutron and RPC messages to the boards).

3.4 Empowered use cases

3.4.1 Fog Computing benefits In IoT

With the proliferation in terms of the number of IoT devices being deployed, IoT gateways are critical components in the actual IoT architectural design. The gateway-based approach is meant to be suitable when dealing with the complexity of



(a) Container migration for load balancing.

(b) Container migration for mobility support.

Figure 36: Use cases for containers migration

IoT deployments (e.g., number of devices, energy profiles, and connectivity models). Gateways in such a complex ecosystem are relatively powerful than end-user devices. In fact, they are characterized by more computational and storage capabilities considering the nature of tasks they must deal with. Hence, they are suitable to host applications in the form of containers as described in Section 3.2.2 and consequently, services can take advantage of their proximity to data sources. This particular feature of setting up containerized applications at the Fog layer opens the door for advanced features, such as services/containers migration (see Figure 36). Specifically, this capability is needed when a service is dependent and influenced by end-users mobility [104] or when a close management system is required for better and efficient management experience [105]. Here we highlight two important features provided by the Fog computing paradigm to enhance IoT services:

Orchestration: the Fog computing paradigm aims to provide a management layer based on small-scale Cloud datacenters or single Fog nodes totally in compliance with the Cloud. This multi-layer stacked architecture should support services migration between the Cloud and Fog nodes (i.e., vertical offloading) to meet the unsteady overtime services demands. In particular, considering the case of a time-sensitive application running on the Cloud, when the latency demands of the Cloud-based application are not satisfied anymore, a migration of the service from the Cloud to an edge node close by the end-devices is a must. In contrast, a migration from the edge

to the Cloud is also conceivable when a Fog-based application requires more computational power, for example. Another interesting use case is the horizontal offloading between Fog nodes for load rebalancing using containers migration as depicted in Figure 36a. Such an approach can even be automated by deploying advanced scheduling algorithms [106] [107].

Mobility: For particular services, containers/services migration is a crucial feature enabling the support of users/end-devices mobility [108] [66]. For instance, an Augmented Reality (AR) application for navigation is a service that requires (certainly) the migration of the service across different sites to ensure a good QoE. This is particularly depicted in Figure 36b where the AR service is shifting between three sites to follow the mobility of the end device. As one of the most known applications that has been enhanced thanks to AR, we can mention sightseeing services. A user/-tourist using a smart device (e.g., smartphone, tablet, smart glasses) equipped with a camera can take a walk through historical sites and get, in real-time, information about the different locations he/she is visiting. The user can even get lookbacks (using video streaming or pictures) in history and display the appearance of the sites long years ago. The AR service running on the smart mobile device superposes the real content of a video seen through the device camera with "artificial/unreal" content (i.e., sounds, videos, 2D images) provided by a Fog-based computational system. In such a case, the service migration between different sites (i.e., Fog nodes) is crucial to collect images sent from the user device, analyze them and then send back information, pictures, or videos to the user about the location in no time. In the same context, the wearable cognitive assistance as deployed in [109] using a platform called Gabriel uses the Fog computing paradigm. Such an application requires fast response time as it must react to the user movements in due time; thus, making the mobility of users an important factor to deal with.

3.4.2 Mobility support using Stack4Things

Owing to the meaningful advancement that Fog computing is knowing, the research community and service providers promote it to provide a set of applications. In the

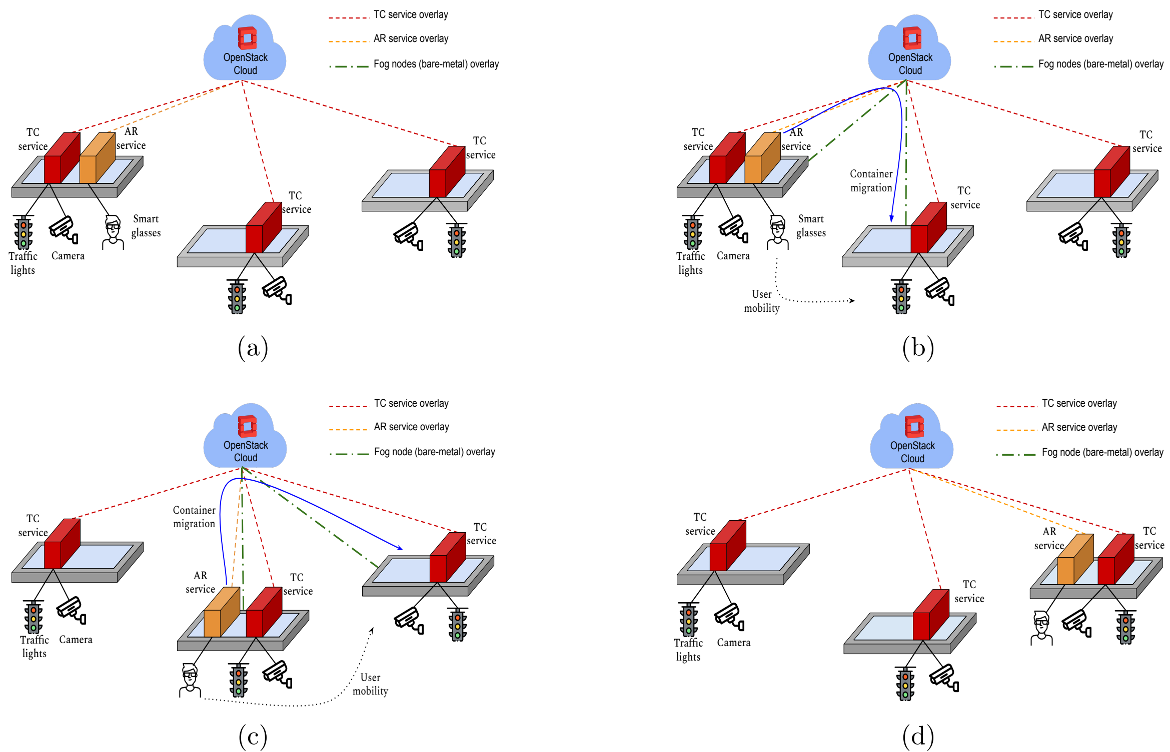


Figure 37: S4T Fog containerization use case. Two different services are depicted in this scenario, i) a Traffic Control (TC) service that collects data based on the smart cameras and traffic lights deployed across different sites ii) an Augmented Reality (AR) service following the user during his/her mobility.

following, we present our design that maintains the high mobility of users through providing applications/containers live migration among different Fog nodes. This approach could also be used for offloading purposes. In our design, we want to be totally agnostic from the infrastructure point of view as an infrastructure owner, in general, expects not to grant an administrator complete management abilities and privileges over his/her physical infrastructure; hence we want to decouple (totally) ownership from administrative capabilities. By doing so, we can provide third-party developers the ability to use the infrastructure in an abstracted fashion regardless of the underlying configurations.

Figure 37 highlights a use case of our platform with multiple Fog nodes where two different services (i.e., Traffic Control (TC) and AR) are running. As depicted in the figure, the two services are provisioned on top of a Fog node based on isolated containers. The S4T containerization solution provides flexibility and network abstraction

for users as the instances (i.e., containers) are totally unaware of the physical connectivity of the hosts. Furthermore, the fact that we deal with distributed containers as instances of the Cloud with their networking capabilities managed via Neutron (through Kuryr/IoTronic) makes the platform efficient. In fact, instead of setting up a new system to manage the geo-dispersed containers, we enhanced the OpenStack system by making it able to manage containers at the network edge. Figure 37 depicts a use case with different instances of a TC service connected to the same network (i.e., red dashed lines); thus, enabling possible advanced services, such as Machine to Machine (M2M) interactions.

Regarding containers live migration use case, the figure depicts an AR service following the user mobility. As shown in Figure 37, when the user is moving from an area to another one, the AR container will reside in proximity of the end-user by following him. To enable this feature, a virtual network is established among the bare metal nodes involved (green dashed lines in the Figure 37) as detailed in Chapter 2. At the end of the migration process, the migrated container will be attached to the same logical network as in the initial state (in Figure 37, the orange container at the end of the process is attached to the orange overlay).

3.4.3 Implementing services using Stack4Things

S4T middleware enable the users to deploy their services on top of nodes deployed at the network edge. Each user can provision a number of services thanks to the lightweight footprint of, and the isolation provided by the containerization technology. The S4T edge containerization solution provides a set of benefits:

Dynamic installation of services: S4T makes users able to deploy services on top of a geo-distributed infrastructure in an abstracted fashion and without dealing with (low-level) issues that regard the infrastructure (e.g., connectivity, networking configurations). Furthermore, the S4T framework provides services mobility through containers migration regardless of the hosts' networking configurations.

IoTronic APIs[14]**:** IoTronic provides for the developers ready-to-use RESTful APIs to conceive their services and enable their migration.

Networking setup: the networking between the geographically distributed virtualized environments (i.e., containers) is strong enough to provide the same level of capabilities as for Cloud instances. Indeed, the networking capabilities are provided, for the edge-based instances, through OpenStack subsystems (i.e., Neutron/Kuryr). Accordingly, having deployments arranging Cloud-based instances (i.e., VMs, containers and bare-metal servers) together with the remote containers are feasible using a unified system (i.e., OpenStack/Neutron).

Resource control: IoTronic provides an efficient management experience considering the multitenancy aspects used on the nodes that require resource control. The resources controlled by IoTronic are the ones provided by the Zun subsystem (e.g., CPU, RAM) and Kuryr/Neutron for networking services. Using IoTronic APIs, users can manage the containers resources based on the tasks they must achieve.

3.5 S4T environment emulation

3.5.1 Motivation

The S4T platform is a management system for IoT deployments that relies on the Edge-to-Cloud continuum depending on the QoS required. In this context, a platform providing the management of IoT deployments and advanced features akin to edge computing should be strong enough and scalable to deal with the complexity of IoT deployments. However, testing such management platforms in the real world is time-consuming and could be costly. Moreover, the uncontrollable behavior of networks' characteristics makes incorporating any feasible test hard to set up. Therefore, results we get in such environments are not expressive and change depending on the network status. In fact, such platforms might work perfectly during a period of the day or with a few number of devices, while the performances could decrease significantly during

[14] IoTronic Python APIs: https://mdslab.github.io/iotronic-api/swagger-ui/

busy hours and huge deployments.

To provide a persistent free test and evaluation system for IoT management platforms, there is a need to emulate, on the one hand, the realistic networks' conditions and, on the other hand, IoT deployments with a large number of devices. As discussed in this chapter and Chapter 2, the virtual networking infrastructure of S4T is (always) hosted on the Cloud. Therefore, any communication based on the overlays should transit through the virtual switches on the Cloud. However, relying all the time on the (remote) Cloud might not always be the best choice as it can induce considerable delays for typical time-sensitive applications. The idea we developed is that, instead of using the Cloud for switching, we can deploy the Neutron virtual switches on the Fog nodes using advanced distributed mechanisms as discussed in [81]. Consequently, IoT devices within a same overlay network can communicate using the virtual networking infrastructure hosted on a nearby Fog node. By knowing the characteristics of the links between the IoT devices meant to be on a same overlay and the available Fog nodes, a suitable decision about where to deploy the virtual switch could be made depending on the metric we want to optimize (e.g., latency, throughput, and packet loss).

3.6 Emulation system

The S4T emulation testbed we developed is based on a tool called Containernet [110]. The tool is a fork from the famous network emulator Mininet [111]. Even though the original Mininet provides an efficient solution to emulate a set of scenarios, the system has a number of limitations that make it inefficient to be adopted in our use case. In particular, the most significant constraint is related to Mininet's partial virtualization approach as it does not provide fully separated emulated hosts. In fact, an emulated host in Mininet is only a Linux process with Network and Mount namespaces with CGroups. Therefore, all the emulated hosts share the filesystem and processes. Containernet, on the other hand, provides a full virtualization approach based on Docker containers that use advanced isolation features (e.g., Mount, UTS,

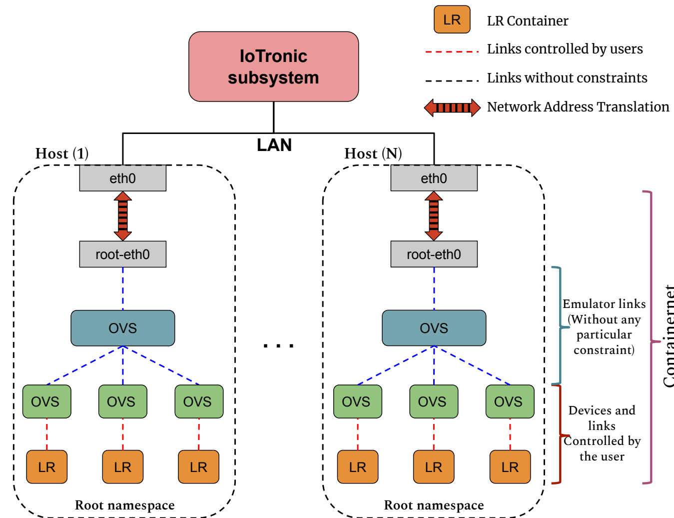


Figure 38: Integration design of S4T with Containernet.

IPC, PID, Network) for better sandboxing. This better isolation capability of the emulated hosts is critical for our S4T environment emulation as hosts should run, independently, a set of processes as required by the node-side agent LR. Furthermore, Containernet provides the capability to add/remove emulated guests (i.e., containers) at runtime.

In our integration scenario, we use containers to mimic the IoT infrastructure, including the Fog nodes and the IoT devices. In this context, Containernet provides considerable control over the containers resources and the deployment in general. Moreover, Containernet enables the users to launch new containers or stop others during the emulation. This particular feature is essential in our case, especially if we consider the dynamicity of IoT deployments.

3.6.1 S4T integration with Containernet

The aim of integrating the S4T platform with Containernet is to emulate the edge IoT devices, Fog nodes as well as the physical layer links (e.g., WAN interconnections). To emulate different types of interconnections between S4T (i.e., IoTronic) and the IoT nodes, we had to conceive a Containernet architecture providing users full control over each container link. Furthermore, these containers should communicate with instances

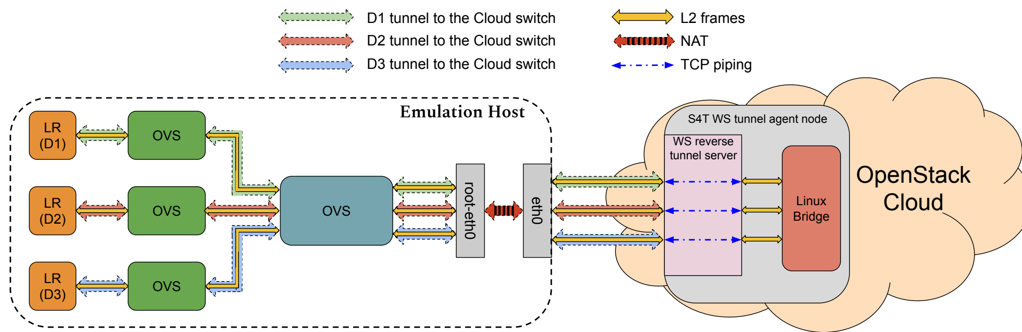


Figure 39: S4T Cloud-based overlay in the emulation deployment.

outside the emulator environment (i.e., the LAN where S4T/IoTronic is deployed and also the Internet). Therefore, having a Containernet deployment capable of reaching the outside world is a must. As a solution for this requirement, we came up with the architecture depicted in the Figure 38. To make the containers able to reach the LAN (where S4T is deployed) and the outside world (i.e., Internet), the S4T emulator creates a Containernet (i.e., a Mininet) node on the Root Namespace (i.e., the host machine where the emulation is running) with an interface named root-eth0. Then, the emulator configures NAT rules using iptables between the physical interface of the host machine and the Root Namespace interface (root-eth0). Afterwards, this later interface is then connected to an OVS switch (blue OVS in Figures 38 and 39) created inside the Containernet environment to which, a number of OVSs (green ones in Figures 38 and 39) get attached too. The links between the later OVSs and the different containers are controlled by the user that can set different constraints, such as bandwidth, latency, packet loss, etc. At the level of these links, the WAN network links can be emulated within our deployment. Whereas the upper part, starting from the links connecting the OVSs to the main OVS, is supposed to be without any particular constraint (i.e., links with throughput in the order of GB/s and without latency or packet loss). In our emulation setup, the containers acting as edge IoT nodes are built using a LR Docker image. We mention that the emulator configures the containers being instantiated to automatically be registered and connected to S4T. Besides, the emulator handles the tasks of exposing, for each container, a number of ports used by the S4T LR agent to provide different services (e.g., exposing local-running services using the Cloud).

Statistics	Physical testbed	Pysical testbed (S4T overlay)	Emulation (S4T overlay)
Min	0.25	0.16	0.31
Max	0.99	0.57	0.41
Mean	0.43	0.38	0.35
Stdev	0.07	0.06	0.56

Table 7: Throughput results (in Mbits/s) while uploading from an edge-device to a Cloud instance

As a result, from a single configuration file, a user can have a set of containers connected to IoTronic with specific constraints regarding their resources and the emulated interconnections to S4T. We mention that there is no limit considering the number of emulated hosts that can be leveraged. Users need only to run the emulation script to deploy the IoT devices with their links constraints and register all the emulated IoT nodes.

3.6.2 Emulator use cases

3.6.2.1 Emulator for metric estimation

In the previous chapter, we evaluated the S4T (Cloud initiated) virtual networking solution for edge-based deployments using a physical testbed. Specifically, we deployed an IoT node (i.e., Orange Pi zero) outside the network where our OpenStack Cloud is deployed. Accordingly, the IoT node can reach the Cloud through a WAN interconnection. We report in Table 7 the results obtained during a TCP upload from the device to a Cloud-based instance within the same virtual network.

In order to emulate such an environment using the S4T Containerenet-based emulator, we used an emulated testbed with one IoT device (i.e., container) connected to its associated OVS using a link with 0,43 Mbit/s as bandwidth (the throughput average we obtained previously using the physical testbed, see Table 7).

We ran the emulation for 10 min during which we measured the throughput using Iperf [15] tool. The result we obtained is aligned with the physical testbed as shown in

[15] https://iperf.fr/

	Cloud	D1	D2	D3
Cloud	–	30 ms	120 ms	70 ms
D1	30 ms	–	300 ms	200 ms
D2	120 ms	300 ms	–	60 ms
D3	70 ms	200 ms	60 ms	–

Table 8: Links latency (point-to-point) without involving the S4T overlay.

	Cloud	D1	D2	D3
Cloud	–	2 Mbits/s	1 Mbits/s	4 Mbits/s
D1	2 Mbits/s	–	3 Mbits/s	1 Mbits/s
D2	1 Mbits/s	3 Mbits/s	–	3 Mbits/s
D3	4 Mbits/s	1 Mbits/s	3 Mbits/s	–

Table 9: Links bandwidth (point-to-point) without involving the S4T overlay.

table 7. Indeed, we got using the emulation, as throughput average 0,35 Mbit/s while considering the physical testbed, we got an average of 0,38 Mbit/s.

3.6.2.2 Emulator as decision making aid

In the following, we present an interesting use case for our emulation setup. Figure 39 depicts the S4T (Cloud-based) network virtualization approach described in Chapter 2 using the emulator. The S4T network virtualization solution is based on the use of the Cloud-based OpenStack/Neutron switching platform (e.g., Linux Bridge) as highlighted in Figure 39. Nevertheless, as mentioned before, relying always on the Cloud virtual networking infrastructure can not be suitable for particular applications (e.g., time-sensitive ones); thus, we can make use of the Fog nodes to host the virtual switching platforms and consequently, reduce latency between IoT devices within a same virtual network. By knowing the characteristics of the links between the IoT nodes that should belong to the same overlay and the Cloud, a suitable decision about where to deploy the switch could be made depending on the metric we want to optimize. To evaluate all the possible scenarios, the emulation setup could be a useful tool to take the "best" decision. We consider, in this part, a scenario with three IoT devices (D1, D2 and D3) that should be interconnected among each other using the S4T virtual networking approach presented in Chapter 2. We suppose that each of the three IoT devices has enough resources to host the switching platform.

	Cloud-based switch	Switch based on D1	Switch based on D1	Switch based on D3
D1 – D2	162.85	306.7	308.95	276.4
D1 – D3	112.55	206.65	368.6	210.3
D2 – D3	201.2	510.5	66.8	68.9

Table 10: Emulation latency results (in ms) using the S4T network virtualization approach.

	Cloud-based switch	Switch based on D1	Switch based on D2	Switch based on D3
D1 – D2	0.936	2.76	2.76	0.94
D1 – D3	1.45	0.95	2.72	0.95
D2 – D3	0.941	0.92	2.86	2.86

Table 11: Emulation throughput results (in Mbits/s) using the S4T network virtualization approach.

Figure 39 shows the scenario where the devices are connected through the standard Cloud-based switch. The point-to-point links characteristics (without involving the S4T virtual networking solution) of the scenario we are considering are reported in Tables 8 and 9. Each of the experiments (i.e., the scenarios mentioned in Table 12) conducted by the emulator, in this part, had a total duration of 10 minutes during which, we measured between all the devices (based on the S4T network virtualization approach), the bandwidth average (using Iperf) and latency (using the MTR Linux-based tool). Accordingly, based on the results obtained, the emulator system could make the best switch placement decision (see Tables 10 and 11). We report here, as an example, the average latency calculation of the scenario when the switch is deployed on D1:

$$Avg_{latency} = \frac{L_{D1-D2} + L_{D1-D3} + (L_{D2-D1} + L_{D1-D3})}{3}$$

$$\Leftrightarrow Avg_{latency} = \frac{L_{D1-D2} + L_{D1-D3} + L_{D2-D3}}{3}$$

$$\Leftrightarrow Avg_{latency} = \frac{306,75 + 206,65 + 510,5}{3} = 341,3$$

The results we obtained are summarized in Table 12. As we can notice from the table, to set up a virtual network between the three devices while having the lowest latency, the Cloud-based scenario is the suitable solution with 158,86 ms. While in

Scenario	Latency (ms)	Bandwidth (Mbits/s)
Cloud-based switch	158.86	1.109
Switch hosted on D1	341.3	1.54
Switch hosted on D2	248.11	2.78
Switch hosted on D3	185.2	1.58

Table 12: Averaged results considering the different scenarios possible (based on the S4T network virtualization approach).

order to get better performance, from the bandwidth point of view, deploying the switch on D2 is the best choice with 2,78 Mbits/s. Of course, complex metrics can be used to make the decisions.

Chapter 4

Enabling SOA in IoT using RESTful Web services

4.1 Introduction

With the advances the hardware design and production fields are knowing, embedded systems prices have decreased significantly, thereby making faster their adoption in different fields. Such devices are becoming nowadays able to communicate as well as performing useful tasks to provide newly added value services. In this context, as one of the main facets of pervasive computing, the IoT paradigm is inspiring and driving innovation in several sectors. In the vision of IoT, the Internet is spreading rapidly beyond its classical core composed of powerful servers and other fringe machines/devices (e.g., computers, smartphones) to billions of constrained embedded devices (e.g., sensors and actuators).

In the previous chapters, we introduced our I/Ocloud view as well as a set of enabling mechanisms aiming at providing virtualized IoT nodes as Cloud-based IaaS instances. In particular, we introduced our approach for sharing the IoT infrastructure/resources and deploying virtual representations of the physical IoT nodes either on the Cloud or the network edge. In this chapter, we focus on the way of exposing the virtualized resources (i.e., virtual sensors and actuators) and the data being processed and managed by these I/Ocloud instances (i.e., VNs) deployed at the network edge. Indeed, after deploying the application on a virtual IoT node, the virtual resources and the processed data should be exposed over the Internet. To this end, we opted for RESTful Web services to be aligned with the Web of Things (WoT) paradigm [112].

The WoT approach aims at adopting Web technologies to interact with IoT resources; thus unifying the application protocol to provide a homogeneous IoT environment. Indeed, due to the diversity in terms of service providers and manufacturers

in the IoT landscape, and even though IoT objects would have Internet connectivity, this field is still fragmented and far from being a homogeneous environment where devices can cooperate transparently. The actual IoT ecosystem seems similar to the Internet before the arise of the World Wide Web in early 90s. At that time, the Internet was based on proprietary solutions and competing hypertext systems that made the Internet a complex system made-up of small incompatible islands that lack a unified communication layer [113].

To merge the cyber world with the physical one and make IoT an integral part of the Internet, reusing existing Web technologies and standards is a suitable choice. In this case, IoT objects, either physical or virtual, will not be only IP-enabled devices connected to the Internet yet, they become able to communicate and cooperate using the same language. Therefore, they can interact among each others and with other components from the existing Web world. In such a homogeneous environment, smart objects will be able to offer their functionalities (e.g., sensed and processed data) via RESTful Web services (also called Web APIs). For example, an embedded system with a temperature sensor that collects measurements from the physical world can provide its real-time sensed data as a smart service (i.e., a Web service). We can build then an ecosystem where the smart objects can offer their functionalities as Web services that other entities (e.g., other devices, Web services, applications) can make use of to provide appealing services and applications.

In this chapter, following the trend aiming at incorporating the cyber world of IoT within the Web wisdom, we introduce an innovative approach to enable the use of the Web services in the IoT landscape. In particular, we accommodate relatively powerful IP-enabled devices to expose hosted services as Web resources that can be consumed in a transparent fashion. Specifically, we expose IoT services in a secure manner using the REST paradigm thus, we talk then about RESTful Web services in IoT.

4.2 Service-Oriented Architecture and IoT

4.2.1 Service-Oriented Architecture (SOA)

The Service-Oriented Architecture (SOA) [114] consists in designing distributed applications using reusable and interoperable components whose interactions take place based on messages exchanges. This architecture offer an obvious advantage which is interoperability. This advantage implies that applications can invoke and interact with each other regardless of their underlying platforms, geographic locations, and languages in which these applications have been developed. The SOA design is based on: (i) the emergence of a service layer separating the services interfaces (the what) from their implementations (the how). This separation makes the clients not concerned with the way how a service will execute their requests. (ii) The use of standard mechanisms for the publication, search, invocation, and composition of these services. Thanks to the service concept, the SOA design has been very successful making it possible to orient the latter towards a wide variety of aspects beyond the initial software architecture domain. Today, SOA is seen as a suitable architectural style to be adopted in different use cases [115].

Software architectures should be built upon well defined characteristics in which loose coupling is the significant one so as to deal with complexity and the continuous modifications of the deployments. It is obvious that any architecture should not be strictly dependant on specific technology. In this context, SOA design is not tied to any technology but can be implemented using Simple Object Access Protocol (SOAP) [116], RPC, Common Object Request Broker Architecture (CORBA), Web Services etc. Every technology has its own advantages and limitations. For instance, CORBA provides a rich development environment but requires to learn a new programming model and does not support interoperability as it is a tightly coupled architecture. On the other hand, the major advantages of Web Services are loose coupling and interoperability.

4.2.2 Web services

Web services [114] constitute a framework for building distributed applications. They have typically been used to build applications that either interact using a web browser or somehow related to the World Wide Web. But the technology that makes up Web services is not tied to the World Wide Web or the particular technology that typically is associated with it, such as web browsers.

A Web Service is a software technique designed to support interoperable M2M interactions over a network. It is an interface described in a machine processable format (specifically Web Services Description Language (WSDL) [117]. Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with Extensible Markup Language (XML) serialization in conjunction with Web-related standard.

Given the prevalence of the Web and its associated technologies, Web services have seen a tremendous adoption in the general purpose IT world in the past couple of years. All major programming languages provide libraries tailored to build web-service-oriented applications; hence, a large body of existing IT systems is built using Web services.

4.2.3 RESTful Web services and IoT

As mentioned before, the Web services concept plays a relevant role in enabling interoperable IoT-oriented communications over the Internet [118]. On this basis, two main categories of Web services are used: the REST-compliant Web services and the arbitrary Web Services (WS-*). The difference between the two models resides in their way of managing communications. While the first approach uses systematic and well-defined operations (e.g., GET, POST, DELETE), the second approach, instead, uses arbitrary operations (e.g., using SOAP). In this context, the choice of the suitable model to implement depends strictly on the use case. For instance, when dealing with ad-hoc services over the Web (so-called mashups) and IoT related use cases, the REST

model is meant to be an efficient solution [119] [120].

From the performance point of view, the RESTful compliant Web services are a convenient choice compared to WS* when it comes to IoT. In fact, the RESTful Web services are characterized by less overhead and parsing complexity while providing stateless interactions [121]. Besides, the fact that WS* supports only XML as an encoding pattern makes it unsuitable to be adopted in particular IoT scenarios, such as low power and low data rate sensor networks. On the other hand, REST affords more data format choices (e.g., plain text, XML and JSON) that make it a flexible model to be adapted to the task at hand. In particular, in IoT scenarios, the use of REST and JSON ensure higher performance implementations than WS* and XML [122] [123].

Another critical parameter to consider when choosing the suitable Web service paradigm for IoT is the software development aspect. In order to promote external developers communities in conceiving new IoT-based services/applications, providing a consistent software architectural style with accurate APIs is critical in adopting IoT services on a larger scale. In this context, developers prefer the REST architectural style as it is less complex to implement and use [124]. In fact, a number of IoT-based deployments uses the RESTful Web services model [125] [126].

4.2.4 Secure Web services in IoT

In IoT deployments, particularly the WoT architectural design, we expect to have services and, therefore, data exposed publicly over the Web. Considering the distributed nature of IoT deployments, ensuring secure data transmission is challenging. In such an environment where the infrastructure (i.e., IoT nodes) is geographically distributed with networks designers having limited control over the infrastructure (e.g., when it is contributed by volunteers), chances of malicious users to falsify the data being transmitted or even spoof IoT nodes is considerably high [127]. As the WoT paradigm aims to merge IoT with the Web, enabling Hypertext Transfer Protocol Secure (HTTPS)-based communications is a convenient choice. In fact, HTTPS is actually the de-facto protocol used in the Web that ensures peers identification and

prevents against communications sniffing and data manipulation [128].

To ensure secure communications between peers, Certification Authorities (CAs) tend to make the servers administrators follow a set of manual tasks while configuring their domains/servers. However, in the kind of IoT deployment we are targeting, the manual configuration of the servers (i.e., devices) certificates may become an error-prone task besides being time-consuming [128] due to the high-density of IoT networks. In this context, the use of an efficient mechanism, such as the Automated Certificate Management Environment (ACME) [129] protocol is essential in enabling secure data exchange in IoT and avoid manual configurations. Efforts are on the way to enhance the ACME Protocol by using a distributed trust mechanism based on Blockchain [130].

4.3 Technologies background

In this section, we give details about the different concepts we made use of to come up with our system. We introduce briefly the architecture of the OpenStack Domain Name System-as-a-Service (DNSaaS) subsystem (i.e., Designate) and the ACME protocol.

4.3.1 The OpenStack DNSaaS system: Designate

Designate is a multi-tenant DNSaaS service for OpenStack. It provides the capability to configure zones and DNS records within the OpenStack environment using REST APIs. Furthermore, this service provides a very handy solution to automate updates to DNS records based on other subsystems actions (e.g., Nova, Neutron).

Similarly to other OpenStack subsystems, Designate is composed of several components (blue system in Figure 40): the API endpoint (designate-api), the (centralized) controller (designate-central), an internal DNS server (MiniDNS or designate-mdns) used to manage down-stream, outward-facing DNS servers. Designate provides a flexible solution for Cloud administrators as it can be backed by a variety of open-source

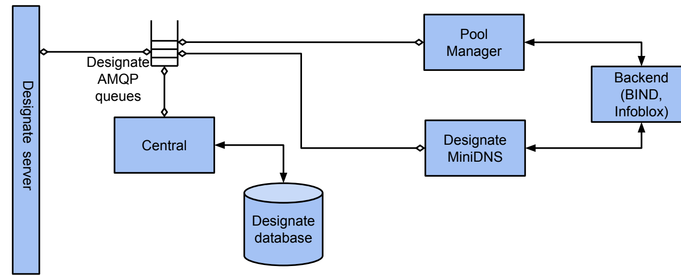


Figure 40: Designate subsystem architecture.

or commercial DNS servers, such as BIND, Infoblox, or PowerDNS. We mention that the backend choice and configurations are managed by the Cloud provider. The tenants (i.e., Cloud consumers), instead, do not have to mange the DNS configurations as they only make use the services Designate provides (create/manage zones and DNS records) through the exposed APIs. We describe here briefly the role of Designate's components:

- **Designate-api:** provides an OpenStack-native interaction interface using REST.
- **Designate-central**: it is the hub of activity. It coordinates all the communications between the different components and carries out the API requests. Furthermore, this component manages the persistent storage for Designate data.
- **Designate-mdns:** a small MiniDNS server used to communicate with other DNS servers over a standard DNS protocol.
- **Designate-pool-manager:** manages the states of the DNS servers that DNSaaS uses. It ensures the synchronization between the DNSaaS and the backend DNS servers.

4.3.2 Automatic Certificate Management Environment (ACME)

Digital certificates in Web Public Key Infrastructure (PKI) are widely used to authenticate domain names. CAs are trusted to verify that a certificate applicant represents legitimately the domain name(s) mentioned in the certificate. In the context of verification, Domain Validation (DV) certificates are the most common type.

The verification in the case of DV issuance consists of verifying that the requester has effective control over the domain. Of course, this is different from the process of verifying the real-world identity of the requester which is managed by other types of certificates, such as Organization Validation (OV) and Extended Validation (EV) certificates.

In deployments that require DV, CAs tend to use a set of ad-hoc protocols for certificate issuance and identity validation. The issuance process is managed by making the administrator to follow interactive instructions from the CA (e.g., Certificate Signing Request (CSR) generation, domain ownership verification, certificate download/installation). However, in particular scenarios like the one we are targeting, Web services for IoT, such a manual configuration of the certificates may become error-prone and a time-consuming task [131] that cause significant frustration and confusion considering the dimension of the deployments. ACME [129], as a recent protocol initially designed by the Internet Security Research Group (ISRG) for their Let's encrypt [132] non-profit CA, can solve the critical pain-point of managing, manually, X.509 certificates issuance. By using a client on the user-side server (e.g., IoT nodes), such as Certbot[16], ACME carries out the certificate issuance process without any human interaction. Let's encrypt CA has been increasingly adopted since a couple of years ago [132].

4.4 Exposing Cloud-enabled IoT-hosted services

This section describes the tunneling approach we conceived to expose, publicly, via the Cloud, TCP services hosted on IoT nodes deployed at the network edge. This feature is then leveraged as an infrastructure-level enabling mechanism to expose the IoT nodes hosted resources (i.e., physical/virtual sensors and actuators) by assigning to them publicly resolvable domain names (a detailed description of this system is reported in the next section).

As remote infrastructure, the deployed IoT nodes will be reachable over restric-

[16]https://certbot.eff.org

tive and even masqueraded IPv4 networks. In this case, the unique assumption that can (always) be considered valid is outgoing Web traffic being authorized; that is, only device-initiated TCP communications over standard HTTP/HTTPS ports are permitted. To cope with the constraint mentioned above, we opted for a standard TCP-based HTTP-borne full-duplex communication, namely WebSocket (WS) coupled with a reverse tunneling mechanism, i.e., IoT nodes will initiate the process of setting up the tunnel to the Cloud. Besides providing bidirectional flows between two ends, WS is a network-agnostic protocol by turning communications into standard HTTP(S) interactions. Any mechanism then that exploits WS can overcome the issues of reaching environments that block Web-unrelated traffic. An interesting feature that can be enabled using WS is establishing TCP tunnels over WS, a way to get client-initiated connectivity to any server/device-side service. For our system, we designed and implemented a suitable reverse tunneling over WS[17] solution as an approach to provide connectivity to any IoT node-hosted service.

Even though we opt for the WS protocol that adds additional overhead to the packets and requires a handshake to establish the client-server connection as a transport medium for our tunneling system, those parameters do not significantly affect the system performance. Indeed, authors in [133] outline that during long WS sessions, the impact of those parameters becomes insignificant just after few messages exchanges (see also subsection 4.6.3).

We depict in Figure 41 the design of the system as well as the process of a WS reverse tunnel (rtunnel) creation. To expose a TCP service (e.g., a Web server) hosted on a remote IoT node, the rtunnel client (which is pre-configured with the IP address of the rtunnel server) sends a WS connection request to the rtunnel server (i.e., the Cloud). In particular, the request specifies a TCP port to be used on the server-side. Once the rtunnel server receives the request, it brings up a TCP server listening on the port indicated, and a WS connection used as a control channel is established (purple arrow in Figure 41). When an external TCP client connects to the TCP server on the rtunnel server (i.e., Cloud), the rtunnel client and server manage this event by

[17] https://github.com/MDSLab/wstun

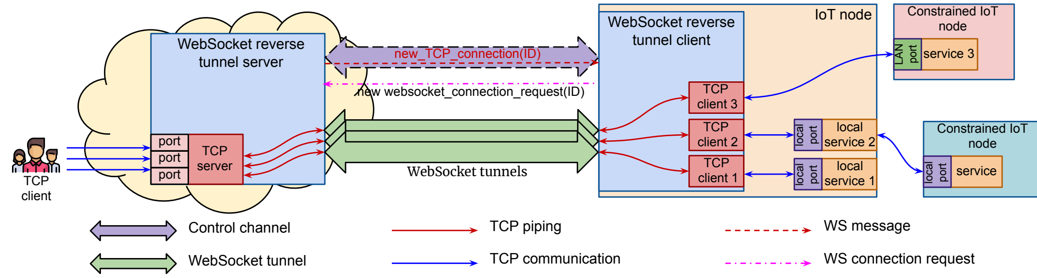


Figure 41: The WebSocket tunneling system.

instantiating a WS-based communication channel (green arrow in Figure 41), and the TCP session gets piped to it. On the rtunnel client-side, a similar mechanism is used by bringing up a TCP client connecting to the local (or remote) service involved and pipe afterwards the traffic to the tunnel.

Figure 41 highlights all the three scenarios the tunneling system may enable. A first scenario, as we mentioned before, is node-provided access to a service running on the node itself (`service 1` in Figure 41). To illustrate a use case, we can consider a Web server exposing the IoT node hosted resources (e.g., sensors and actuators). The second use case, similarly to the first one, the service runs on the IoT node itself, but it can forward/map requests to other constrained nodes behind it (`service 2` in Figure 41). A relevant use case we can mention is a proxy relaying HTTP requests to a constrained network (e.g., 6LoWPAN-based) using an HTTP-COAP proxy. Last and not least, the system can also provide access to services running on other devices deployed on the same LAN as the gateway (`service 3` in Figure 41). In this case, the gateway, a relatively powerful node capable of handling the complexity of setting up the WS tunnel; therefore, only applications flows are forwarded to the constrained device behind it.

It is worth mentioning that the tunneling approach can also be extended to expose UDP services (e.g., based on Quick UDP Internet Connections (QUIC) protocol). In fact, to expose a UDP service hosted on remote IoT nodes, we accommodate UDP flows on both sides (i.e., rtunnel client and server) to fit the WS tunnels TCP server/client and transit via the WS tunnel. Specifically, we use the Linux `Socat` tool that can establish bidirectional byte streams between two extremities (i.e., ports)

whatever the transport protocol used (UDP or TCP) and transfers data between them using TCP. Consequently, UDP packets on both sides (i.e., client and server) get adapted to fit the (TCP-based) WS tunnel. This way, a UDP flow coming from an external UDP client towards the Cloud gets tunneled and reaches services deployed at the network edge.

4.5 S4T Dynamic DNS system

In this chapter, we aim at introducing a novel approach to expose, to the Web, services running on IoT nodes deployed at the network edge. By assigning publicly resolvable domain names to the services hosted on the distributed IoT nodes, we can expose their resources (i.e., sensors and actuators) using REST APIs. We are the moving towards a more decentralized, yet developer-side uniform IoT ecosystem. In the following, we describe our OpenStack-based Dynamic DNS (DDNS) system that uses IoTronic and the DNSaaS subsystem, Designate.

4.5.1 Overview of the Stack4Things Dynamic DNS system

We report in this subsection an overview of our tunneled reverse proxying approach capable of assigning globally resolvable domain names to services deployed within IPv4 masqueraded networks. In particular, the approach uses only one publicly registered domain name to make the distributed services identified using sub-domains of the public one. That says, no public IP or public domain name associated with the IoT node is required.

To conceive our system, we are considering, as mentioned before, that the IoT nodes are typically deployed behind NATs and firewalls; therefore, they do not have routable public IP addresses. To expose the IoT nodes hosted services, and by transitivity the physical or virtual sensors/actuators they may host (using for example a Web server), our approach uses Designate to cope with the management of DNS records (which are sub-domains of the public one) associated with the edge-based services. To route requests based on the Uniform Resource Locators (URLs) indicated,

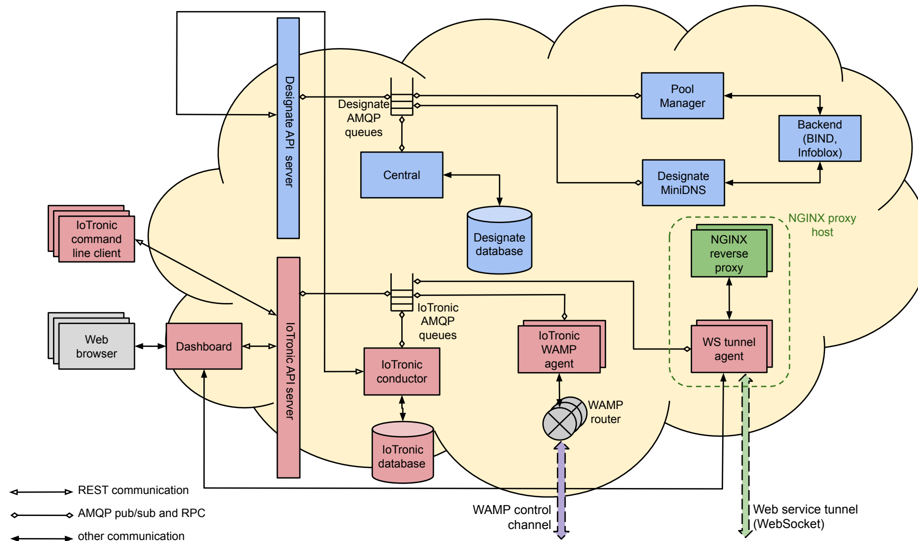


Figure 42: DDNS Cloud-side system architecture.

IoTronic and LR, together with two NGINX reverse proxies[18] deal with requests forwarding (see Figures 42 and 43). In particular, for each sub-domain created and assigned to a service, IoTronic and LR manage the instantiation of a reverse WS tunnel between the Cloud and the IoT node (see Section 4.4). Afterward, clients requests in the destination of the service are forwarded through the WS tunnel using the NGINX reverse proxies rules managed by IoTronic and LR: the user does not have to do any configuration.

Regarding the DV certificate issuance and validation, once the reverse WS tunnel gets created and the two NGINX reverse proxies configured, LR manages the X.509 certificate issuance and validation using the ACME-based Let's Encrypt CA client, namely Certbot. Clients then, such as Web browsers and mobile applications, can communicate using HTTPS with the edge-based services running on the IoT node. Specifically, a client request is sent to the Cloud NGINX reverse proxy that manages, based on the URL indicated in the request (specifically, the sub-domain part), the forwarding/routing of the request through the suitable WS tunnel to reach the IoT node/service concerned.

[18]https://www.nginx.com

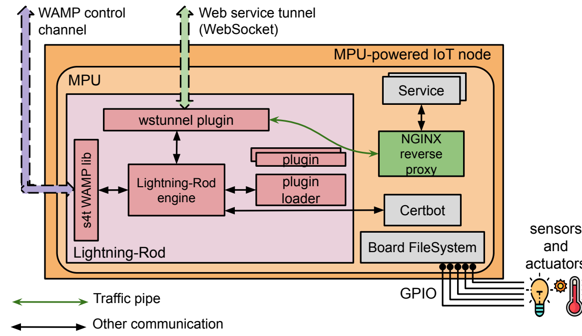


Figure 43: The device side Web services system.

4.5.2 Workflow of exposing a service

In the following, we report a detailed description of the workflow when a user wants to expose, publicly, a service (a Web server in this example) running on an IoT node. The full domain name we are considering to be assigned to the Web server is *web-server.node-A.example.com*. Consequently, the registered public domain used is *example.com*, whereas the rest of the domain name, i.e., *node-A* and *web-server* are managed by our DNS system to identify the IoT node and the service concerned, respectively (an IoT node can host multiple services). Regarding the Cloud NGINX reverse proxy, we consider that the host where it is running has as an IP address 1.1.1.1, while the Web server hosted on the IoT node runs on port 9000. We assume that the IoT node has already gone through a set of verification processes (e.g., authentication) required by S4T; thus, it is registered and connected to the Cloud. The following list of sequences takes place when exposing the Web server, with low-level operations as depicted and numbered in Figure 44):

1. The user send a request to expose a service (e.g., Web server) running on a specific IoT node using the OpenStack dashboard or the CLI. In particular, the user chooses the service name (in this case, named *web-server*), DNS zone that indicate the IoT node where the server is running (i.e., *node-A*), and the port on which the Web server is listening (i.e., 9000).

2. The dashboard/CLI sends a REST request to the IoTronic API server that pushes a new message into the AMQP queue.

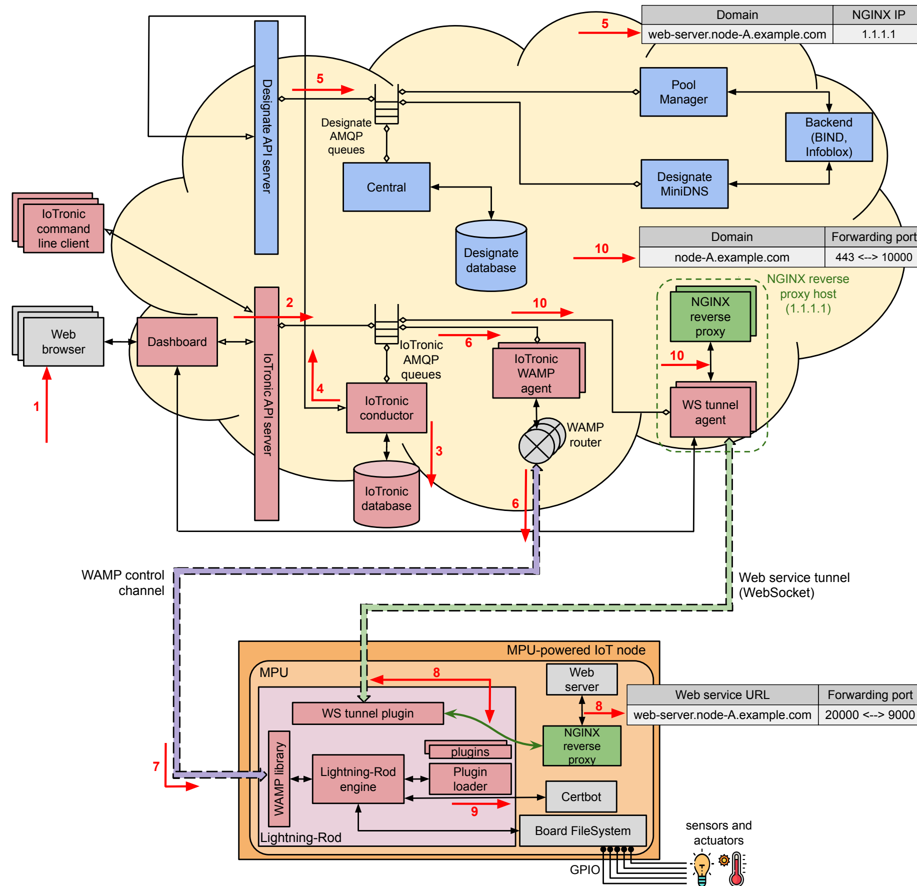


Figure 44: A detailed workflow description of exposing a service (a Web server in this case) hosted on an edge IoT node.

3. The IoTronic conductor gets the message from the queue and does a check using its database. Specifically, IoTronic verifies if the IoT node is registered and online. It also looks up the WAMP agent and the WS tunnel agent managing the IoT node concerned (the WS tunnel agent is co-hosted on the same machine where the Cloud NGINX reverse proxy is deployed. it has as IP address 1.1.1.1).

4. IoTronic interacts with the Designate API server to request the creation of a type A DNS record using the IP address of the NGINX reverse proxy from step 3 (i.e., 1.1.1.1) and the information provided by the user in step 1 (i.e., the sub-domain names: *web-server* and *node-A*).

5. Designate creates the DNS record within its backend environment.

6. The IoTronic conductor sends an RPC to the WAMP agent managing the IoT

node to start the configuration process. Subsequently, the IoTronic WAMP agent sends a WAMP-based RPC to the NGINX reverse proxy running on the IoT node to instantiate a WS tunnel to the Cloud. A random port number is generated and sent as an argument of the RPC to create the tunnel. In our scenario, we consider as random port number 10000.

7. The IoT node receives the RPC through the WAMP library.

8. The LR agent manages the setup of the (reverse) WS tunnel by generating a random port number (20000 in our scenario) and taking into consideration the port number received as an argument of the RPC (i.e., 10000). Next, LR configures the NGINX reverse proxy to forward requests in destination of the URL *web-service.node-A.example.com* to the port on which the Web server is listening (specified by the user in step 1, 9000 in this case)

9. LR manages, using the local Certbot daemon, the issuance of the DV certificate from the Let's Encrypt CA.

10. The IoTronic conductor interacts with the WS tunnel agent managing the IoT node (using an RPC) to configure the Cloud-based NGINX reverse proxy. Specifically, the proxy is used to forward requests in the destination of *node-A.example.com* to the WS tunnel instantiated in step 6. That says, requests reaching the Cloud NGINX reverse proxy on port 443 and having as destination *node-A.example.com* are forwarded to port 10000 (i.e., the port on which the WS tunnel is running).

To illustrate a functional workflow that uses the mechanisms described before, we report in subsection 4.6.2 a detailed workflow of a request being routed using the tunneling system.

4.6 Implementation and experimental results

In this section, we provide an online accessible testbed powered by the S4T DDNS system. We also report a set of experiments results to assess the performance of the

approach.

4.6.1 Testbed description

To prove the feasibility of our approach, we used a Raspberry Pi board to host Web server running on a Docker container with the GPIO sysfs mounted on it (i.e., an I/Ocloud Virtual Node). Thus, we can expose publicly a set of hosted sensors/actuators as depicted in Figure 45. The OpenStack environment, including our IoTronic system, is hosted at the Department of Engineering, University of Messina, Italy. The Raspberry Pi runs the device-side LR agent and hosts a Web application container that uses the built-in Flask Web server running on port 5000 (we use the Docker port mapping capability). We used the approach presented in Section 4.5 to expose the Web server running inside the container. As URL, we choose *https://wot.rasp-univ.iot.felooca.eu* under which */temperature*, */humidity*, */redled* and */greenled* are made available as resources. We remind that the choice of the URL is up to the user except for its public part (in this case, *felooca.eu*). Other services hosted on the same board are exposed as well, such as a video streaming from a Web camera using *streaming.rasp-univ.iot.felooca.eu* as URL and a Node-RED instance[19].

We mention here that the registered (and globally resolvable) domain name the approach uses is *felooca.eu*. The *iot* sub-domain was created for management purposes as the domain (i.e., *felooca.eu*) is used for production by the smartme.IO[20] spin-off company. The *iot* sub-domain does not affect the workflows described before: it is transparent with regard to the approach here presented. To be aligned with the descriptions presented in Section 4.5, we can consider that our public domain is *iot.felooca.eu*.

By enabling the use of globally resolvable URLs associated with geographically

[19] We exposed three services that the readers can access:

- **The web page**: https://wot.rasp-univ.iot.felooca.eu
- **The video streaming**: https://streaming.rasp-univ.iot.felooca.eu/?action=stream/
- **A Node-RED instance**: https://node-red.rasp-univ.iot.felooca.eu

[20] https://smartme.io/

distributed services deployed at the network edge, we are able to create mashups based on distributed Web services. For instance, we used the video streaming URL to incorporate the streaming on the Web page. The mechanisms provided by the system make user agents, for example, web browsers, able to interact with the board-hosted resources using HTTPS. The reader can access the following URL: *https://wot.rasp-univ.iot.felooca.eu* that point out the Flask Web server hosted on the Raspberry Pi. In this deployment, all the requests made to get sensors values (i.e., temperature and humidity), as well as the LEDs status, are HTTPS-based GET requests. The Linux command-line tool, `curl`, can also be used to retrieve the value of a metric from a resource, for example, temperature: *curl -X GET -H "Accept: application" https://wot.rasp-univ.iot.felooca.eu/temperature*. The reader can also interact, in real-time, with the two LEDs using the corresponding Web page widgets by sending HTTPS POST requests under the hood.

4.6.2 Functional workflow

We report in the following the functional workflow when a Web client requests the value of the temperature sensor. The URL we consider is the one mentioned before and made available over the Web: *https://wot.rasp-univ.iot.felooca.eu* and the resource involved is */temperature*. The complete workflow is reported in Figure 45 (we skip the different TCP/TLS handshakes for a matter of simplicity and to make the workflow easier to grasp):

1. The client, a Web browser in this case, sends a DNS resolver query about the URL (i.e., wot.rasp-univ.iot.felooca.eu) to the ISP public DNS server.
2. The public DNS server sends a response back to the client about the *felooca* domain name. The response contains the public IP address of the S4T Cloud DNS server.
3. The client sends a new DNS request to the S4T DNS server.
4. The DNS server sends a response back to the client with the IP address of the

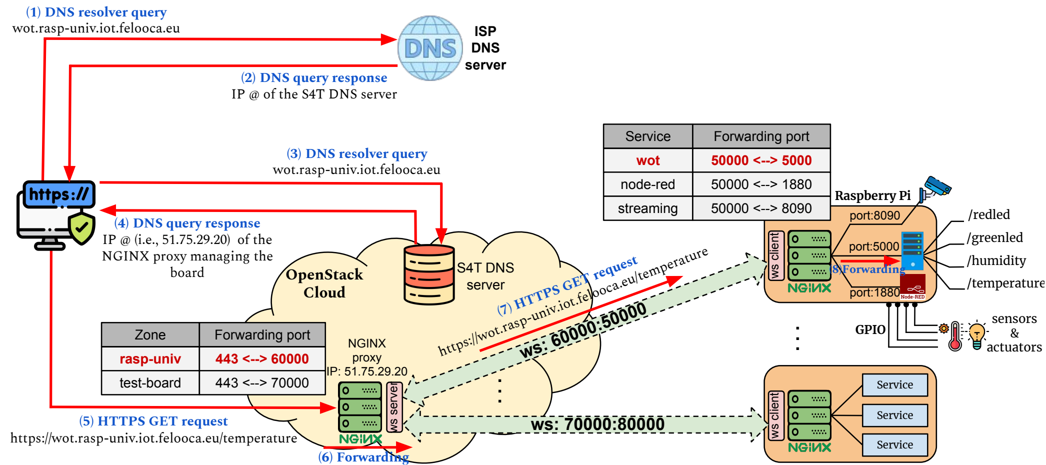


Figure 45: The Stack4Things-based routing mechanism.

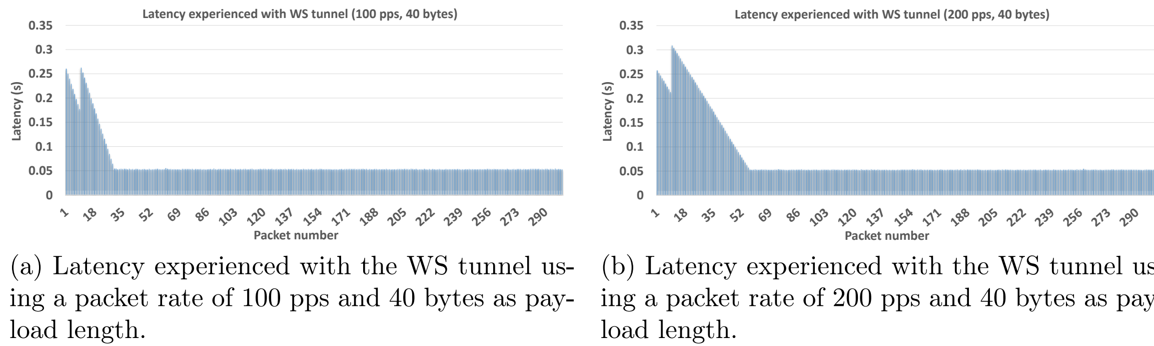
NGINX reverse proxy managing the board concerned (in the testbed we are making accessible online, the proxy IP address is 51.75.29.206).

5. The client sends an HTTP GET request to the IP address specified by the DNS server from the previous step (i.e., 51.75.29.206).

6. Once the NGINX reverse proxy receives the HTTP request, it checks the URL mentioned. Based on the sub-domain specified (i.e., *rasp-univ*) that indicates the board, the NGINX reverse proxy forwards the request through the appropriate WS tunnel connecting the Cloud to the board. Specifically, in our online testbed, requests received on port 443 and having as URL *https://wot.rasp-univ.iot.felooca.eu* are forwarded via the WS tunnel running on port 60000.

7. The request reaches the board NGINX reverse proxy through the WS tunnel.

8. The NGINX reverse proxy checks the URL of the request specifically, the service name (i.e., *wot*). Based on its forwarding rules (created when exposing the Web server, see Section 4.5.2), the proxy forwards the request to the port on which the Web server is listening. In our scenario, the service is named *wot* and runs on port 5000. As a result, the request reaches the Web server, and the response (value of the */temperature* resource) travels back the same way.



(a) Latency experienced with the WS tunnel using a packet rate of 100 pps and 40 bytes as payload length.

(b) Latency experienced with the WS tunnel using a packet rate of 200 pps and 40 bytes as payload length.

Figure 46: Latency experienced with the WS tunnel when using packets with 40 bytes of payload.

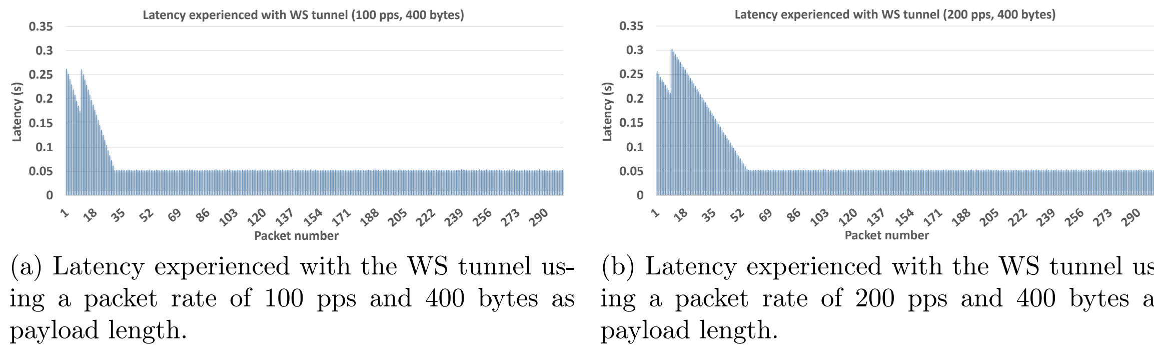
4.6.3 Performance evaluation

In this subsection, we evaluate firstly the performance of the tunneling approach we implemented. Then, we present the results of deploying the whole solution (i.e., WS tunnel client and NGINX) on an IoT node.

To assess the performance of the tunneling mechanism, we used two VMs as UDP client and server communicating through a WS tunnel. Each of the VMs has two vCPUs and 2 GB of RAM. The VMs are hosted on a 2020 Intel i5 MacBook Pro while being timely synchronized using Network Time Protocol (NTP).

To evaluate the impact of the WS tunnel on the latency it may introduce, we fixed the latency between the 2 VMs (using the virtual bridge interface) at 50 ms. We measured then the delay between the timestamp when a packet with 40 bytes of data is sent by the client and the timestamp when the same packet reaches the server. For the sake of clarity, we state here that the latency we intend to evaluate is a one-way measure. Specifically, we used UDP as it brings more flexibility and control over the testbed since the packet sending rate can be controlled with higher granularity.

Figures 46a and 46b depict the results of our experiments at 100 and 200 packets per second (pps), respectively. The packet number (x-axis) represents the sequential number of the packets received by the UDP server at a given packets sending rate. We mention here that to accommodate the UDP traffic sent/received by the client and



(a) Latency experienced with the WS tunnel using a packet rate of 100 pps and 400 bytes as payload length.

(b) Latency experienced with the WS tunnel using a packet rate of 200 pps and 400 bytes as payload length.

Figure 47: Latency experienced with the WS tunnel when using packets with 400 bytes of payload.

make it tunneled through the TCP WS tunnel, we used the `Socat` tool as discussed in Section 4.4. As reported in Figure 46a, the latency stabilizes at around 50 ms (i.e., the latency fixed between the two VMs) after some packet transmissions. This result shows that the WS tunnel's impact on latency is negligible during long sessions.

At the beginning of the communication, the higher latency value is due to the three-way TCP handshake used to set up the WS tunnel (the second peak is attributed to the TCP windowing). Indeed, while the WS tunnel is being created, the UDP client keeps sending the packets at a fixed generation rate (i.e., 100 pps); those packets got queued and, therefore, delayed. For the packets generated at 200 pps (Figure 46b), similarly, the first train of packets was affected by a higher latency, whereas a latency of approximately 50 ms identifies the steady-state response. In this case, we notice that a higher number of packets was affected by the WS tunnel setup (i.e., TCP handshake), as the packets generation rate is higher than in the first case; thus, more packets were delayed in the queue. Albeit higher rate, the latency always stabilizes around the same value of 50 ms (i.e., the latency we fixed between the two VMs). It is worth mentioning that even though we depict only the first 300 packets in the graphs, we run each experiment for 10 minutes. For all cases, the latency remained constant and close to the values being shown. Regarding the performance of the tunnel vis-à-vis the packets size, we used larger payloads. Specifically, as reported in Figures 47a and 47b, we used packets with a payload 10 times larger than the first case (i.e, 400 bytes long) and we got the same results.

	Packets with 40 bytes of payload (CPU usage in %)			**Packets with 400 bytes of payload (CPU usage in %)**		
Packets generation rate (pps)	**WS tunnel client**	**Socat**	**Total**	**WS tunnel client**	**Socat**	**Total**
100	2.48	0.38	2.86	2.6	0.4	3
200	4.37	0.76	5.13	4.31	0.71	5.02
300	5.2	1.12	6.32	5.23	1.09	6.32
400	5.78	1.4	7.18	5.62	1.42	7.04
500	6.18	1.66	7.84	6.11	1.58	7.69

Table 13: WS tunnel client and `Socat` CPU usage with different packets' payload lengths.

To overcome the tunnel's instantiation queuing issue at the beginning of the communications, a solution that would be conceivable when exposing services is to instantiate the WS tunnels more proactively (i.e., once the user expose the service and before receiving any request), anticipating the reception of the messages and keeping TCP sessions alive to avoid the three-way handshakes.

Another aspect we evaluated is the CPU usage since the WS tunneling mechanism we are introducing is meant to be implemented on IoT nodes. We report in Table 13 the CPU usage of the WS tunnel client. As we can notice, the packet size does not impact the CPU resource usage as the results when using packets with 40 and 400 bytes payload lengths are aligned. We mention that the CPU usage of `Socat` is reported as well since we are tunneling, in this case, a UDP traffic. The case when using TCP-based flows `Socat` is not required (see Section 4.4).

We conducted a set of other experiments to evaluate the performance of the WS tunnel and NGINX reverse proxy hosted on the IoT node. In particular, we measured the NGINX reverse proxy and the WS tunnel client CPU and RAM usage using a Raspberry Pi 3 Model B+ (Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU with 1 GB of RAM). To generate GET HTTP requests, we emulated the users by means of the open-source load testing tool, Locust[21]. We configured each emulated user to send one GET request per second (using the *constant_pacing* function). We report in Figure 48 the CPU usage of the NGINX reverse proxy and the WS tunnel client. The CPU usage of the built-in Flask Web server used is reported as well (we opted for

[21]https://locust.io

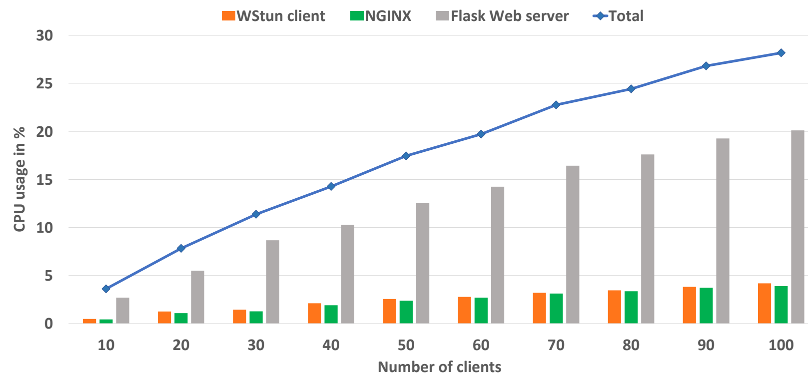


Figure 48: CPU usage on the Raspberry Pi 3 Model B+.

the Flask built-in Web server for a matter of implementation simplicity). Of course, other Web servers (e.g., Apache) can be used to improve the server performance. As shown in the figure, when generating ten requests per second by ten users, the WS tunnel client and NGINX uses 0.91%. This value keeps increasing quite linearly to reach 8.07% when reaching 100 users. Regarding the RAM usage, the value remained constant all along with the experiments while varying the number of users. Practically, the NGINX reverse proxy and WS tunnel client combined use exactly 6.6% of the total amount of 2 GB of RAM available.

Chapter 5

Deviceless: an approach extending Serverless to IoT deployments

5.1 introduction

Over the last few years, industry and academic research communities have proposed many IoT applications. As discussed in the previous chapters, to deal with IoT data management and processing, most of the solutions rely on Cloud platforms [134]. These Cloud-oriented approaches adopted in IoT data management can be framed into the data-centric category [46] as the only operations provided are data manipulation ones. Albeit the massive amount of resources the Cloud offers (e.g., compute and storage), Cloud platforms consider often the IoT devices as only data providers uploading data towards datacenters. The Cloud role is then restricted to be a scalable sink dealing with data processing. This management approach has several drawbacks stemming from the non-real-time access to IoT data and the inability to personalize the business logic running on the IoT nodes.

To deal with this short comings, we introduced a set of enabling mechanisms for the I/Ocloud computing paradigm that aims at providing IoT infrastructure as IaaS Cloud resources (a paradigm aligned with the IoT-as-a-Service [135]). The I/Ocloud view can be enabled using virtual instances (i.e., containers or VMs) instantiated either at the Cloud level or at the network edge to meet the requirements of IoT applications. Besides, we also provided virtual networking services for the remote containers and bare-metal IoT nodes to enable the users to set up customized networking topologies regrouping them. In this IaaS model, a Cloud user has low-level access to the virtualized I/O resources of the physical device. However, in some cases, this low-level access is not required. Besides, in the IaaS computing model, the Cloud

user pays for the virtual instance (i.e., a container or VM) and other resources required to run applications from when they provision those resources until the time the customer explicitly decommissions them. The provisioning periods of the instances (i.e., VMs and/or containers) can be then long although the tasks to handle can be short in time. Indeed, even though a process should be executed only few times in response to specific exceptional events, in the IaaS model, the developer has to provision an instance continuously even if no process is running.

On the other hand, with the rise of the Cloud computing paradigm, we have seen a transition from buying and managing bare metal servers to using instances (in the form of VMs or containers) hosted in a Cloud datacenter. Recently, we are even shifting from Cloud-based instances to the Serverless computing model [136] where all traces of the actual server platform have disappeared. Specifically, the application developer still writes the server-side logic yet, unlike traditional architectures, it runs on stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by the Cloud provider. The developers can focus then only on the business logic of their applications whilst delegating all infrastructure management tasks (i.e., scalability, provisioning, etc.) to the Cloud provider; thus leading to a new utility computing scheme, namely Function-as-a-Service (FaaS) [137]. In this computing model, the functions to be deployed have to be short running processes to not exceed a delay fixed by the Cloud provider (e.g., for AWS Lambda, a function can be configured to run up to 15 minutes per execution [138]).

In the context of enabling seamless interactions with IoT resources (e.g., sensors and actuators), our approach aims at extending the Cloud Serverless paradigm [139] towards the network edge to use it on top of a (shared) IoT infrastructure. Application developers can then make use of the IoT resources (i.e., sensors and actuators) in a Serverless-like fashion without managing the infrastructure or the used communication protocols. Furthermore, they will be charged for actual usage at a millisecond granularity. We refer to this new computing model as Deviceless.

In addition to IoT-as-a-Service provided by the I/Ocloud paradigm, a user can use

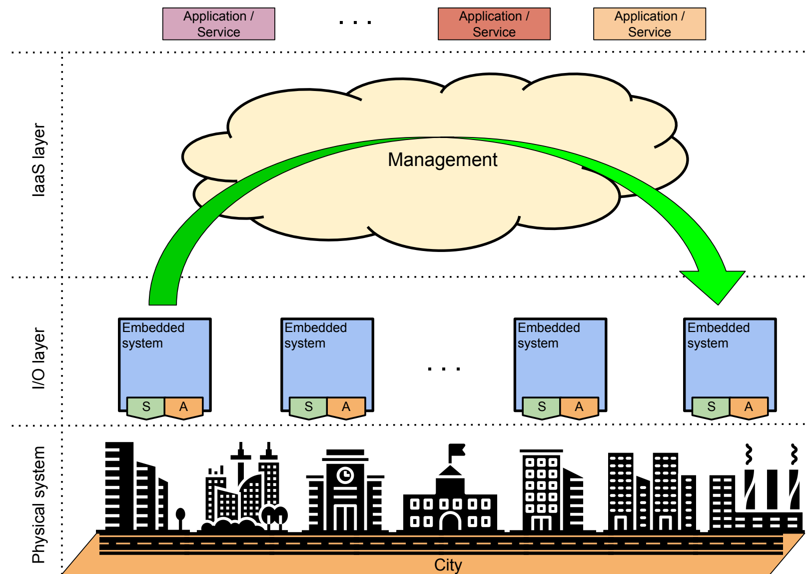


Figure 49: A Software Defined City as closed-loop system.

Deviceless based on a event-programming model without resorting to VNs provisioned for long periods (when not needed). This programming model can drive significant infrastructure cost savings. In fact, if we consider the AWS Serverless offering, a tenant has to pay around 21 USD per month for a Linux-based VM with 1 CPU and 2 GB of RAM located in the Frankfurt, Germany's AWS datacenter [140]. On the other hand, if the tenant opt for AWS Lambda (the AWS serverless computing platform), he/she has to pay only 0.0105 USD for 5 million function's executions (the function we consider has 1 ms a duration and requires 128 MB of memory) [141]. In an IoT context, the event programming model fits very well the kind of application we may need and for closed loop systems, such as configuring triggers for a range of (dispersed) actuators based on sensing activities from geographically distributed sensing resources (see Figure 49).

It is worth mentioning here that the objective of the Deviceless paradigm is to simplify the way of conceiving IoT applications: the software engineering aspect. We do not target issues related to time-sensitive applications. Our view of the Deviceless aims to make it usable in conjunction with code deployed based on different coding styles, such as microservices or monoliths. Therefore, we aim to provide Deviceless and Serverless using the same platform to achieve complete integration between the

two computing models.

In this chapter, we introduce our Deviceless approach to explore the Serverless concept in an IoT scenario. Its implementation is based on our S4T framework and two OpenStack-based subsystems, Qinling and Zun, that have been customized for this purpose. Analogously with how the Serverless paradigm exempts users from managing/operating the infrastructure, the S4T Deviceless abstraction model dispenses developers from managing the IoT infrastructure and interact with remote sensors/actuators using short running atomic functions.

5.2 The Serverless paradigm

5.2.1 Before Serverless

To build a Cloud-based Web application, the developers have to deal with the management of the instances hosting the service on the Cloud. Typically, the IaaS cloud computing model is used. As such, the developers or the infrastructure team are responsible for managing and provisioning the infrastructure. There are a few issues with this:

- Users (i.e., the Cloud consumers) are billed for keeping the server up even when it is not serving any request.
- The Cloud consumer is responsible of managing his/her server and its resources.
- The time to market can be significant due to infrastructure management [142].

For individual developers and smaller companies, dealing with infrastructure provisioning could be hard and a time consuming task. These duties usually distract developers from their main job of building and maintaining the service/application. For bigger companies and organizations, developers are not responsible of the infrastructure management. Infrastructure provisioning tasks are often delegated to specialized teams. However, since the developers cannot deploy applications without

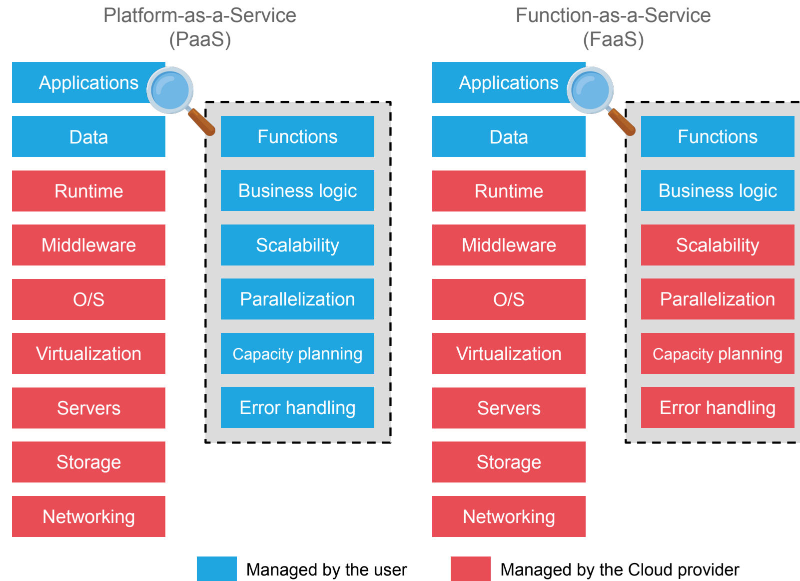


Figure 50: Management responsibilities in PaaS and FaaS .

cooperating with the infrastructure team, this can lead to slowing down the development time. Recently, to deliver applications and services at high velocity, we are seeing a new kind of developers managing both the software and the infrastructure, a trend known as DevOps [12]. To push even this concept beyond and extremely shorten applications delivery time, we are shifting now to NoOps [143] (a paradigm based on Serverless) were all the infrastructure management tasks are delegated to a third party (e.g., a Cloud provider) while the developers, instead, focus only on applications logic.

5.2.2 Serverless computing

Serverless computing (or serverless for short) [136], is an execution model of cloud computing in which the Cloud provider allocates machine resources on demand, taking care of the servers on behalf of its customers.

In traditional application deployments, server computing resources are a fixed recurring cost regardless of the amount of computing work actually performed by the server. In a serverless IT deployment, the Cloud client only pays for the use of the service; there is never a cost associated with downtime or inactivity periods: it can

be considered as a form of utility computing.

Serverless is a misnomer in the sense that servers are still used by the Cloud providers to run code for developers. However, developers of serverless applications are not concerned with capacity planning, configuration, management, maintenance, fault tolerance, or scaling of containers, virtual machines, or physical servers. Developers can add code, build back-end applications, create event-handling routines, and process data, all without worrying about servers, virtual machines, or the underlying computing resources, as the hardware and the infrastructure configurations are all managed by the supplier (see Figure 50). The code that is sent to the Cloud provider for execution is usually in the form of functions.

In the recent few years, the Serverless Cloud computing model has been rapidly adopted in the IT field since its first appearance in 2014 with AWS Lambda[22]. Most of the Cloud providers, such as Microsoft and Google have introduced comparable Serverless/FaaS services in their commercial offerings (i.e., Azure Serverless[23] and Google Cloud Functions[24], respectively). Besides, other opensource solutions has been developed, such as Apache OpenWhisk[25], Kubeless[26] and Fission[27].

5.2.3 Serverless for Edge computing

The Serverless computing model provides a set of attractive benefits from the developer perspective [144]. With the emergence of new services, and to meet their requirements in terms of, for example, latency and bandwidth usage, solutions based on edge computing have been adopted [145]. Furthermore, adopting the Serverless paradigm at the network edge can be an efficient approach in a set of scenarios [146] [147]. Nevertheless, extending the Serverless computing model to cover edge deployments is not a straightforward process and brings a set of particular challenges,

[22]https://aws.amazon.com/lambda/
[23]https://azure.microsoft.com/en-us/solutions/serverless/
[24]https://cloud.google.com/functions
[25]https://openwhisk.apache.org
[26]https://kubeless.io
[27]https://fission.io

such as resource pooling and infrastructure provisioning/management [148] [149]. Besides, when considering IoT deployments, extending the Serverless to cover such distributed environments can be more challenging due to networking issues [147].

Unlike Cloud-based deployments where the execution infrastructure is deployed within the same datacenter (i.e., same physical network) and thus, servers connectivity is taken for granted, IoT scenarios, instead, are complex and hard to manage. Indeed, IoT deployments are composed of geographically distributed nodes deployed, most of the time, behind networking middleboxes (e.g., NATs and Firewalls).

Recent efforts are in progress to expand applicability of Serverless to cover IoT gateways and devices, such as Amazon Greengrass [78] and Azure IoT Edge [77] that provides edge-based runtimes dealing with IoT data processing. However, these solutions are not real extensions of Serverless but just an extension of the Cloud computing paradigm [148]. Furthermore, these proprietary systems makes the users dependent to their platforms; thus, increasing vendor lock-in issues [136]. Exploiting IoT resources (i.e., sensors and actuators) in a Serverless-like fashion can be a fruitful for the developers as it exempt them from all infrastructure management duties. An interesting platform extending Serverless to the IoT ecosystem is OpenWhisk-Light (OWL)[28]. Yet, the solution is limited in sense that the functions deployed on the IoT nodes triggers local actions based on only local detected events (i.e., no possible interactions with other nodes/instances). For instance, the platform can not trigger actions on an IoT node based on an event happening on another device or the Cloud; hence leading to limited applicability of the solution. In the same spirit, the Kappa system [150] enables the developers to conceive flow-based workflows in a Serverless-like fashion using both Cloud-based instances and IoT nodes.

In our Deviceless view, we aim to abstract the hardware layer of the IoT nodes. In particular, we would like to make the developers able to interact with IoT resources (sensors and actuators) in a Serverless-like way through stateless and personalized atomic functions. In particular, a code that uses Deviceless must behave as a Cloud

[28]https://github.com/kpavel/openwhisk-light

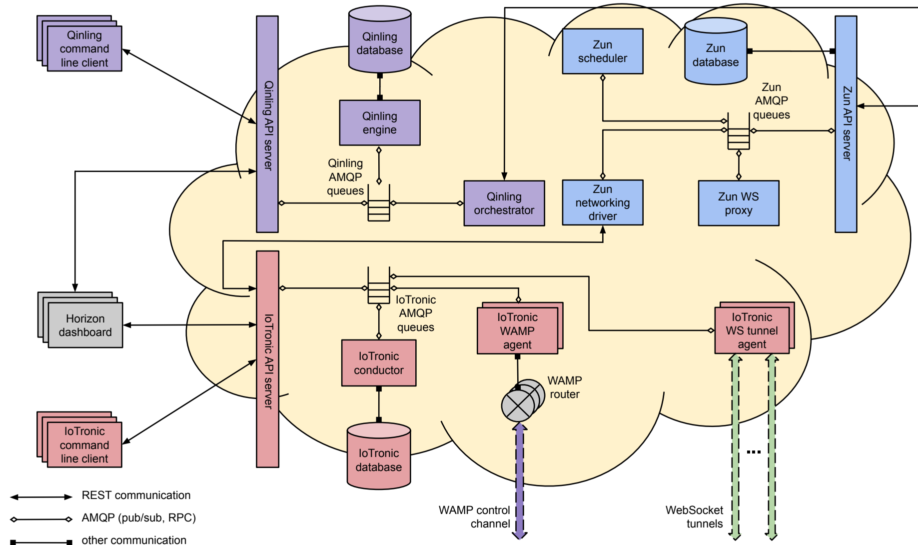


Figure 51: The Cloud-side Deviceless system architecture.

Serverless native code when it comes to interacting with any kind of application: microservices, monolithic or even Serverless applications.

5.3 The OpenStack FaaS subsystem: Qinling

Qinling (violet subsystem in Figure 51) is an OpenStack service that aims at providing a platform to support Serverless functions (like AWS Lambda). The Qinling system is highly flexible as it can be used with different Container Orchestration Engines (COEs), such as Kubernetes and Docker Swarm. The Qinling system is composed of:

- ***Qinling-API:*** represents the entry point of the interactions with Qinling. It exposes a set of REST APIs through which, users or other OpenStack services can communicate with the system (e.g., to create runtimes and execute functions). It handles the received request by routing them either to the Qinling-engine or the Qinling-orchestrator.

- ***Qinling-engine:*** it is the core subsystem that handles communications between the Qinling components as well as managing backend operations (e.g., pods instantiation, function creation/execution). It also represents the unique

entry point to the database where metadata about functions and their containers is maintained.

- ***Qinling-orchestrator:*** it is the component responsible for selecting the best candidate where to provision a runtime (e.g., based on CPU/RAM available) and scaling up/down (through the interaction with a COE) the number of containers deployed based on the number of requests received.

In a nutshell, Qinling is meant to instantiate the environment required to create runtimes end then execute users' functions. Specifically, when Qinling receives a request to create the runtime for a function, it instantiates (using Zun, see blue subsystem in Figure 51) the containers needed to execute the function. In particular, Qinling creates three containers used for different purposes (the environement composed of these containers is called a *capsule*[29]) :

- ***Runtime container:*** it is the isolated environment where users' functions get executed. Qinling supports three runtimes namely, Python2, Python3, and Node.js.
- ***Sidecar container:*** it is the container where the functions required packages are downloaded/stored. This container is used subsequently to provide these packages for the runtime container.
- ***Pause container:*** the container that ensures network reachability for the two containers mentioned above since is the only one attached to the network.

5.4 The Deviceless system description

In this section, we describe the architecture of the Deviceless system architecture and workflows to execute functions at the network edge. The system is based on IoTronic/LR and the two OpenStack Cloud-oriented subsystems Zun and Qinling that have been customized to extend their (limited) capabilities and deal with IoT

[29]the capsule is equivalent to the pod concept in Kubermetes

deployments. A major modification, when compared to typical Cloud-based Serverless deployments, is that, in our case, we split the Zun components between the Cloud and the devices deployed at the network edge. In our approach, we consider the (remote) IoT nodes as a computing infrastructure instead of the Cloud-based compute nodes. Therefore, to make this infrastructure reachable by the Cloud subsystems, we used suitable mechanisms to deal with NATs and firewalls traversal. Zun and Qinling can then manage containers and functions deployed at the network edge. In the following, we report details about the integration of Zun and Qinling with IoTronic to cope with IoT environment constraints.

5.4.1 Containers orchestration

In typical OpenStack Serverless deployments, Qinling uses non-compatible OpenStack COEs (e.g., Kubernetes and Docker Swarm). In such deployments, the integration of Zun compute nodes as part of Kubernetes/Docker Swarm clusters is not straightforward and requires manual configuration by the administrator. However, this kind of (manual) configuration in an IoT context would become a hurdle considering, on the one hand, the large number of IoT devices that should be managed, and on the other hand, the high dynamicity of IoT deployments (i.e., adding new devices to deployments or removing others). In such a situation, having an automated mechanism to include/remove into/from the COE cluster can bring more flexibility for the system. To deal in our situation with this limitation, we have designed for Qinling an OpenStack-compliant COE based on Zun; therefore, a compute node (i.e., an IoT device in our case) will be integrated automatically within the COE cluster.

The second aspect that has been extended in Zun is related to containers reachability. Indeed, if we consider a typical OpenStack deployment, containers reachability within a datacenter is assured by the overlay networking services provided by the networking subsystem, Neutron. Nevertheless, for instances (i.e., containers) deployed outside datacenters, as is the case for the kind of deployments we are targetting, containers reachability can not be handled by standard Cloud mechanisms [14]. Indeed, IoT nodes are often deployed behind NATs and firewalls. We developed then for Zun

a new networking driver that uses IoTronic in order to make the remote containers reachable. Specifically, the new driver uses WebSocket as a transport channel with a port forwarding capability provided by the Cloud. Therefore, requests reaching the Cloud on a specific port will be forwarded through the WS tunnel to an associated container (i.e., each port is associated with a remote container).

Another extension to Zun, specifically on its scheduler, has been done. In our Deviceless view, a user will request the setup of a runtime then the execution of a function on a specific IoT node to get, for example, the value of a sensor (or even do a preprocessing of this value). The standard Zun scheduling policies do not provide this kind of control to specify a particular compute node to instantiate a container (a set of filters instead are used, such as RAMFilter, CPUFilter, LabelFilter, etc.). We then added a new policy based on the name/id of the compute node called *HostnameFilter*.

5.4.2 Functions executions

The Deviceless system entry point is Qinling. To create a runtime on a specific IoT node or group of nodes then execute functions on that/those runtime(s), a Cloud user interacts with Qinling (i.e., he/she has to make the selection). Qinling, afterwards, interact with Zun (to create the containers) and IoTronic to provide network reachability for the containers. To enable the selection of IoT nodes in Qinling using their hostname/id or labels, we added a new selection parameter called *NodeName* and *nodeSelector*. The Qinling uses the hostname/id and labels to send the request to Zun that uses in its turn, the *HostnameFilter* and *LabelFilter* filtering rules, respectively (see the previous subsection).

Till now, the new features introduced in Qinling are related to the creation of the runtimes. However, for functions execution, this is not enough. Indeed, if we consider standard Qinling implementations in Cloud environments, to identify the runtime where a function should be executed, Qinling relies on the IP addressing associated by the OpenStack networking service (i.e., Neutron/Kuryr) to the container. In our case, the edge-based capsules cannot use the standard OpenStack networking service

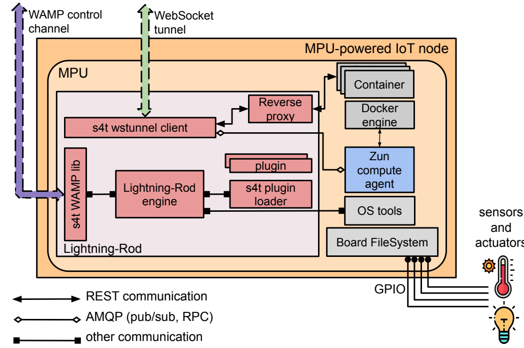


Figure 52: The device-side FaaS system architecture.

since it is unable to cope with the particularity of IoT deployments (Neutron does not manage distributed instances deployed outside the Cloud). Hence, we are not able to select the runtime based on its private/overlay IP address. To manage the association between the incoming execution requests and the runtimes they are associated with, we make an association between the couples {Cloud IP address, port}, and runtimes. Therefore, each of the runtimes deployed at the network edge is reachable through the Cloud public IP address and a specific port (i.e., the service forwarding capability provided by S4T). On the IoT node side, we come up with the reverse proxy to route the requests received (see Figure 52). In particular, we modified the request going from Qinling to the edge-based devices by adding a new field, *runtime_id*, that points to the correct runtime on the IoT device. Once the reverse proxy receives a request, it checks the *runtime_id* field contained in the request and associates it with one of the capsules it manages (each pause container has a label called *runtime_id*). Consequently, we modified the Qinling database to store the capsules' *runtime_ids* as well.

5.4.3 Deviceless functional workflows

5.4.3.1 Runtime creation

In this subsection, the workflow of creating a runtime on a specific edge-based node using the Qinling subsystem is described. In particular, we put the spotlight on the different interactions among the subsystems involved (i.e., Qinling, Zun, and IoTronic). As a use case essential requirement, we presume that the hosting node

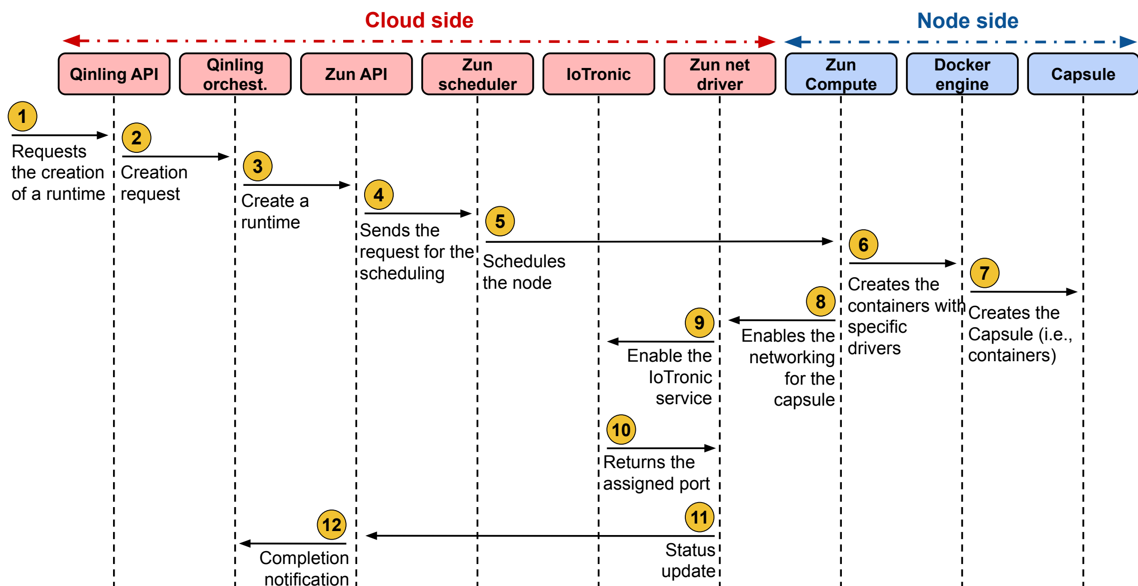


Figure 53: A runtime instantiation workflow.

(i.e., the IoT device) is already registered to the Cloud. The following steps describe the workflow of instantiating a runtime on a device deployed at the network edge:

1. The user requests to deploy a runtime on a specific remote node, either through the dashboard or the CLI. Then, the dashboard/CLI performs a specific Qinling API call via REST.

2. The Qinling API server forwards the runtime creation request to the Qinling (Zun-based) orchestrator.

3. The Qinling orchestrator sends the request to the Zun-api server with a particular body where it specifies the *nodeName* attribute (i.e., the hostname of the device where the runtime should be created (see Section 5.4)).

4. The Zun-api server forwards the request to the Zun scheduler in order to select the host (i.e., IoT device) where the runtime should be created.

5. The Zun scheduler apply the new filter *HostnameFilter* (see Section 5.4.2) to schedule the appropriate host. Afterwards, it sends to the Zun-compute agent running on that host a request to create a *capsule* (the three required containers).

6. After receiving the request, the Zun-compute agent sends an HTTP request, on localhost, to the Docker engine to create the *capsule*.

7. Docker creates the containers needed.

8. After creating the containers, the Zun-compute agent requests the Zun networking driver, that uses IoTronic, to make the container reachable by exposing the *capsule* (i.e., the pause container) to the user.

9. The Zun networking driver interacts with the IoTronic subsystem to expose the *capsule*. Specifically, IoTronic exposes the capsule using a particular port associated with a public IP address on the Cloud then, a WS tunnel is created between the IoT device and the Cloud. Hence, any request received on the Cloud port/IP address will be forwarded through the WS tunnel to reach the IoT node where the concerned runtime is instantiated (reaching exactly the runtime container during a function execution will be discussed in the next workflow that exploits the reverse proxy).

10. IoTronic sends back to the Zun networking driver the port and the Cloud IP address associated with the *capsule* then, the metadata of the capsule already created in step 7 (i.e., capsule *runtime_ id*, IP/port) will be stored in the Zun database.

11. The Zun-compute agent sends a notification about the status of the operation to the Zun-scheduler. The response contains the *runtime_ id* of the capsule created that will be stored in the Qinling database (the *runtime_ id* is important to reach the runtime when a function should be executed).

12. The Zun-scheduler forwards the notification to the Zun-api server that contacts, in its turn, the Qinling subsystem to store on its database the *capsule runtime_ id*.

The workflow reported previously concern the instantiation of a runtime on a single IoT node. In order to deploy runtimes on several IoT nodes having the same label,

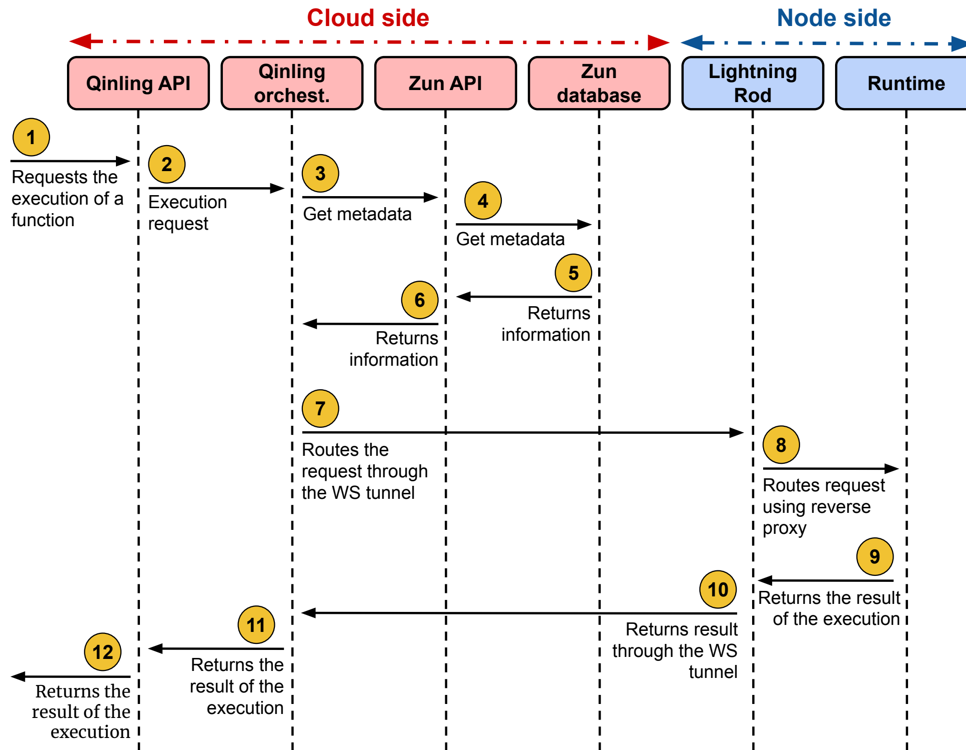


Figure 54: A function execution workflow.

the same workflow will take place while specifying in step 3 the *nodeSelector* attribute instead of *nodeName*. Therefore, the Zun scheduler will use the *LabelFilter* to select the set of IoT nodes where the runtimes should be provisioned.

5.4.3.2 Execution workflow

The workflow of executing a function on a specific runtime deployed on an edge-node using our FaaS system subsystem is reported bellow. As a use case essential requirement, we presume that a hosting node (i.e., an IoT device) is already registered to the Cloud and the user has already written his/her function and associated it to a particular runtime (a *function_id* is stored on the Qinling database) after deploying a runtime on the edge device. The following steps describe the workflow of executing the function on the runtime:

1. The user sends a request to execute a function on a specific runtime created on a remote node, either using the dashboard or the CLI. Then, the dashboard/CLI

performs a specific Qinling API call via REST. The request body contains the *function_id*.

2. After getting from its database the *runtime_id* involved, the Qinling-api server forwards the request to the Qinling (Zun-based) orchestrator.

3. The orchestrator contacts Zun using the API server to get the metadata of the runtime involved (i.e., IP/port used by IoTronic to expose the runtime).

4. The Zun-api server forwards the request to the Zun database to get the information.

5. The Zun database sends back the metadata to the Zun-api server.

6. The Zun-api server forwards back the response to the Qinling orchestrator.

7. Based on the metadata received, the orchestrator routes the execution request through the WS tunnel already created when the runtime was deployed on the edge node. The request forwarded contains a field with the *runtime_id*.

8. Once the request reaches the IoT node through the WS tunnel, LR uses the reverse proxy to identify the runtime concerned. Specifically, the reverse proxy makes use of the *runtime_id* field in the received request to make the association.

9. After the execution of the function on the runtime, a result is sent back to LR.

10. Using the WS tunnel, LR forwards back the result to the Qinling-orchestrator.

11. The orchestrator sends the result to the Qinling-api server.

12. Finally, the function execution result is sent to the user/dashboard.

5.5 A FaaS-powered flow-based development tool for distributed IoT environments

In this section, we report a use case where the edge-FaaS system described before has been used. In particular, we enhanced the capabilities of the flow-based

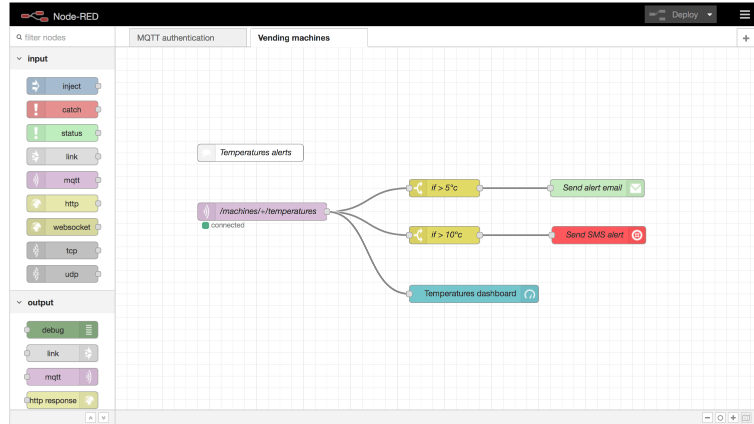


Figure 55: Node-RED browser-based flow editor.

development tool for visual programming namely Node-RED to cover distributed IoT deployments.

5.5.1 Node-RED

Node-RED[30] is a is a flow-based development tool for visual programming for wiring, hardware devices, APIs and online services as part of the IoT topic. It provides a browser-based flow editor (see Figure 55) to create JavaScript actions. The runtime of the tool is built on Node.js. The data pipelines programs created using Node-RED are called flows consisting of nodes connected by wires. The user interface is easy to use as it consists of a flow editor with node templates on the left (see Figure 55) that can be dragged and dropped into a flow canvas. As soon as the user create his/her flow or modifying it, he/she can deploy it by saving it into the server and (re)starting its execution on the Node-RED server [151].

Node-RED is supported by IBM and a large community of users that contribute new nodes and flows. New nodes can be implemented in JavaScript and added to the system by adding an HTML file to implement the UI in the browser, and a JavaScript file for data processing or integration on the server. Text representations of flows can be imported and exported between instances.

While Node-RED have been found to be useful on its own as data flow tool, several

[30] https://nodered.org

IoT scenarios require the coordination of computing resources across a distributed environment: on servers, gateways and devices themselves. A feature that Node-RED does not provide [151]. An interesting feature then is to extend the simple flows model to include different types of wires. In a distributed flow, the wires between nodes are not all local connections in the same execution engine, but may involve the transfer of data between servers and devices over a local or wide area network. "Local" wires are hosted on the same execution engine, while "remote" wires will require a network connection.

5.5.2 Extended Node-RED

We exploited the Deviceless paradigm to extend the capabilities of the Node-Red flow-based development tool for visual programming. In particular, we added a new type of nodes that exploit, underneath, the functions managed by Qinling. Thanks to the Deviceless system, a user can design workflows/pipelines among IoT devices deployed at the network edge. Furthermore, the solution can also use the standard Cloud-based Serverless computing model as shown in Figure 56 that depicts the high-level architecture of the approach. The Deviceless approach enables a seamless orchestration of Qinling action containers deployed at the network edge with Docker and Node-RED. From the user perspective, it is just an extended version of Node-RED allowing the feature of enabling Qinling actions using a distributed IoT infrastructure. Besides, instead of using only JavaScript to create actions/functions, our approach extends the Node-RED programming languages choice to include other languages such as Python.

When a user envisions the provisioning of a distributed flow, he/she can do it without interacting or managing any remote IoT node. We report here a simplified functional workflow (see Figure 56):

1. The user develops/defines the business logic to be deployed on a set on nodes (i.e., functions to read the value of a sensor or actuate an action).

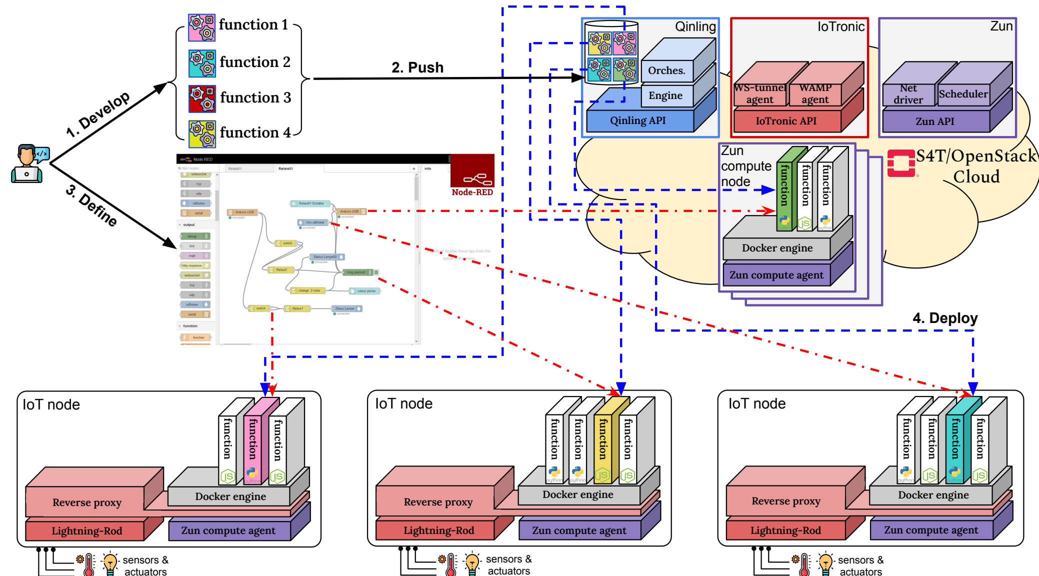


Figure 56: Integration of the flow-based Node-RED development tool with the Deviceless approach.

2. The user pushes the functions to the Cloud Qinling repository. Then, he/she configures Node-RED nodes to use Qinling function (see Figure 57).

3. The user, using the extended Node-Red GUI that also has the new nodes based on Qinling. The user can design his/her customized workflow among a set of distributed IoT nodes and the Cloud-based instances as well. We mention here that the IoT nodes do not host any Node-RED server. The user do not manage the IoT nodes that are not even aware about Node-RED.

4. Once the user runs his/her Serverless/Deviceless flow using Node-RED GUI, Qinling, together with Zun and IoTronic, takes the responsibility of instantiating the functions and executing them as required by the application.

Edit qinling-node node
Delete Cancel Done
Properties
Endpoint 10.0.0.11:7070
Keystone 10.0.0.11:5000
Username qinling
Password ············
Function 10193433-f494-452d-9a84-a904c8d20b6f
Input {"param1":"value1"}

Figure 57: Node-RED node for Qinling: configuration editor

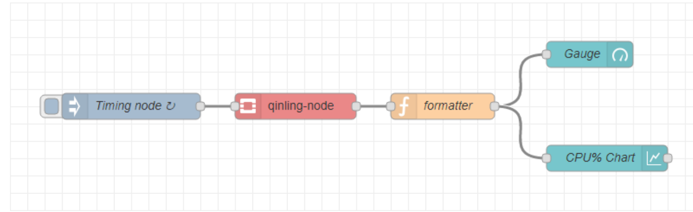


Figure 58: Qinling node-based example flow.

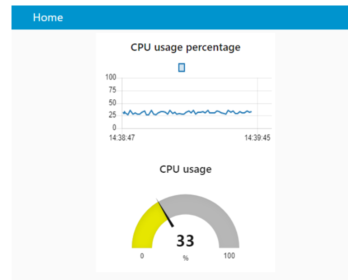


Figure 59: Graph and gauge generated by the Qinling functions.

To highlight a use case of our approach, we created a simple flow that makes use of Qinling to monitor the CPU usage of an edge IoT node. We mention that the flow is created on a Cloud-based VM and not the IoT node: the device is not even aware about Node-RED.

In Figure 58, we show a flow that uses a Qinling-based Node-RED function. In particular, the flow is meant to report the CPU usage on a Raspberry Pi every one second. A Node-RED *Timing node* is used to trigger the function deployed on the board (the node is named Qinling-node in the figure) to report the CPU percentage usage. Afterwards, the value received is sent to a *formatter* in order to create a 10 minutes history CPU usage graph and a gauge showing the last value received (see Figure 59). We mention here that the Node-RED flow is conceived on a Cloud-based instance and not the IoT node itself (unlike OpenWhisk Light, see Section 5.2.3)

5.6 Experimental results

In the following, a preliminary evaluation of the Deviceless system is provided. In particular, we focus on the resource usage of the system on the IoT nodes and the impact of the new Zun networking driver we introduced. For that purpose, we

	Idle		10 requests	
	CPU	**RAM**	**CPU**	**RAM**
Reverse proxy	$\simeq 0\%$	1.1%	3.4%	1.1%
WS tunnel client	$\simeq 0\%$	4.1%	4.2%	4.1%
Zun agent	$\simeq 0\%$	9.7%	0%	9.7%
Lightning-Rod	$\simeq 0\%$	0.8%	0%	0.8%
Total	$\simeq$ **0%**	**15.7%**	**7.6%**	**15.7%**

Table 14: The resources usage of the Deviceless system on an IoT node

deployed the Cloud-side of the system (i.e., IoTronic, Qinling, Zun) on a 10th gen i5 Intel-based late 2020 Macbook Pro. We mimic the IoT node using a VM hosted on the same machine to have a fixed latency emulating a WAN interconnection. Therefore, we can evaluate the impact of the networking driver we conceived. In particular, we used the Linux Trafic Control (TC) with queuing disciplines (qdisc) to set the latency between the Cloud and the IoT node. The IoT node (i.e., VM) is configured with one vCPU and 1GB of RAM.

In Table 14 we report the averaged CPU and RAM usage of the Deviceless system on the IoT node. In particular, we report the resource usage of all the processes involved on the node-side (i.e., WebSocket tunnel client, Zun compute agent, LR, and the reverse proxy). We are considering two scenarios, an idle mode, when all the processes mentioned earlier are running but not executing any task (during 10 minutes). The second scenario, instead, is when the IoT node receives ten simultaneous functions execution requests (the table reports the results averaged over 20 runs).

As we can notice from the table, when the IoT node is not receiving any execution request, the Deviceless system does not use any CPU resources. Regarding the RAM usage, the system uses precisely 15.7% of the 1 GB of memory available. We mention that the RAM values for all the processes remained constant and equal to the values shown in the table during the 10 minutes. When the IoT node receives ten simultaneous execution requests, the CPU values increase for both the LR reverse proxy (that routes the requests to the runtime) and the WS tunnel client (a WS tunnel is established between the Cloud and the IoT node). Yet, both the Zun agent and LR do not load the CPU as they are not involved during functions execution (they are

	Serverless (Cloud-based)	**Deviceless (50 ms latency)**	**Deviceless (100 ms latency)**
Execution time	150 ms	419 ms	850 ms

Table 15: Execution time comparison between Serverless and Deviceless

part of the control plane: creating runtimes and instantiating WS tunnels).

To further evaluate the solution, we conducted another experiment to assess the impact of the network latency on the functions execution time. We specifically compared the execution time between a standard Cloud-based Serverless deployment and the Deviceless approach that uses the new WebSocket-based Zun networking driver.

Table 15 reports the results obtained when comparing Serverless and Deviceless execution times (results averaged over 10 runs are reported). In the table, the execution time refers to the whole process starting from sending the request to execute a function until getting the result. This process includes the management of the request on the Cloud as well as the delays to reach the containers (either on the Cloud for Serverless or the network edge for Deviceless), execute the function and sending back the result.

Of course, when talking about Deviceless, we are expecting higher delays due to the network latency. This is what we can see from Table 15. Executing a function on the Cloud requires only 150 ms. On the other hand, when the containers are on the network edge, more delays are introduced. We would like to mention that the delay added is not equal to the RTT: sending the request and receiving the result. In fact, if it were the case, we would expect a value around 250 ms when the latency introduced by the network is equal to 50 ms (i.e., Serverless execution time + RTT = 150 ms + 100 ms). However, since the new driver we are introducing uses a TCP-based Websocket tunnel, which means a three-way TCP handshake (i.e., 1.5 RTT) plus the time required to send the execution request and receiving the result through the tunnel (i.e., 1 RTT), the delay increases accordingly (see Table 15).

To deal with the higher latency introduced by the three-way TCP handshake, we

would like to improve the performances in terms of the delay required to execute the functions. In particular, a solution that can be considered to decrease functions execution time is instantiating WebSocket tunnels more proactively. Thus, anticipating demand for function execution requests with predictive routines and keeping TCP sessions alive to avoid three-way handshakes for most functions invocations. Besides, to overcome the Cloud issues related to latency and enhance the Deviceless approach performances, another aspect to investigate is exploiting the proximity introduced by Mobile-edge Computing (MEC) [152] to IoT nodes to orchestrate the Serverless functions executions at the network edge.

Chapter 6

Industrial use case: Stack4Things as a Fog system for Industrial IoT monitoring applications

6.1 Introduction

To improve the performances of industrial processes and achieve significant optimizations, the fourth industrial revolution (also known as Industry 4.0) aims at substituting the mechanical operations and human interventions with automated systems relying on machine-aided decisions [153]. In this context, ICTs promote the use of embedded systems within the industrial and manufacturing fields. Indeed, IoT is the main facet able to introduce automation in industrial processes by making devices/things able to communicate among each other and/or with digital systems [154]. To emphasize the particularity of IoT usage in the industrial sector, the term Industrial Internet of Things (IIoT) [155] has been promoted as a substitute name instead of the all-encompassing IoT term.

In IIoT deployments, Industrial Wireless Sensors Network (IWSN) technologies play a key role in deploying monitoring and control systems thanks to the higher flexibility in sensors and actuators placement [156]. However, to make IWSN deployments capital expenditures affordable, sensor nodes are mainly composed of inherently resource-constrained devices unable to cope with data processing or storage. Therefore, IWSNs mostly rely on transferring data toward centralized systems with enough computing and storage resources [157]. One of the mainstream solutions to cope with this problem is the Cloud computing paradigm. Nevertheless, even though providing enough resources to deal with data management/processing, the Cloud approach induces, at the same time, several drawbacks (e.g., high latency, security/privacy issues, data storage cost) [158]. As such, the Fog computing paradigm has been introduced to

solve (most) of the issues by pushing resources to the network edge and thus, enabling smart operating edge environments with enhanced capabilities [159]. In this design, a number of WSN motes can be clustered around a (powerful) Fog node capable of dealing with data processing/storage and derive actionable intelligence locally. The Fog nodes can also have the possibility to vertically offload workloads to the Cloud when more resources are required; thus constituting a stacked Fog/Cloud architecture [160].

In this chapter, we propose a Fog-based architecture suited for IIoT monitoring scenarios. We used our S4T middleware in particular, the remote (re)programmability of the Fog nodes to provide a flexible system for industrial monitoring duties. Moreover, we exploited the Fog computing to implement a data aggregation technique able to improve network performance in terms of packet delivery and latency. As a case study, we focused on a distributed monitoring system for induction motors. In particular, with the aim of testing the proposed architecture in a real-world industrial scenario, we designed a specific sensor-board for real-time monitoring of motors health parameters.

6.2 System Architecture And Description

In this section, we present a detailed description of our layered Fog-based monitoring system (see Figure 60) as well as its hardware and software requirements.

6.2.1 The sensing layer

To implement the *Sensing layer* for our industrial monitoring system, we had to make it easily scalable and cost-effective. For this purpose, we opted for the use of battery-powered and wireless-capable devices with extremely low power data transmission. In particular, we use devices based on the widely adopted IEEE 802.15.4 protocol [161]. Among the available motes based on the IEEE 802.15.4 standard and the 2.4 GHz ISM band, we picked IRIS mote products by MEMSIC Inc. As the operating system for the motes, we chose TinyOS[31] that makes the source code easily

[31] http://www.tinyos.net/

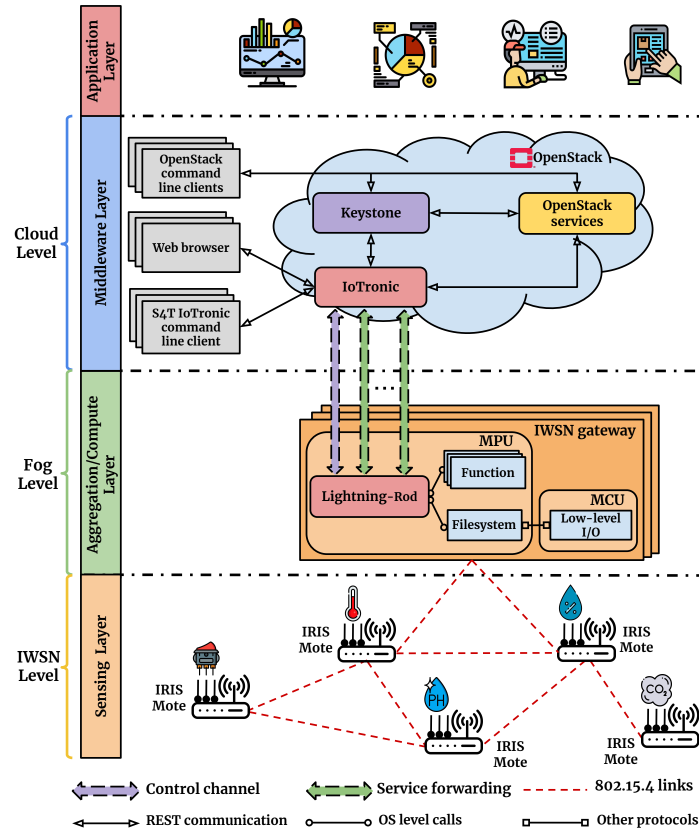


Figure 60: Monitoring system layers.

portable when other commercial platforms developed, for instance, by Advanticsys and Zolertia.

As regards the rest of the protocol stack, we used IPv6 and IPv6 over Low-Power Wireless Personal Area Networks (6LowPAN) [162] as network layer protocols and the Routing Protocol for Low-Power and Lossy Networks (RPL) as routing protocol. Moreover, UDP has been selected for the transport layer.

For the particular use case we are addressing, i.e. the monitoring of three-phase motors, a sensor board has been specifically designed. We report its block diagram in Figure 61. Technically, this sensor board can interface a IWSN mote with a three-phase motor with voltages in the range of 10-500V and currents up to 50A. The board uses hall effect transducers to measure voltages and currents while providing a perfect (galvanic) isolation between the motor and the mote. Regarding the motors characteristics to monitor, temperature and accelerometer sensors are integrated in

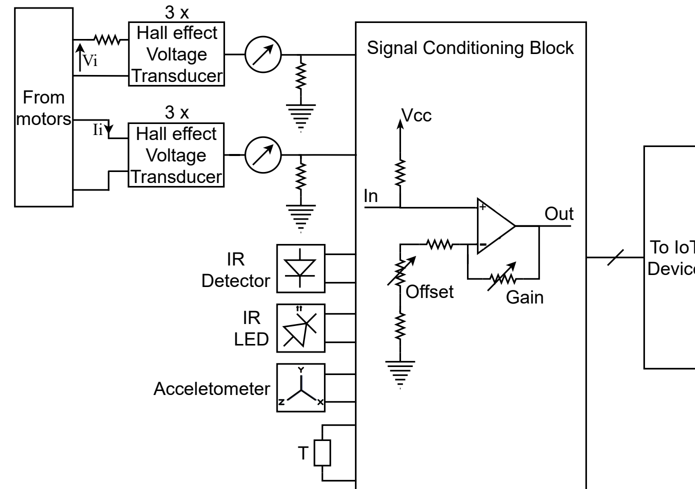


Figure 61: Block diagram of the sensor board used in our experiments for interfacing 3-phase motors. The board is equipped with all the necessary sensors for measuring voltages, currents, speed, temperature and mechanical vibrations.

the board to monitor the temperature and mechanical vibrations. Finally, a system composed of an Infrared (IR) light source and an optical detector is put in place to measure rotational motion.

6.2.2 The aggregation/compute Layer

To process/manage the data generated by the motes composing the *Sensing layer*, we introduced in our system a Fog layer composed of relatively powerful nodes. We used specifically as Fog nodes the low-cost MPU-powered Arancino[32] single board computers commercialized by the academic spinoff company Smartme.IO. The prominent feature of these boards is the joint on-board availability of an MPU alongside one microcontroller unit (MCU) in order to assign workloads and peripherals (e.g., sensors) according to the unique features of every module, specialized for specific duties, while making room for the two subsystems to work closely enough to cooperate. In particular, on the MCU side, the Arancino host, in its base configuration, an Atmel SAMD 32-bit part, which acts as a bridge between sensors and the MPU where most logic will run. In terms of MPU, a socket can host different MPUs. In particular, for our scenario, we used a Raspberry Pi 3 compute module.

[32]https://arancino.cc/

Being an MPU powered board, the Arancino can deal with most of processing tasks in IoT scenarios and, in particular, data aggregation algorithms. Indeed, the board can host different Linux distributions (e.g., Raspbian, OpenWRT) and therefore, it can host several runtime environments (e.g., Python, Node.js, Java). Consequently, a wide range of available libraries can be used. In particular, we used the Python SciPy library to deploy a data aggregation algorithm (see Section 6.4). To efficiently manage the deployment of the aggregation algorithms, we used the feature of remote programmability of boards provided by the *Middleware layer* (see next subsection).

Besides performing data aggregation, this layer is also responsible for setting up topology and managing topology information. In particular, we implemented a simple clustering scheme where Fog nodes send packets periodically and each sensor node selects its cluster head by measuring the Received Signal Strength Indicator (RSSI) of received packets. In particular, for each sensor node, the cluster head is chosen considering the Fog gateway corresponding to the maximum RSSI level. It is worth noting that the proposed scheme can be profitably used also in the case of dynamic networks.

6.2.3 The middleware layer

In a large industrial field, it is likely to have Fog nodes distributed within different sites/working zones and, therefore networks with different configurations and security policies. In this context, to enhance the capabilities of our industrial monitoring system and manage efficiently the Fog nodes, we used our OpenStack-based S4T platform meant for IoT infrastructure management as a *Middleware layer*.

S4T provides the capability to inject custom code on any edge-based node, even at runtime, under the guise of independent pluggable modules, i.e., a way to adapt the behavior of the IoT node under consideration to the task at hand (i.e., the context). The runtime environment of choice in our case is Python as it provides advanced programming facilities with a huge collection of third-party libraries and a comprehensive open source ecosystem overall. The injection process is carried out by transferring the

code as payload of a WAMP[33] RPC message. An instantiated plugin can be defined in this context as any process running in parallel that executes a workflow, e.g., a continuous collection of sensor measurements. This (re)programmability capability makes the Fog layer flexible in the sense that it can be adapted to networks changes for example, when adding or removing sensors to/from a network. Besides, the flexibility of Fog nodes extends the scope of possible application scenarios (e.g., schedule motors maintenance, modeling energy consumptions, automatic detection electrical and mechanical faults).

6.2.4 The application layer

The top layer of the system is the *Application layer*. This layer exploits the data stored at the *Middleware layer* to provide for final users suitable GUIs to display the monitoring information using Grafana[34]. Applications also can provide easier interactions with the *Middleware layer* to manage the Fog nodes through RESTful interactions with IoTronic.

6.3 Security mechanisms of the system

Security is one of the most important aspects to consider in IWSN applications due to the fact that even basic system hacking could damage machines, business, and operators too [163]. In general, four basic security services are needed for IWSN applications: confidentiality, data integrity, authentication, and access control. The proposed architecture addresses all the above services also exploiting reconfigurability at the Fog layer to achieve the desired trade-off between security, complexity, and energy consumption. While standard mechanisms have been exploited to guarantee security at the Fog, Cloud, and Application layer, at the *Sensing layer*, a new ad-hoc approach has been designed and implemented. In this section, we provide an overview of the security mechanisms available in our architecture mainly focusing on the *Sensing layer*, where an original contribution is introduced.

[33]https://wamp-proto.org
[34]https://grafana.com

6.3.1 Sensing layer security services

Confidentiality mechanisms are needed to guarantee that relevant business information are not read from unauthorized users. The common method for achieving that goal is encryption. Basically, encryption converts the original information into a form that is not intelligible by unauthorized users. Despite several encryption algorithms already exist, such as Data Encryption Standard (DES) and Advanced Encryption Standard (AES) [164], low-complexity encryption algorithms should be preferred for WSNs to take into account their constrained computational and energy resources.

In particular, for the *Sensing layer*, we designed a simple symmetric encryption algorithm specifically suited for WSNs that uses a 128-bit key to encrypt and decrypt packet payloads. The algorithm can be thought as a simplified version of the AES, developed with the main aim of reducing energy consumption and storage resources in comparison to the original AES algorithm. The algorithm has been implemented in TinyOS and tested with IRIS motes. Moreover related energy consumption has been measured.

Data integrity services avoid that unauthorized users or hackers could modify information while packets are in transit. As regards communications between the gateway and the motes, we introduced a simple data integrity mechanism at the link-layer by concatenating a random number at the end of each packet before encryption.

A simplified block diagram of the encryption algorithm is shown in Figure 62. The 128-bit secret key is logically decomposed in 16 sub-keys of 8 bits each, i.e. $K_1, ..., K_{16}$, which are used to encipher blocks of data. More precisely, we assume that the payload of each packet is logically decomposed in blocks of four words of 16 bits each, i.e. $W_1, ..., W_4$. Each block is then processed in six consecutive steps to obtain an enciphered block, i.e. $C_1, ..., C_4$. In each step a 16-bit word is transformed by a function $f()$ to obtain another 16-bit word. Henceforward, we use the notation $[MSB^{(k+1)}, LSB^{(k+1)}] = f([MSB^{(k)}, LSB^{(k)}])$ to indicate that in the step k the Most Significant Byte (MSB) and east Significant Byte (LSB) are transformed by the func-

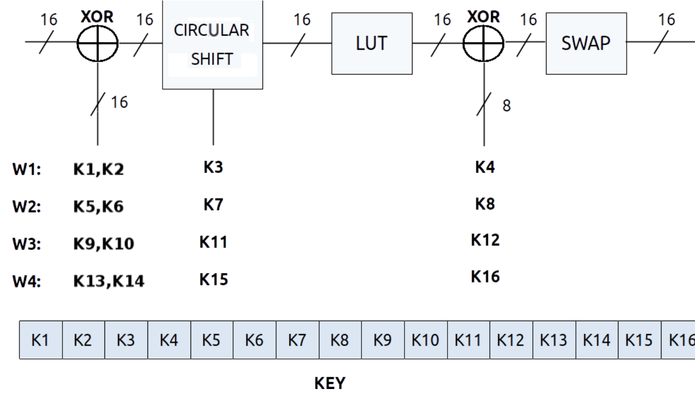


Figure 62: Block diagram of the encryption algorithm used for the *Sensing layer*

tion $f()$. Moreover, we indicate with $rot_n()$ a cyclic shift of n bits. The proposed algorithm is shown in Algorithm 1 and, according to the previous notation, it can be summarized as follows.

(a) **SPLIT:** each input word is logically splitted in two parts corresponding to the most significant and least significant byte, i.e. $[MSB^{(0)}, LSB^{(0)}] = [MSB(W_i), LSB(W_i)]$

(b) **XOR1:** the word $[MSB^{(0)}, LSB^{(0)}]$ is xored with sub-keys $[K_{4\cdot(i-1)+1}, K_{4\cdot(i-1)+2}]$;

(c) **CIRCULAR SHIFT:** the sub-key $K_{4\cdot(i-1)+3}$ is splitted in two parts, $[MSB(K_{4\cdot(i-1)+3}), LSB(K_{4\cdot(i-1)+3})]$ which are used to rotate the word $[MSB^{(1)}, LSB^{(1)}]$;

(d) **SCRAMBLING:** A LookUpTable (LUT) with 256 entries stores a random permutation of all integer numbers in the range from 0 to 255 and is used to obtain the word $[MSB^{(3)}, LSB^{(3)}] = [LUT(MSB^{(2)}), LUT(LSB^{(2)})]$;

(e) **XOR2:** another xor operation is performed so that bytes $MSB^{(3)}$ and $LSB^{(3)}$ are both xored with the sub-key $K_{4\cdot(i-1)+1}$;

(f) **SWAP:** finally, the cyphered word C_i is obtained by swapping least and most significant bytes obtained in the previous step, i.e. $C_i = [MSB^{(5)}, LSB^{(5)}] = [LSB^{(4)}, MSB^{(4)}]$.

The algorithm can be easily implemented even in simple microcontrollers, as those used in low-cost IoT platforms. The related TinyOS source code is only 16 rows

Algorithm 1: Encryption algorithm for the sensing layer

Inputs:
Four 16-bit unciphered words: $W_1, ..., W_4$
A 128-bit secret key $K = \{K_1, ..., K_{16}\}$
Output:
Four 16-bit ciphered words: $C_1, ..., C_4$

```
1  for i ← 1 to 4 do
       // (a) SPLIT
2      MSB^(0) = MSB(W_i)
3      LSB^(0) = LSB(W_i)
       // (b) XOR1
4      MSB^(1) = MSB^(0) ⊕ K_{4·(i−1)+1}
5      LSB^(1) = LSB^(0) ⊕ K_{4·(i−1)+2}
       // (c) CIRCULAR SHIFT
6      MSB^(2) = rot_{MSB(K_{4·(i−1)+3})}(MSB^(1))
7      LSB^(2) = rot_{LSB(K_{4·(i−1)+3})}(LSB^(1))
       // (d) SCRAMBLING
8      MSB^(3) = LUT(MSB^(2))
9      LSB^(3) = LUT(LSB^(2))
       // (e) XOR2
10     MSB^(4) = MSB^(3) ⊕ K_{4·(i−1)+4}
11     LSB^(4) = LSB^(3) ⊕ K_{4·(i−1)+4}
       // (f) SWAP
12     MSB^(5) = LSB^(4)
13     LSB^(5) = MSB^(4)
14     C_i = [MSB^(5), LSB^(5)]
15 return C_1, C_2, C_3, C_4
```

Figure 63: Voltage samples corresponding to a single packet transmission, with and without encryption.

and its implementation requires only 256 bytes of RAM (to store the LUT) and 604 bytes of ROM (for the program). We measured energy consumption of the proposed algorithm when implemented on IRIS platforms. With this aim, we connected a sensing resistors of R = 10 Ω in series to the battery pack of an IRIS node using the measurement scheme described in [165]. Furthermore, we programmed the mote to transmits packets of 100 byte every 200 ms and to wake-up with a period of 50 ms. Voltage signals across sensing resistors have been sampled with a Keysight MSOX3012 [166] digital oscilloscope so that energy consumption in a period T can be estimated as

$$E = \frac{V_{cc}}{R} \sum_{k=1}^{N} v_k T_s \tag{1}$$

where v_k is the k-th voltage sample, $N = 2000$ is the number of acquired samples and $T_s = \frac{T}{N} = 1$ ms is the sampling time. In this case, one measure of E is obtained every $T = NT_s = 2$ seconds which coincides with the time needed to send 10 packets. Finally, mean energy consumption per packet (E_p) has been estimated by averaging collected energy measurements over twenty experiments, i.e. considering an overall number of 200 packet transmission periods.

In Figure 63 we report voltage samples corresponding to a single packet transmission, measured with and without encryption. Additional energy consumption due to

the proposed enciphering algorithm can be appreciated by observing the time interval between 1.332 and 1.337 seconds delimited by vertical black lines.

As it is possible to notice, enciphering is done in about 5 ms and related mean energy consumption is $E_p = 112$ μJ. Note that this energy is negligible in comparison to the energy needed to send and receive a packet (i.e., 4.2 mJ, according to measurements collected in the interval 1.34-1.42 s) or either to sense the channel (i.e., 380 μJ, according to the measurements collected in the interval 1.47-1.48 s).

In [167] authors evaluated performance of several block ciphers for WSNs implemented on Mica2 and Arduino platforms. In particular, authors reported that encryption time of AES-128 for a 32-byte packet is 5 ms. Note that, in the same time, the proposed algorithm is able to encrypt a packet of 100 bytes. Mica2 and IRIS motes are based on microcontrollers within the same family and have the same clock speed and bus size. Therefore, we can state that, in comparison to AES-128, the proposed algorithm is able to reduce encryption time by a factor of three. Alternatively, when the same packet length is considered, the proposed algorithm reduces energy consumption by three times.

6.3.2 Fog layer security services

At the Fog layer, in order to achieve confidentiality and data integrity, it is necessary to focus on ensuring that the software running on the Fog nodes cannot be tampered with (even by users with physical access to the nodes). Moreover, it is necessary to ensure that impersonation attacks with respect to the Cloud layer cannot be conducted. Consequently, no user, no software, no device can act as an authorized gateway without the Cloud layer detecting the attack.

For such a reason, we equipped the Fog nodes (i.e., the Arancino boards) with a cryptochip, namely the ATECC608A produced by Microchip Technology Inc. This chip is partially able to act akin to a Trusted Platform Module (TPM) chip, e.g., as a secure enclave for keys and for on-chip (host-invisible) crypto-operations, such as signing, verification of a signature and random number generation.

First of all, we implemented a secure boot mechanism for the MPU of the Arancino. Once the system is booting, the cryptochip that holds the public key uses it to verify that the image digest of the operating system and application level software has been signed by the matching and genuine private key; therefore, no tamper of the software is possible without the Fog node refusing to boot. Moreover, a remote attestation mechanism has been implemented so that each Fog node can be uniquely identified by the Cloud counterpart that can reject any device trying to connect to the Cloud without being authorized. This latter mechanism is based on the use of WebSocket Secure (WSS) (which is based on HTTPS) as a communication channel between the edge device and the Cloud with a mutual authentication in place. An X.509 certificate stored in the cryptochip is used to authenticate the Fog node. The use of WSS also contributes to ensure data confidentiality and integrity being the communication channel encrypted with a negotiated session symmetric key.

6.3.3 Cloud layer security services

As previously described, the Cloud layer represents the management layer for the entire architecture. For what concerns security, authentication and access control are the major requirements [168]. The Cloud layer exposes an HTTPS-based user interface based on the OpenStack Horizon project. Basic user authentication via simple username+password credentials is managed using the OpenStack identity service namely, Keystone. Besides, the middleware deals with access control via a Role-based Access Control (RBAC) authorization engine implemented within the IoTronic service. The engine implements the concepts of tenants, fleets, roles, delegations allowing a fine-grained management of user permissions.

Finally, in order to defend the system against malicious Cloud administrators, a mechanism for strong authentication using Hardware Security Modules (HSMs) has been implemented. Each user is equipped with an HSM and each command sent to the Cloud is signed with the corresponding private key. The Cloud, then delivers the command, together with the signature, to the Fog node deployed at the edge. Verification is performed on the signature directly by the Fog node cryptochip so that

the command is executed only if the signature is verified.

6.4 Case study

In this section we consider a real-world IIoT case study where our Fog-enabled architecture has been customized for an Industry 4.0 factory and used to monitor twenty three-phase induction motors. For this purpose, twenty IRIS motes have been deployed across the factory and used for measuring amplitude and frequency of stator currents. All motes have been programmed so that currents are sampled at a fixed rate $f_s = 1$ kHz and acquired with the built in 10-bit Analog-to-Digital Converter (ADC). After acquiring 60 samples, i.e. 20 samples for each stator current, data are transmitted in packets of 75 bytes each.

Note that under the above settings, one packet is transmitted every 20 ms and thus, samples belonging to the same packet, represent exactly one Alternating Current (AC) cycle of the nominal frequency of the power supply, i.e. $f_n = 50$ Hz. Moreover, considering that the Nyquist frequency satisfies the condition $\frac{f_s}{2} > 9 \cdot f_n$, harmonic distortions up to the 9-th order can be detected. On the other hand, a data stream of $R_b = \frac{75B}{20ms} = 30$ kbps must be transmitted for each three-phase motor and, as will be shown in this section, this traffic cannot be handled without the Fog-layer. To prove the above statement, we compare three IWSN architectures:

1. an IWSN based on a simple star topology where all motes can communicate directly and wirelessly with the sink;
2. a more complex cluster-tree IWSN where the 20 motes are disposed in 4 clusters of 5 nodes each and, for each cluster, a gateway exists which acts as cluster head able to forward all received packets to the sink;
3. a third solution, based on the proposed Fog-enabled architecture shown in Figure 64, where we assume the same number of clusters and gateways of the previous scenario but with Fog nodes able to exploit a data aggregation technique, introduced with the aim of reducing the traffic towards the sink.

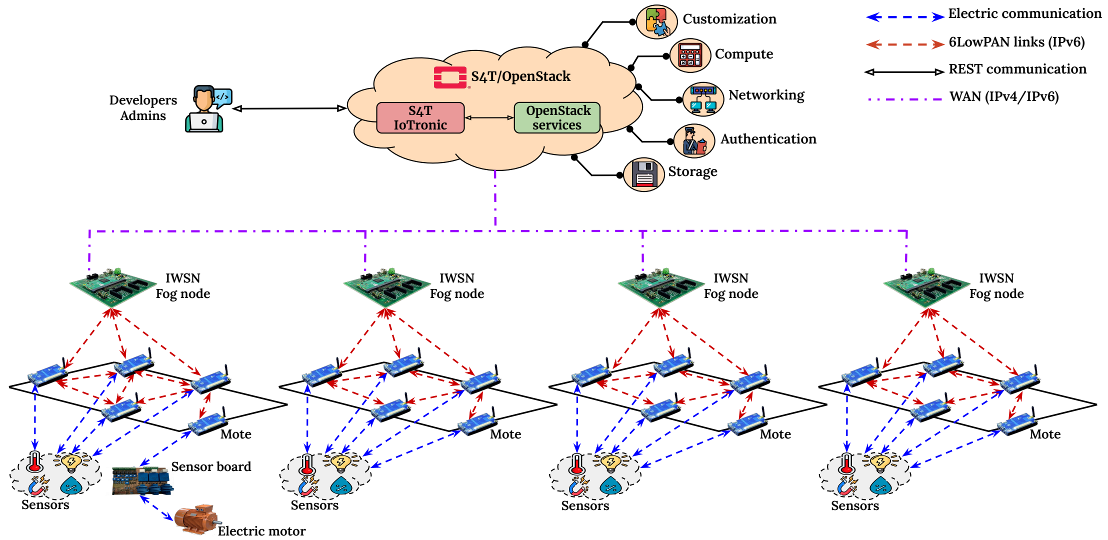


Figure 64: Proposed Architecture

In the next subsections we investigate the above scenarios considering both simulations and analytical models. In both cases, we considered the beaconless mode of the IEEE 802.15.4 MAC protocol where MAC parameters have been fixed according to Table 16.

6.4.1 Analytical model

In order to estimate the Packet Delivery Ratio (PDR) in the first scenario, i.e. a star network topology, we simplified and adapted the model proposed in [169]. In particular, we assume that sensor nodes transmit their data at a constant packet rate λ, here measured in packets per second (pk/s). Note that, in the proposed case study, where each node generates one packet every 20 ms, the packet rate per node is $\lambda = \frac{1}{20m} = 50$ pk/s.

According to the beaconless mode of the 802.15.4 protocol, all nodes, before to

Parameter	Value
macminBE	3
macmaxBE	5
macMaxCSMABackoffs	4
aMaxFrameRetries	3

Table 16: 802.15.4 MAC parameters used for simulation and analytical results.

transmit, wait for a random backoff period and then tests the channel to determine whether the channel is busy or idle by performing the Clear Channel Assessment (CCA) procedure. Basically, a node transmits only if the CCA succeeds, otherwise it waits for another random backoff period. This process is repeated up to a maximum number of attempts equal to macMaxCSMABackoffs+1, i.e. 5. When the maximum number of attempts is reached, the packet is simply discarded.

Therefore, if we indicate with α the probability of a single CCA failure, the probability that a node discards a packet after 5 consecutive CCA failures is given by $\beta = \alpha^5$. Even if the packet passes the CCA phase, and thus it is transmitted, it can fail to be received due to either collisions or channel impairments. According to the 802.15.4 protocol, we assume that a packet can be retransmitted at most aMaxFrameRetries=3 times if required, before being discarded.

By indicating with γ the overall packet failure probability due to both collisions and channel impairments, the probability of not receiving a packet due to either 5 CCA failures or 3 failed retransmissions is given by

$$\delta = \beta + (1-\beta)\gamma[\beta + (1-\beta)\gamma[\beta + (1-\beta)\gamma[\beta + (1-\beta)\gamma]]] \tag{2}$$

In [169], by modeling CCA attempts as independent Poisson processes, authors derived a set of fixed point equations which can be solved using an iterative scheme to obtain α, γ and thus δ. Finally, the packet delivery ratio can be evaluated as $PDR = 1 - \delta$. It is worth observing that γ can be expressed in terms of the probability of packet collision, p_c, and the packet error rate, p_e, as $\gamma = p_c + (1 - p_c) \cdot p_e$. In general, p_e is a time-varying quantity related to the Signal-to-Interference-plus-Noise Ratio (SINR). However, for sake of simplicity, we assumed that p_e is constant. In particular, to obtain numerical results reported in this section, we fixed $p_e = 10^{-2}$.

In Figure 65, we report the PDR obtained with the above model for different values of the packet generation rate λ in the case of a star topology IWSN with all nodes in the same carrier sensing range. In particular, we reported results for two networks with, respectively, 20 nodes (red line) and 5 nodes (black line). As it is possible

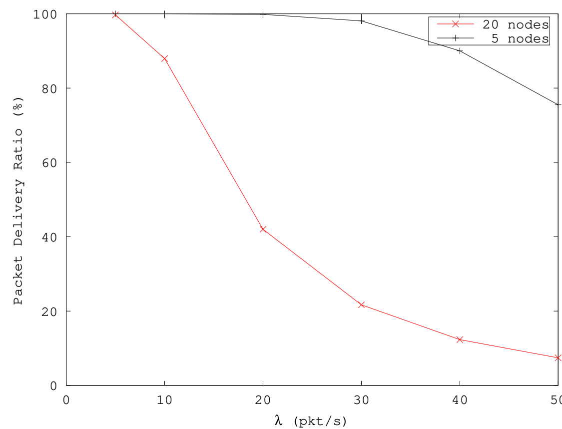


Figure 65: Packet Delivery Ratio (PDR) as a function of the traffic generation rate (λ) for 20-node (red line) and 5-node (black line) star-topology WSNs.

to observe, in the case of a star topology with 20 nodes, the PDR corresponding to $\lambda = 50$ pk/s is lower than 10% that is a too low value for any practical IWSN application. Therefore, connecting all 20 nodes in a simple star topology is not a viable solution for the proposed case study.

The low PDR value achieved in the previous scenario can be easily justified considering that with 20 motes, each transmitting at the rate of $R_b = 30$ kbps, the overall offered traffic is equal to $20 \times 30k = 600$ kbps and thus it is larger than the maximum bit-rate specified for the physical layer of IEEE 802.15.4 protocol equal to 250 kbps.

To solve this problem, a solution commonly adopted in IWSN consists in partitioning the network into clusters and use a different channel of 802.15.4 for different clusters. One more channel is then used for communications between cluster-heads and the sink. According to the IEEE 802.5.4 standard, a total of 16 channels are available in the 2.4-GHz band, numbered from 11 to 26. In particular, we assume that nodes are distributed in 4 clusters of 5 nodes each, i.e. the second scenario investigated for this case study. The four clusters exploit channels 15, 16, 20 and 21, respectively. Finally, channel 26 is reserved for communications between cluster-heads, i.e., the Fog gateways, and the sink.

As shown in Figure 65, where it is possible to compare the PDR for two star topologies with, respectively, 20 nodes (red line) and 5 nodes (black line), by reducing

the number of nodes that use the same channel, the PDR increases due to a reduced number of packet collisions. However, even with the previous solution, a traffic rate of $5\lambda = 250$ pk/s, i.e. 150 kbps, is generated by each cluster, which saturate the channel used for communications between the cluster heads and the sink.

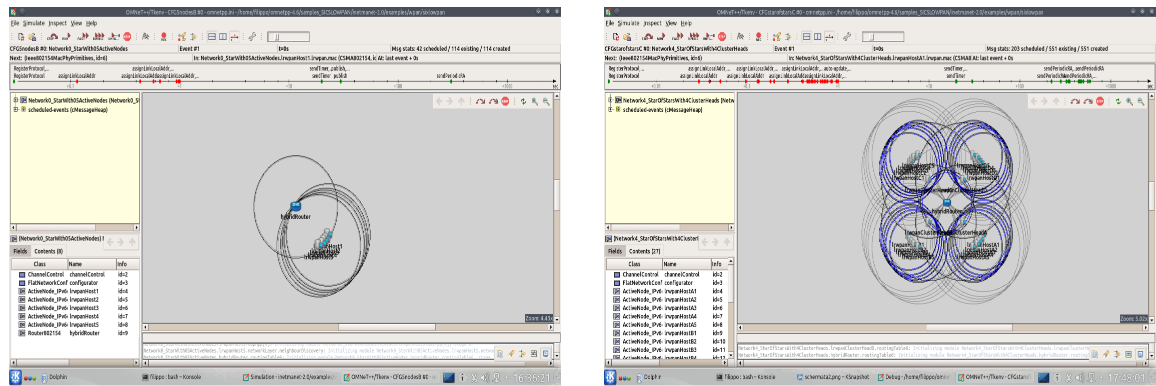
6.4.2 Exploiting the Fog-layer for data aggregation

With the aim of reducing the network traffic, we exploited the Fog layer to implement a data aggregation technique based on Singular Value Decomposition (SVD) [170]. Data aggregation exploits correlations among data to reduce energy consumption [171] and together with duty-cycling techniques is one of the most used energy reduction technique in IWSNs [172]. Basically, SVD allows to rewrite a $N \times M$ matrix A as the product of three matrices, U, Σ and V, such that $A = U\Sigma V^T$. Such a decomposition can be used for both compression and denoising.

We assume that each gateway rearranges collected data, i.e., 5 packets representing 60 current samples each, into a matrix A of 15×20 elements. Afterwards, the gateway performs SVD to obtain the set of matrices $\Gamma = \{U, \Sigma, V\}$ and thus, their submatrices $\Gamma_k = \{U_k, \Sigma_k, V_k\}$ where Σ_k is the $k \times k$ upper left corner submatrix of Σ and the matrices U_k and V_k are given by the first k columns of U and V, respectively. From SVD theory, it is known that the maxtrix $A_k = U_k \Sigma_k V_k^T$ approximate A by minimizing the norm $||A - A_k||$. Therefore, given Γ_k, A_k can be reconstructed.

It is worth noting that the overall number of nonzero elements of Γ_k are at most $k(N+1+M)$ and can be less than the number of elements of the original matrix A. In particular, every time that the condition $k(N+1+M) < NM$ holds, the set of matrices $\Gamma_k = \{U_k, \Sigma_k, V_k\}$ can be considered a compressed representation of A. Therefore, by transmitting Γ_k instead of the original matrix A, the amount of data that must be sent from each gateway to the sink is reduced by a factor $F = \frac{NM}{k(N+1+M)} \approx \frac{N}{2k}$, where the last approximation holds when $M \approx N$.

In particular, by fixing $k = 2$ it is possible to achieve a compression factor $F =$



(a) OMNET++: Single cluster with 5 nodes.

(b) OMNET++: Multi-hop network with 4 clusters.

Figure 66: Omnet++ simulation environment

$\frac{15\cdot 20}{2(15+1+20)} \approx 4.17$ so that the traffic that must be forwarded by each gateway is $\lambda_g = \frac{5\lambda}{F} \approx 60$ pk/s instead of 250 pk/s.

Obviously, we expect that, by reducing the amount of data that must be transmitted, packet collisions reduce too and thus the PDR increases. The simulation results reported in the next subsection confirm the previous statement.

6.4.3 Simulation results

We have simulated the above scenarios with OMNET++ [173]. OMNET++ is a modular, component-based C++ simulation library and framework for modeling communication networks, multiprocessors and other distributed or parallel systems. In particular, we used OMNET++ to compare the following three scenarios:

1. A single cluster with 5 nodes and 1 sink, disposed as shown in Figure 66a;
2. A cluster-tree network with 4 clusters of 5 nodes each, disposed as shown in Figure 66b and where all gateways act as simple relayers by forwarding all received packets to the sink;
3. The same cluster-tree network in the previous scenario but with the proposed SVD-based data aggregation technique enabled in the Fog layer.

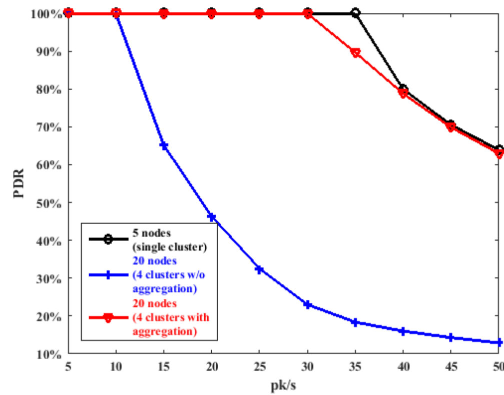


Figure 67: Packet Delivery Ratio (PDR) for different traffic generation rates λ in the case of: 1) a cluster with 5 nodes (black line); 2) a cluster-tree network with 4 clusters and without data aggregation (blue line); 3) a cluster tree-network with 4 clusters with data aggregation (red line).

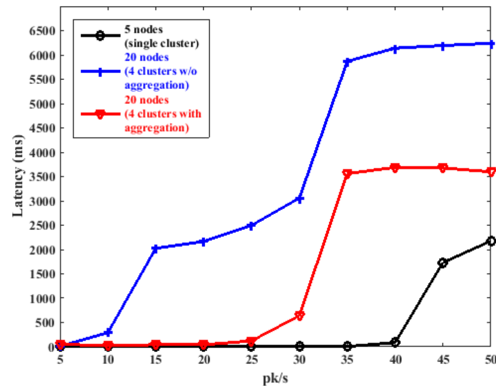


Figure 68: End-to-end latency for different traffic generation rates λ in the case of: 1) a cluster with 5 nodes (black line); 2) a cluster-tree network with 4 clusters and without data aggregation (blue line); 3) a cluster tree-network with 4 clusters with data aggregation (red line).

PDR and end-to-end latency achieved in the above scenarios for different traffic generation rates are reported in Figure 67 and Figure 68, respectively.

As it is possible to observe, according to simulation results in Figure 67, the PDR obtained with data aggregation (red line) is quite greater respect to the case without aggregation (blue line) and almost coincide with that achieved in the case of a single cluster of 5 nodes (black line). Moreover, as shown in Figure 68, data aggregation largely reduces network latency, i.e. end-to-end delay.

Nevertheless, simulations results show that, even with data aggregation, the net-

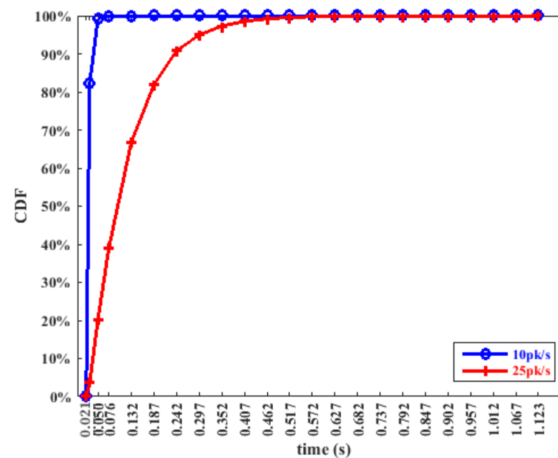


Figure 69: Cumulative Distribution Function (CDF) of packet delay for λ =10 pk/s and λ =25 pk/s.

work saturates with a traffic rate per node of 35 pk/s. Therefore, in order to achieve a better and reasonable PDR, at the end we decided to reduce the sampling frequency f_s from 1 KHz to 500 Hz, that is enough to detect possible harmonic distortions up to the 5th order. As a consequence, λ is reduced from 50 pk/s to 25 pk/s so that a PDR of 99.95% and an average latency of 120 ms are achieved by exploiting data aggregation.

For sake of completeness, in Figure 69 we reported the Cumulative Distribution Function (CDF) of packet delay for two values of λ, i.e. 25 pk/s and 10 pk/s. It is worth observing that, when the offered traffic per node is limited to 10 pk/s, almost all transmitted packets (around 99.3%) experimented an end-to-end latency lower than 50 ms. This result is in agreement with requirements imposed by the NIST for wireless monitoring applications of factory workcell [174].

It is worth mentioning that the time required to perform the proposed SVD-based data aggregation technique is very minimal and do not (significantly) impact the end-to-end latency. Indeed, we conducted an experiment to measure this time using randomly generated 15 x 20 matrices with float values in the interval [0, 500]. The result of performing 1000 SVD calculation shows that the average time required is lower than 2 ms. In practice, end-to-end delay is mostly due to the backoff mechanism at the MAC layer and to the time needed to collect enough packets before to exploit

λ (pk/s)	Technique	Latency (ms)	PDR (%)
10	DISCUS	34.7	99.99
	SVD	29.3	100.00
25	DISCUS	2808.1	67.86
	SVD	121.0	99.95

Table 17: End-to-end latency and PDR of DISCUS and proposed SVD-based aggregation scheme for two different traffic generation rates.

data aggregation.

Several other data aggregation techniques suitable for IWSNs exist, see for instance [175], [176] and references therein, and can be profitably implemented with the proposed Fog architecture.

For sake of completeness, we compared performance of the proposed data aggregation technique with DISCUS [171], a Distributed Source Coding (DSC) data aggregation scheme inspired by the Slepian-Wolf theorem. According to DISCUS, in each cluster, one node sends uncompressed data (as side information) while all other nodes in the same cluster transmits encoded (i.e., compressed) data. For simulation purposes, we considered that nodes are able to encode original 10-bit words with only 2 bits, i.e. we assumed for DISCUS a compression factor of 5, slightly greater than the factor achieved with our proposed SVD-based aggregation scheme.

End-to-end latency and PDR of both DISCUS and proposed SVD-based aggregation technique are reported in Table 17 for two different traffic generation rates, i.e., $\lambda = 10$ pk/s and $\lambda = 25$ pk/s.

As it is possible to observe, even considering similar compression ratios and the same network topology, the proposed scheme outperforms DISCUS in terms of both latency and PDR.

Higher latency of DISCUS can be justified considering that side information and compressed information are transmitted in two consecutive transmission attempts due to the Maximum Transmission Unit (MTU) imposed by the IEEE 802.15.4 protocol, i.e. 127 bytes at PHY layer. Therefore, at the least two CCAs should be performed

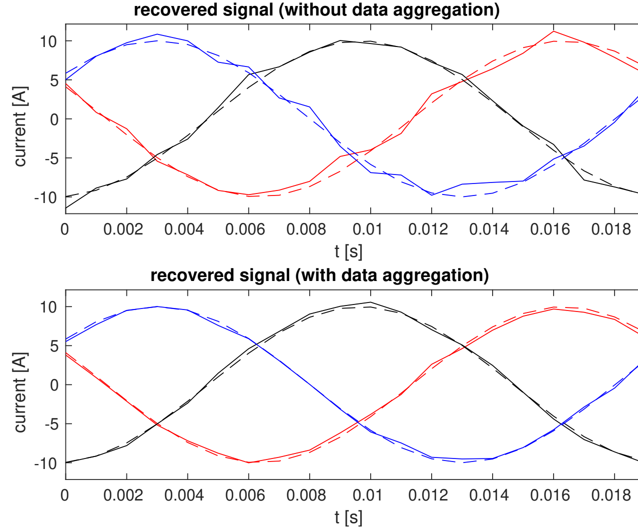


Figure 70: Recovered currents with and without data aggregation.

by the cluster heads in the case of DISCUS which obviously reflects on the latency.

With the aim of further improving network performance, we started the integration of channel coding techniques [177] and IoT-specific compression algorithms [178].

6.5 Further advantages of data aggregation

In the previous subsections we have shown that by exploiting data aggregation at the Fog layer, it is possible to substantially improve PDR and latency. However, higher PDR and lower latency are not the only advantages introduced by the Fog layer. In fact, SVD improves measurement accuracy as well. This can be observed in Figure 70 where we reported a set of three-phase currents reconstructed with and without the proposed SVD-based data aggregation technique. Note that, using SVD, a substantial reduction of the noise is achieved.

Obviously, by reducing the noise it is possible to achieve a greater accuracy in estimation of interesting parameters. For instance, in Figure 71 we reported the mean relative error achieved on measurements of maximum values of stator currents (I_{max}), obtained with and without data aggregation for different values of the Signal-to-Noise Ratio (SNR). More precisely, mean relative error values reported in Figure 71 have been obtained by averaging the relative error $err_{I_{max}} = 100 \cdot \frac{|\hat{I}_{max} - I_{max,true}|}{|I_{max,true}|}$ on 1000

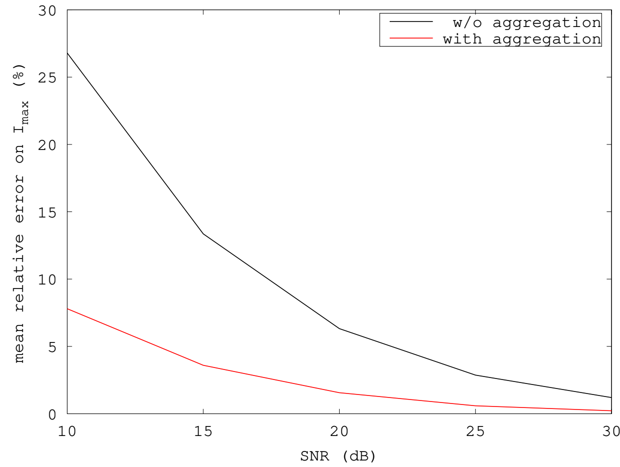


Figure 71: Mean relative error on the estimation of maximum current with and without SVD-based data aggregation for different values of SNR.

sinusoidal periods, where $I_{max,true}$ is the true (noise-free) peak value of the current and $\hat{I}_{max} = \max\{I_i\}$ is the corresponding measured/estimated value obtained from $N_s = 20$ samples.

As it is possible to observe, independently from the SNR, data aggregation provides a substantial reduction of measurement errors. This positive effect of SVD-based aggregation is observed also for other quantities of interest, i.e. root mean square values of currents [179] and also on frequency estimation [180].

Finally, it is worth mentioning that the Fog layer is mandatory for the proposed SVD-based data aggregation. In fact, SVD cannot be implemented with commercial motes commonly used in IWSNs. Moreover, exploiting reprogrammability of the Fog layer, Fog gateways can be remotely configured in order to adapt parameters of SVD, i.e., N, M and k, to other possible application scenarios, where different kind of signals or a different number of nodes must be handled.

Conclusion and future works

This dissertation proposes a set of new mechanisms to adapt the Cloud as-a-Service approach to IoT infrastructure while taking into consideration the emergence of new applications that require edge computing capabilities. To this purpose, an infrastructure-oriented Cloud approach is proposed (i.e., I/Ocloud) in this context to provide mechanisms for virtualizing IoT resources (i.e., sensors and actuators) and offer them as typical Cloud resources (e.g., compute, storage and networking). Besides, the approach also enables edge computing capabilities to meet the requirements of typical applications. The S4T framework has been enhanced with features to promote the adoption of the I/Ocloud paradigm within an OpenStack-based environment. Of course, the approach can also be extended to other Cloud management platforms.

In the first chapter of this thesis, we have described our view that aims at providing IoT infrastructure as shareable (Cloud) resources. We have investigated and showed the advantages of using the approach in enabling the device-centric model in IoT instead of adopting the limited Cloud-oriented data-centric approach. We have also outlined the capability of enabling edge computing using the I/Ocloud approach. In Chapter 2, we have introduced an OpenStack compatible solution for extending Cloud-based overlays to the network edge. The system enables users to connect their personal IoT devices to the Cloud-defined virtual networks; thus, they can communicate among each other and with Cloud-based instances as if they were on the same physical network. The solution can obviously bring a considerable level of flexibility to deal with actual real-world needs and future emerging deployments. It is worth mentioning that we had the opportunity to introduce our approach for networking at the edge to the OpenStack community at large, receiving broadly favorable feedback[35]. In Chapter 3, to enable the I/Ocloud view of adopting edge computing in specific scenarios, we have proposed a solution to deal with the instantiation and

[35] https://www.openstack.org/summit/vancouver-2018/summit-schedule/events/21201/an-edge-computing-case-study-for-monasca-smart-city-ai-powered-surveillance-and-monitoring

management of containers at the network edge. Thanks to the networking solution proposed in this chapter and the previous one, a user can regroup, within the same overlay, virtualized IoT nodes, physical IoT devices he/she may own as well as Cloud-based VMs. Besides, we also described a set of use cases where the containerization at the edge can also be used to provide a multi-tenant PaaS solution for enabling Fog computing. Furthermore, we described our Mininet-based S4T emulator for the Edge-to-Cloud continuum. In Chapter 4, an approach for exploiting IoT resources (either virtual or physical) using RESTful Web services is detailed. The solution is aligned with the WoT design. The approach can generally enable Web services in distributed environments using a Dynamic DNS feature provided by S4T. In Chapter 5, to have a complete integration between IoT and the Cloud, we have introduced our solution for enabling Serverless computing using edge-based IoT resources (a paradigm named Deviceless). In such a case, users can avoid provisioning I/Ocloud instances for long periods (when not needed) by making use of Serverless-like interactions to interact with remote sensors and actuators. Finally, in Chapter 6, an industrial user case where a set of S4T capabilities were exploited to enable Fog computing is reported. An exhaustive set of experimental results showing the efficiency of using Fog computing in industrial monitoring applications is provided.

Future works on S4T will be devoted to extending the platform integration with environments enabling Fog computing, such as the ETSI MEC [181]. In fact, if we consider the actual S4T architecture, most of the enabled functionalities rely on the Cloud. As future works, we would like to adapt S4T and make it usable in conjunction with Fog environments orchestrated by a Cloud-based S4T deployment. Such an aspect will significantly enhance the performance of the provided services (e.g., the DDNS routing mechanism that relies, in the actual design, on the Cloud). Another research direction will be focused on extending and integrating other OpenStack services (e.g., Keystone) and enhancing the security of S4T using advanced decentralized technologies, such as Blockchain. Besides, containers migration aspect in S4T should be deeply investigated to provide efficient mechanisms preserving the services' URLs after their migration to enable services continuity. Besides, another aspect we would

like to investigate is related to software rejuvenation [182] in IoT. Indeed, since the IoT nodes have to host long-running processes to enable communication with the Cloud and deal with data processing, mechanisms to cope with systems' failure and performance degradation are essential to avoid software aging and crash/hang.

Bibliography

[1] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad hoc networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[2] M. Patel, J. Shangkuan, and C. Thomas. "What's new with the internet of things?" (2017), [Online]. Available: https://www.mckinsey.com/industries/semiconductors/our-insights/whats-new-with-the-internet-of-things (visited on 01/19/2022).

[3] United Nations. "World population prospects: The 2017 revision." (2017), [Online]. Available: https://population.un.org/wpp/DataQuery/ (visited on 01/19/2022).

[4] Statista. "Iot: Number of connected devices worldwide 2015-2025." (2016), [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ (visited on 01/19/2022).

[5] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[6] Y. Liu, K. A. Hassan, M. Karlsson, Z. Pang, and S. Gong, "A data-centric internet of things framework based on azure cloud," *IEEE Access*, vol. 7, pp. 53 839–53 858, 2019.

[7] A. Botta, W. De Donato, V. Persico, and A. Pescape, "Integration of cloud computing and internet of things: A survey," *Future generation computer systems*, vol. 56, pp. 684–700, 2016.

[8] E. Cavalcante *et al.*, "On the interplay of internet of things and cloud computing: A systematic mapping study," *Computer Communications*, vol. 89, pp. 17–33, 2016.

[9] A. Detti *et al.*, "Viriot: A cloud of things that offers iot infrastructures as a service," *Sensors*, vol. 21, no. 19, p. 6546, 2021.

[10] K. Su, J. Li, and H. Fu, "Smart city and the applications," in *2011 international conference on electronics, communications and control (ICECC)*, IEEE, 2011, pp. 1028–1031.

[11] I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow, and P. Polakos, "Wireless sensor network virtualization: Early architecture and research perspectives," *IEEE Network*, vol. 29, no. 3, pp. 104–112, 2015. DOI: 10.1109/MNET.2015.7113233.

[12] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016.

[13] Z. Benomar, "Phd forum abstract: I/ocloud: Adopting the iaas paradigm in the internet of things," in *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, IEEE, 2021, pp. 412–413.

[14] Z. Benomar *et al.*, "Extending openstack for cloud-based networking at the edge," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, IEEE, 2018, pp. 162–169.

[15] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, "Cloud-based network virtualization in iot with openstack," *ACM Transactions on Internet Technology (TOIT)*, vol. 22, no. 1, pp. 1–26, 2021.

[16] ——, "Enabling container-based fog computing with openstack," in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, IEEE, 2019, pp. 1049–1056.

[17] ——, “Cloud-based enabling mechanisms for container deployment and migration at the network edge,” *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 3, pp. 1–28, 2020.

[18] Z. Benomar, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, “A mininet-based emulated testbed for the i/ocloud,” in *2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, IEEE, 2019, pp. 277–283.

[19] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, “A stack4things-based web of things architecture,” in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, IEEE, 2020, pp. 113–120.

[20] ——, “Enabling secure restful web services in iot using openstack,” in *2020 IEEE 17th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, IEEE, 2020, pp. 410–417.

[21] ——, “A cloud-based and dynamic dns approach to enable the web of things,” *IEEE Transactions on Network Science and Engineering*, 2021.

[22] ——, “Deviceless: A serverless approach for the internet of things,” in *2021 ITU Kaleidoscope: Connecting Physical and Virtual Worlds (ITU K)*, IEEE, 2021, pp. 1–8.

[23] Z. Benomar, G. Campobello, F. Longo, G. Merlino, and A. Puliafito, “Fog-enabled industrial wsns to monitor asynchronous electric motors,” in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, IEEE, 2020, pp. 434–439.

[24] B Zenomar, G. Campobello, F. Longo, G. Merlino, and A. Puliafito, “A new fog-enabled wireless sensor network architecture for industrial internet of things applications,” in *Proceedings of the 24th IMEKO TC4 International Symposium*

and 22nd International Workshop on ADC and DAC Modelling and Testing, Palermo, Italy, 2020, pp. 14–16.

[25] Z. Benomar, G. Campobello, A. Segreto, F. Battaglia, F. Longo, G. Merlino, and A. Puliafito, "A fog-based architecture for latency-sensitive monitoring applications in industrial internet of things," *IEEE Internet of Things Journal*, 2021.

[26] Y. Qin, Q. Z. Sheng, N. J. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When things matter: A survey on data-centric internet of things," *Journal of Network and Computer Applications*, vol. 64, pp. 137–153, 2016.

[27] P. Mell, T. Grance, *et al.*, "The nist definition of cloud computing," *National Institute of Standards and Technology (NIST) Special Publication 800-145*, 2011.

[28] P. Banerjee *et al.*, "Everything as a service: Powering the new information economy," *Computer*, vol. 44, no. 3, pp. 36–43, 2011.

[29] X. Sheng, J. Tang, X. Xiao, and G. Xue, "Sensing as a service: Challenges, solutions and future directions," *IEEE Sensors journal*, vol. 13, no. 10, pp. 3733–3741, 2013.

[30] R. Mizouni and M. El Barachi, "Mobile phone sensing as a service: Business model and use cases," in *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, IEEE, 2013, pp. 116–121.

[31] A. Bhawiyuga, D. P. Kartikasari, K. Amron, O. B. Pratama, and M. W. Habibi, "Architectural design of iot-cloud computing integration platform," *Telkomnika*, vol. 17, no. 3, pp. 1399–1408, 2019.

[32] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM sigcomm computer communication review*, vol. 39, no. 1, pp. 50–55, 2008.

[33] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *2010 24th IEEE international conference on advanced information networking and applications*, Ieee, 2010, pp. 27–33.

[34] A. Yousefpour *et al.*, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[35] D. Beimborn, T. Miletzki, and S. Wenzel, "Platform as a service (paas)," *Business & Information Systems Engineering*, vol. 3, no. 6, pp. 381–384, 2011.

[36] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE transactions on services Computing*, vol. 5, no. 2, pp. 164–177, 2011.

[37] S. Singh and I. Chana, "Cloud resource provisioning: Survey, status and future research directions," *Knowledge and Information Systems*, vol. 49, no. 3, pp. 1005–1069, 2016.

[38] S. Ibrahim, B. He, and H. Jin, "Towards pay-as-you-consume cloud computing," in *2011 IEEE International Conference on Services Computing*, IEEE, 2011, pp. 370–377.

[39] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet computing*, vol. 13, no. 5, pp. 14–22, 2009.

[40] J. Soldatos *et al.*, "Openiot: Open source internet-of-things in the cloud," in *Interoperability and open-source solutions for the internet of things*, Springer, 2015, pp. 13–25.

[41] A. Zaslavsky, C. Perera, and D. Georgakopoulos, "Sensing as a service and big data," *arXiv preprint arXiv:1301.0159*, 2013.

[42] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Sensing as a service model for smart cities supported by internet of things," *Transactions on emerging telecommunications technologies*, vol. 25, no. 1, pp. 81–93, 2014.

[43] X. Sheng, J. Tang, X. Xiao, and G. Xue, "Sensing as a service: Challenges, solutions and future directions," *IEEE Sensors journal*, vol. 13, no. 10, pp. 3733–3741, 2013.

[44] D. Bruneo, S. Distefano, F. Longo, G. Merlino, and A. Puliafito, "I/ocloud: Adding an iot dimension to cloud infrastructures," *Computer*, vol. 51, no. 1, pp. 57–65, 2018.

[45] M. U. Ilyas, M. Ahmad, and S. Saleem, "Internet-of-things-infrastructure-as-a-service: The democratization of access to public internet-of-things infrastructure," *International Journal of Communication Systems*, vol. 33, no. 16, e4562, 2020.

[46] S. Distefano, G. Merlino, and A. Puliafito, "Device-centric sensing: An alternative to data-centric approaches," *IEEE Systems Journal*, vol. 11, no. 1, pp. 231–241, 2015.

[47] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {fuse} or not to {fuse}: Performance of user-space file systems," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, 2017, pp. 59–72.

[48] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4things: A sensing-and-actuation-as-a-service framework for iot and cloud integration," *Annals of Telecommunications*, vol. 72, no. 1-2, pp. 53–70, 2017.

[49] I. Fette and A. Melnikov, "The websocket protocol," IETF, Tech. Rep. RFC 6455, 2011.

[50] G. Merlino, D. Bruneo, F. Longo, A. Puliafito, and S. Distefano, "Software defined cities: A novel paradigm for smart cities through iot clouds," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, IEEE, 2015, pp. 909–916.

[51] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[52] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[53] D. L. Passmore and J. Y. Freeman, "The virtual lan technology report," 1997.

[54] H. A. Seid and A. Lespagnol, *Virtual private network*, US Patent 5768271, 1998.

[55] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[56] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[57] S. Hanks, T. Li, D. Farinacci, and P. Traina, "Generic routing encapsulation over ipv4 networks," IETF, Tech. Rep. RFC 1702, 1994.

[58] M. Mahalingam *et al.*, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks," IETF, Tech. Rep. RFC 7348, 2014.

[59] J. Hoebeke, E. De Poorter, S. Bouckaert, I. Moerman, and P. Demeester, "Managed ecosystems of networked objects," *Wireless Personal Communications*, vol. 58, no. 1, pp. 125–143, 2011.

[60] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*, IEEE, 2017, pp. 1–7.

[61] O. Tomanek and L. Kencl, "Security and privacy of using alljoyn iot framework at home and beyond," in *2016 2nd international conference on intelligent green building and smart grid (IGBSG)*, IEEE, 2016, pp. 1–6.

[62] M. Villari, A. Celesti, M. Fazio, and A. Puliafito, "Alljoyn lambda: An architecture for the management of smart environments in iot," in *2014 International Conference on Smart Computing Workshops*, 2014, pp. 9–14.

[63] T. Høiland-Jørgensen, C. A. Grazia, P. Hurtig, and A. Brunstrom, “Flent: The flexible network tester,” in *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2017, pp. 120–125.

[64] F. Callegati, W. Cerroni, and C. Contoli, “Virtual networking performance in openstack platform for network function virtualization,” *Journal of Electrical and Computer Engineering*, vol. 2016, 2016.

[65] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012, pp. 13–16.

[66] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, “Fog computing for the internet of things: A survey,” *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, p. 18, 2019.

[67] D. Soldani, Y. J. Guo, B. Barani, P. Mogensen, I Chih-Lin, and S. K. Das, “5g for ultra-reliable low-latency communications,” *Ieee Network*, vol. 32, no. 2, pp. 6–7, 2018.

[68] P. Schulz *et al.*, “Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture,” *IEEE Communications Magazine*, vol. 55, no. 2, pp. 70–78, 2017.

[69] “Aws latency monitoring.” (2021), [Online]. Available: https://www.cloudping.co/grid (visited on 11/02/2021).

[70] J. Zhou, Z. Cao, X. Dong, and A. V. Vasilakos, “Security and privacy for cloud-based iot: Challenges,” *IEEE Communications Magazine*, vol. 55, no. 1, pp. 26–33, 2017.

[71] Statista. “Global iot and non-iot connections 2010-2025.” (2021), [Online]. Available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/ (visited on 10/14/2021).

[72] ——, "Global iot connections data volume 2019 and 2025." (2020), [Online]. Available: https://www.statista.com/statistics/1017863/worldwide-iot-connected-devices-data-size/ (visited on 10/14/2021).

[73] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 414–454, 2013.

[74] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[75] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for internet of things: A primer," *Digital Communications and Networks*, vol. 4, no. 2, pp. 77–86, 2018.

[76] R. Bruschi, P. Lago, G. Lamanna, C. Lombardo, and S. Mangialardi, "Openvolcano: An open-source software platform for fog computing," in *2016 28th International Teletraffic Congress (ITC 28)*, IEEE, vol. 2, 2016, pp. 22–27.

[77] Microsoft. "Azure iot edge." (2019), [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/ (visited on 11/02/2021).

[78] Amazon. "Aws greengrass." (2019), [Online]. Available: https://aws.amazon.com/greengrass/ (visited on 11/02/2021).

[79] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud paas architecture based on raspberry pi clusters," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, IEEE, 2016, pp. 117–124.

[80] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "Fog computing conceptual model," NIST, Tech. Rep. Special Publication (NIST SP) 500-325, 2018.

[81] G. Merlino, R. Dautov, S. Distefano, and D. Bruneo, "Enabling workload engineering in edge, fog, and cloud computing through openstack-based middle-

ware," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, p. 28, 2019.

[82] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Data agility through clustered edge computing and stream processing," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 7, pp. 1–1, 2021.

[83] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2016, pp. 1–13.

[84] M. Ahmad, J. S. Alowibdi, and M. U. Ilyas, "Viot: A first step towards a shared, multi-tenant iot infrastructure architecture," in *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, IEEE, 2017, pp. 308–313.

[85] S Senthil Kumaran, *Practical LXC and LXD: linux containers for virtualization and orchestration*. Springer, 2017.

[86] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, "Picasso: A lightweight edge computing platform," in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, IEEE, 2017, pp. 1–7.

[87] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[88] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, 2008, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=1344209.1344217.

[89] B Asvija, R Eswari, and M. Bijoy, "Security in hardware assisted virtualization for cloud computingstate of the art issues and challenges," *Computer Networks*, vol. 151, pp. 68–92, 2019.

[90] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," *International Journal of Innovative Technology and Exploring Engineering*, vol. 2, no. 3, pp. 222–225, 2013.

[91] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, ACM, vol. 41, 2007, pp. 275–287.

[92] M. Zabaljáuregui, "Hardware assisted virtualization intel virtualization technology," pp. 1–54, 2008. [Online]. Available: https://lettieri.iet.unipi.it/virtualization/Vtx.pdf (visited on 01/02/2021).

[93] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.

[94] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ACM, 2017, p. 11.

[95] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017.

[96] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," *Haifux, May*, vol. 186, 2013.

[97] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, Citeseer, vol. 1, 2006, pp. 101–112.

[98] M. A. Babar and B. Ramsey, "Understanding container isolation mechanisms for building security-sensitive private cloud," CREST, University of Adelaide, Australia, Tech. Rep., 2017.

[99] Z. Rejiba, X. Masip-Bruin, and E. Marin-Tordera, "A survey on mobility-induced service migration in the fog, edge, and related computing paradigms," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–33, 2019.

[100] E. N. Ciftcioglu, K. S. Chan, R. Urgaonkar, S. Wang, and T. He, "Security-aware service migration for tactical mobile micro-clouds," in *MILCOM 2015-2015 IEEE Military Communications Conference*, IEEE, 2015, pp. 1058–1063.

[101] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino, "Companion fog computing: Supporting things mobility through container migration at the edge," in *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, IEEE, 2018, pp. 97–105.

[102] C. Anderson, "Docker [software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[103] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in *Proceedings of the 18th international conference on distributed computing and networking*, 2017, pp. 1–10.

[104] C. Puliafito, E. Mingozzi, and G. Anastasi, "Fog computing for the internet of mobile things: Issues and challenges," in *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, IEEE, 2017, pp. 1–6.

[105] Y. Jiang, Z. Huang, and D. H. Tsang, "Challenges and solutions in fog computing orchestration," *IEEE Network*, vol. 32, no. 3, pp. 122–129, 2017.

[106] D. Puthal, M. S. Obaidat, P. Nanda, M. Prasad, S. P. Mohanty, and A. Y. Zomaya, "Secure and sustainable load balancing of edge data centers in fog computing," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 60–65, 2018.

[107] Y. Nan, W. Li, W. Bao, F. C. Delicato, P. F. Pires, Y. Dou, and A. Y. Zomaya, "Adaptive energy-aware computation offloading for cloud of things systems," *IEEE Access*, vol. 5, pp. 23 947–23 957, 2017.

[108] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[109] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ACM, 2014, pp. 68–81.

[110] M. Peuster, H. Karl, and S. van Rossem, "Medicine: Rapid prototyping of production-ready network services in multi-pop environments," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490.

[111] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[112] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the internet of things to the web of things: Resource-oriented architecture and best practices," in *Architecting the Internet of things*, Springer, 2011, pp. 97–129.

[113] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2019.

[114] C. Ferris and J. Farrell, "What are web services?" *Communications of the ACM*, vol. 46, no. 6, p. 31, 2003.

[115] A. S. Mohamed and C. Al-Atroshi, "Adaptability of soa in iot services–an empirical survey," *International Journal of Computer Applications*, vol. 975, p. 8887, 2018.

[116] D. Box *et al.*, "Simple object access protocol (soap) 1.1," W3C, Tech. Rep., 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[117] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl) 1.1," Tech. Rep., 2001. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[118] M. A. G. Maureira, D. Oldenhof, and L. Teernstra, *Thingspeak–an api and web service for the internet of things*, 2014. [Online]. Available: https://staas.home.xs4all.nl/t/swtr/documents/wt2014_thingspeak.pdf.

[119] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a multi-faceted metric for service design," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 911–920.

[120] C. Prehofer and I. Gerostathopoulos, "Modeling restful web of things services: Concepts and tools," in *Managing the Web of Things*, Elsevier, 2017, pp. 73–104.

[121] Z. Shelby, "Embedded web services," *IEEE Wireless Communications*, vol. 17, no. 6, pp. 52–57, 2010.

[122] A.-R. Breje, R. Győrödi, C. Győrödi, D. Zmaranda, and G. Pecherle, "Comparative study of data sending methods for xml and json models," *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, vol. 9, no. 12, pp. 198–204, 2018.

[123] D. Yazar and A. Dunkels, "Efficient application integration in ip-based sensor networks," in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, 2009, pp. 43–48.

[124] D. Guinard, I. Ion, and S. Mayer, "In search of an internet of things service architecture: Rest or ws-*? a developers' perspective," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Springer, 2011, pp. 326–337.

[125] B. N. Silva, M. Khan, and K. Han, "Integration of big data analytics embedded smart city architecture with restful web of things for efficient service provi-

sion and energy management," *Future generation computer systems*, vol. 107, pp. 975–987, 2020.

[126] A. Tiberkak, A. Hentout, and A. Belkhir, "Lightweight remote control of distributed web-of-things platforms: First prototype," in *2020 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)*, IEEE, 2021, pp. 103–108.

[127] R. Yugha and S. Chithra, "A survey on technologies and security protocols: Reference for future generation IoT," *Journal of Network and Computer Applications*, vol. 169, p. 102 763, 2020. DOI: 10.1016/j.jnca.2020.102763. [Online]. Available: https://doi.org/10.1016/j.jnca.2020.102763.

[128] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring {https} adoption on the web," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1323–1338.

[129] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic certificate management environment (ACME)," IETF, Tech. Rep. RFC 8555, 2019. DOI: 10.17487/rfc8555. [Online]. Available: https://doi.org/10.17487/rfc8555.

[130] E. F. Kfoury, D. Khoury, A. AlSabeh, J. Gomez, J. Crichigno, and E. Bou-Harb, "A blockchain-based method for decentralizing the acme protocol to enhance trust in pki," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, IEEE, 2020, pp. 461–465.

[131] C. Tiefenau, E. von Zezschwitz, M. Häring, K. Krombholz, and M. Smith, "A usability evaluation of let's encrypt and certbot: Usable security done right," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1971–1988.

[132] J. Aas *et al.*, "Let's encrypt: An automated certificate authority to encrypt the entire web," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2473–2487.

[133] D. Skvorc, M. Horvat, and S. Srbljic, “Performance evaluation of websocket protocol for implementation of full-duplex web streams,” in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, IEEE, 2014, pp. 1003–1008.

[134] A. R. Biswas and R. Giaffreda, “IoT and cloud convergence: Opportunities and challenges,” in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, IEEE, 2014, pp. 375–376.

[135] E. G. Petrakis, S. Sotiriadis, T. Soultanopoulos, P. T. Renta, R. Buyya, and N. Bessis, “Internet of things as a service (itaas): Challenges and solutions for management of sensor data on the cloud and the fog,” *Internet of Things*, vol. 3, pp. 156–174, 2018.

[136] I. Baldini *et al.*, “Serverless computing: Current trends and open problems,” in *Research advances in cloud computing*, Springer, 2017, pp. 1–20.

[137] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural implications of function-as-a-service computing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1063–1075.

[138] Amazon Web Services. “Aws lambda.” (2022), [Online]. Available: https://aws.amazon.com/lambda/faqs/ (visited on 01/19/2022).

[139] G. McGrath and P. R. Brenner, “Serverless computing: Design, implementation, and performance,” in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2017, pp. 405–410.

[140] Amazon Web Services. “Amazon ec2 on-demand pricing.” (2022), [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/ (visited on 01/19/2022).

[141] ——, “Aws lambda pricing.” (2022), [Online]. Available: https://aws.amazon.com/lambda/pricing/ (visited on 01/19/2022).

[142] G. Adzic and R. Chatley, “Serverless computing: Economic and architectural impact,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 884–889.

[143] D Ferguson, C Pahl, M Helfert, A Jindal, and M Gerndt, “From devops to noops: Is it worth it?” In *Cloud Computing and Services Science10th International Conference, CLOSER 2020, Prague, Czech Republic, May 7–9, 2020, Revised Selected Papers*, vol. 1399, 2021, pp. 178–202.

[144] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “The rise of serverless computing,” *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[145] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[146] S. Nastic *et al.*, “A serverless real-time data analytics platform for edge computing,” *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[147] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, “Towards a serverless platform for edge {ai},” in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[148] S. Nastic and S. Dustdar, “Towards deviceless edge computing: Challenges, design aspects, and models for serverless paradigm at the edge,” in *The Essence of Software Engineering*, Springer, Cham, 2018, pp. 121–136.

[149] A. Glikson, S. Nastic, and S. Dustdar, “Deviceless edge computing: Extending serverless computing to the edge of the network,” in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–1.

[150] P. Persson and O. Angelsmark, “Kappa: Serverless IoT deployment,” in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pp. 16–21.

[151] M. Blackstock and R. Lea, “Toward a distributed data flow platform for the web of things (distributed node-red),” in *Proceedings of the 5th International Workshop on Web of Things*, 2014, pp. 34–39.

[152] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust, "Mobile-edge computing architecture: The role of mec in the internet of things," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 84–91, 2016.

[153] R. Beudert, L. Juergensen, and J. Weiland, "Understanding smart machines: How they will shape the future," *Schneider-Electric*, 2015.

[154] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[155] L. D. Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Trans. on industrial informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[156] V. C. Gungor and G. P. Hancke, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *IEEE Transactions on industrial electronics*, vol. 56, no. 10, pp. 4258–4265, 2009.

[157] F. Battaglia and L. L. Bello, "A novel jxta-based architecture for implementing heterogenous networks of things," *Computer Communications*, vol. 116, pp. 35–62, 2018.

[158] M. Craciunescu, S. Mocanu, and G. Manea, "Towards practical integration of wsn in cloud dedicated to smart environments," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, IEEE, 2017, pp. 447–452.

[159] Y. Guan, J. Shao, G. Wei, and M. Xie, "Data security and privacy in fog computing," *IEEE Network*, vol. 32, no. 5, pp. 106–111, 2018. DOI: 10.1109/MNET.2018.1700250.

[160] I. Azimi, A. Anzanpour, A. M. Rahmani, T. Pahikkala, M. Levorato, P. Liljeberg, and N. Dutt, "Hich: Hierarchical fog-assisted computing architecture for healthcare iot," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, 2017, ISSN: 1539-9087. DOI: 10.1145/3126501.

[161] J. Zheng and M. J. Lee, "A comprehensive performance study of ieee 802.15.4," *Sensor network operations*, vol. 4, pp. 218–237, 2006.

[162] G. Mulligan, "The 6lowpan architecture," in *Proceedings of the 4th workshop on Embedded networked sensors*, 2007, pp. 78–82.

[163] I. Tomić and J. A. McCann, "A survey of potential security issues in existing wireless sensor network protocols," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1910–1923, 2017. DOI: 10.1109/JIOT.2017.2749883.

[164] S. Heron, "Advanced encryption standard (aes)," *Network Security*, vol. 2009, no. 12, pp. 8–12, 2009.

[165] A. Di Nisio, T. Di Noia, C. G. C. Carducci, and M. Spadavecchia, "High dynamic range power consumption measurement in microcontroller-based applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 65, no. 9, pp. 1968–1976, 2016.

[166] Keysight, *MSOX3012A Mixed Signal Oscilloscope*. [Online]. Available: https://www.keysight.com/en/pdx-x201841-pn-MSOX3012A/mixed-signal-oscilloscope-100-mhz-2-analog-plus-16-digital-channels.

[167] K. Biswas, V. Muthukkumarasamy, X.-W. Wu, and K. Singh, "Performance evaluation of block ciphers for wireless sensor networks," in *Adv. Comp. and Communication Technologies*, Springer, 2016, pp. 443–452.

[168] A. Famulari, F. Longo, G. Campobello, T. Bonald, and M. Scarpa, "A simple architecture for secure and private data sharing solutions," in *2014 IEEE Symp. on Computers and Communications (ISCC)*, 2014, pp. 1–6. DOI: 10.1109/ISCC.2014.6912518.

[169] R. Srivastava, S. M. Ladwa, A. Bhattacharya, and A. Kumar, "A fast and accurate performance analysis of beaconless ieee 802.15. 4 multi-hop networks," *Ad Hoc Networks*, vol. 37, pp. 435–459, 2016.

[170] G. W. Stewart, "On the early history of the singular value decomposition," *SIAM Review*, vol. 35, no. 4, pp. 551–566, 1993.

[171] G. Campobello, A. Segreto, and S. Serrano, "Data Gathering Techniques for Wireless Sensor Networks: A Comparison," *Int. J. of Distributed Sensor Networks*, vol. 12, 4156358:1–4156358:17, 2016. DOI: 10.1155/2016/4156358.

[172] G. Campobello, S. Serrano, A. Leonardi, and S. Palazzo, "Trade-Offs between Energy Saving and Reliability in Low Duty Cycle Wireless Sensor Networks Using a Packet Splitting Forwarding Technique," *EURASIP Journal on Wireless Communications and Networking*, vol. 2010, 8:1–8:11, 2010. DOI: 10.1155/2010/932345.

[173] A. Varga, "Omnet++," in *Modeling and tools for network simulation*, Springer, 2010, pp. 35–59.

[174] K. Montgomery, R. Candell, M. Hany, and Y. Liu, *Wireless User Requirements for the Factory Workcell*. National Institute of Standards and Technology (NIST), 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ams/NIST.AMS.300-8r1-upd.pdf.

[175] G. Dhand and S. Tyagi, "Data aggregation techniques in wsn:survey," *Procedia Computer Science*, vol. 92, pp. 378–384, 2016, 2nd Int. Conf. on Intelligent Computing, Communication & Convergence, ICCC 2016, 24-25 January 2016, Bhubaneswar, Odisha, India, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.07.393.

[176] N. John and A Jyotsna, "A survey on energy efficient tree-based data aggregation techniques in wireless sensor networks," in *2018 Int. Conf. on Inventive Research in Computing Applications (ICIRCA)*, 2018, pp. 461–465. DOI: 10.1109/ICIRCA.2018.8597222.

[177] G. Campobello, A. Leonardi, and S. Palazzo, "On the use of chinese remainder theorem for energy saving in wireless sensor networks," in *2008 IEEE International Conference on Communications*, IEEE, 2008, pp. 2723–2727.

[178] G. Campobello, A. Segreto, S. Zanafi, and S. Serrano, "Rake: A simple and efficient lossless compression algorithm for the internet of things," in *2017 25th*

European Signal Processing Conference (EUSIPCO), 2017, pp. 2581–2585. DOI: 10.23919/EUSIPCO.2017.8081677.

[179] Z. Benomar, G. Campobello, F. Longo, G. Merlino, and A. Puliafito, "A new fog-enabled wireless sensor network architecture for industrial internet of things applications," in *24th IMEKO TC4 International Symposium*, 2020, pp. 179–184.

[180] G. Campobello, A. Segreto, and N. Donato, "A new frequency estimation algorithm for iiot applications and low-cost instrumentation," in *2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, IEEE, 2020, pp. 1–5.

[181] F. Giust, X. Costa-Perez, and A. Reznik, "Multi-access edge computing: An overview of etsi mec isg," *IEEE 5G Tech Focus*, vol. 1, no. 4, p. 4, 2017.

[182] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, 2005.

END