Dipartimento di Ingegneria Elettrica, Elettronica e Informatica

Dottorato di Ricerca Internazionale in
Ingegneria Informatica e delle Telecomunicazioni
XXVIII Ciclo

Tesi di Dottorato

SENSING AND ACTUATION AS A SERVICE, A DEVICE-CENTRIC PARADIGM FOR THE IOT: ANALYSIS, DESIGN AND CASE STUDIES

Ing. Giovanni Merlino

Coordinatore	Tutor
Chiar.ma Prof. V. Carchiolo	Chiar.ma Prof. V. Carchiolo

to my families

SOMMARIO

La crescita impressionante, che non accenna ad arrestarsi, del numero di dispositivi distribuiti e connessi alla rete globale in quella che viene definita l'Internet delle Cose, ovvero "Internet of Things" (IoT), richiede la disponibilità di tecniche per la gestione di infrastruttura hardware in grado di affrontare un livello di complessità talmente schiacciante, specialmente alla luce del crescente impatto dell'economia della condivisione e del ruolo giocato dalla cosiddetta "coda lunga" (*long tail*). In questo contesto, l'approccio *as-a-Service* fornisce meccanismi noti ed affidabili per il provisioning di infrastrutture e servizi, ed una sfida interessante sta nel valutarne l'applicabilità all'istanziazione e gestione del ciclo di vita di un'infrastruttura dinamica, possibilmente virtualizzata, composta da sensori (ed attuatori). A parte la flessibilità di poter noleggiare questo tipo di risorse secondo il modello delle *utilities*, fornendo l'accesso al livello più basso possibile, il percorso di ricerca intrapreso per questa dissertazione è pensato per fornire un respiro ancora più ampio in relazione agli scenari presi in considerazione, a partire da una piattaforma per il *mobile crowdsensing* di tipo opportunistico e cooperativo fino ad arrivare ad un modello per Smart City "definite dal software", laddove la riconfigurazione dinamica del "cablaggio" tra oggetti in ultima istanza abilita anelli di controllo retroazionato su scala geografica, e su richi-

esta. Eppure nessuno di questi traguardi, ed in particulare la facoltà di plasmare l'ambiente circostante, risulta essere obiettivo effettivamente raggiungibile in assenza di strumenti per interagire bidirezionalmente, e col massimo controllo esercitabile, con sistemi fisici attraverso sensori ed attuatori remoti a portata di mano. La premessa dunque risiede nell'impostare la ricerca da una prospettiva **incentrata sul dispositivo**. I tentativi di gestire l'IoT a livello di infrastruttura e piattaforma, il framework **Stack4Things** così come altri risultati non sono altro che esiti di questo approccio.

In questa dissertazione sono quindi presentate le analisi degli scenari d'interesse, i dettagli relativi alle architetture ed i casi di studio affrontati, descrivendo ed evidenziando le scelte progettuali effettuate.

ABSTRACT

The huge and steady growth in the number of distributed devices connected to the global network as a so-called Internet of Things (IoT) calls for infrastructure management techniques able to deal with this overwhelming complexity, especially in light of the growing impact of the sharing economy and the role played by the so-called *long tail*. In this context, the as-a-Service approach provides well investigated mechanisms for infrastructure and service provisioning, and an interesting challenge lies in evaluating its application to the instantiation and lifecycle management of a dynamic, possibly virtualized, infrastructure of sensing (and actuation) resources. Apart from the flexibility of renting this kind of resources according to the utility model, by providing access at the lowest level where possible, the research path undertaken in this dissertation is meant to provide even bigger scope to the scenarios under consideration, from a platform for opportunistic and cooperative mobile crowdsensing to a model for Software-Defined Smart Cities, where dynamic reconfiguration of the wiring among Things ultimately enables wide-area feedback control loops on demand. Yet none of these outcomes, and in particular shaping the surrounding environment, may be achievable without means to interact bidirectionally, and with the greatest control that may be exerted, with physical systems through remote sensors and actuators

at one's own fingertips. The premise then lies in engaging the research from a **device-centric** perspective. The proposed infrastructure and platform takes on IoT, the **Stack4Things** framework as well as other results are then outcomes of this approach.

Analytical descriptions of the scenarios, details of the architectures and investigated case studies therefore are here provided, while also reporting and highlighting design choices.

CONTENTS

Introduction **1**

1 Sensing and Actuation as a Service **5**

1.1 Introduction 5
1.2 Overview: vision, approach 7
1.2.1 The big picture 7
1.2.2 Device-centric paradigms 9
1.3 A device-centric stack 12
1.3.1 High-level IaaS architecture 13
1.3.2 Core modules 15
1.4 Basic device interactions 21
1.5 Proof of concept 24
1.5.1 Case Study 24
1.5.2 Testing 27
1.6 Related work 31

2 Stack4Things: a framework for SAaaS **37**

2.1 Introduction 37
2.2 Sensing-and-Actuation-as-a-Service 38
2.3 Background 40
2.4 Stack4Things architecture 44

2.4.1 Board-side 45
2.4.2 Cloud-side - control and actuation 47
2.4.3 Cloud-side - sensing data collection 48
2.5 Stack4Things REST API 49
2.6 Use cases 52
2.6.1 Use case: provide the list of nodes registered to the Cloud 52
2.6.2 Use case: retrieve the current value of a pin on a specific board 53
2.6.3 Use case: create an SSH connection toward a node 55
2.6.4 Use case: store readings from a sensor in the Cloud 57
2.6.5 Use case: inject a CEP rule and set a reaction . 59

3 Mobile CrowdSensing as a Service: a platform for opportunistic sensing 67
3.1 Introduction and motivations 67
3.2 Preliminary concepts and related work 71
3.2.1 Mobile Crowd-Sensing 71
3.2.2 MCS taxonomy 72
3.2.3 Related work 74
3.3 MCSaaS paradigm 76
3.3.1 Vision 76
3.3.2 Stack 80
3.4 Infrastructure 82
3.5 Platform: MCSaaS module 85
3.6 Setup and deployment of MCS applications 88
3.6.1 MCSaaS platform setup 88
3.6.2 MCS application configuration and deployment 91
3.7 The MCSaaS implementation 94
3.8 MCS app: case study 96

3.8.1 Pothole mapping 98
3.8.2 Traffic monitoring 99
3.8.3 Testing and evaluation 100

4 A crowd-cooperative approach for ITS 107
4.1 Introduction and motivations 107
4.2 Background and related work 109
4.2.1 An overview of ITS 109
4.2.2 MCS for ITS 111
4.3 A novel cooperative strategy 112
4.3.1 A distributed MCS pattern 112
4.3.2 Stigmergic approach 115
4.4 ITS implementation 120
4.4.1 Motivating example 121
4.4.2 MoCSACO application to ITS 122
4.5 Modeling and evaluation 125
4.5.1 MA model of the MoCSACO algorithm 125
4.5.2 Results 128

5 Network Function Virtualization for CPS 135
5.1 Introduction 135
5.2 Network virtualization for IoT 137
5.2.1 Tunneling 137
5.2.2 Layering 139
5.3 A real-world example 141

6 Software-Defined City:
an elastic model for the Smart City 149
6.1 Introduction 149
6.2 Related work 151
6.3 Overview of the approach 152
6.3.1 Data Plane: Cyber-Physical Systems 153

6.3.2 Control Plane: Smart Cities 154
6.4 Reference architecture 158
6.4.1 Requirements 158
6.4.2 Sensing and Actuation as a Service for SDC . . 161
6.5 Use case 162

Conclusions and future work **167**

LIST OF FIGURES

1.1 A generalized Von Neumann architecture 8
1.2 Device-centric stack: architecture and deployment . . 14
1.3 SAaaS Hypervisor: architecture 16
1.4 SAaaS Hypervisor: modules 16
1.5 SAaaS Planning Agent: architecture 19
1.6 SAaaS Task Manager: architecture 20
1.7 SAaaS: resource acquisition AD 22
1.8 SAaaS: request submission AD 34
1.9 SAaaS: submission management AD 35
1.10 SAaaS: observation access AD 35

2.1 Stack4Things: reference scenario 39
2.2 Stack4Things: distributed system 42
2.3 Stack4Things: board-side architecture 44
2.4 Stack4Things: Cloud-side architecture 45
2.5 S4T: listing of registered nodes 62
2.6 S4T: retrieving current value of a pin 63
2.7 S4T: creation of an SSH connection 64
2.8 S4T: storing readings from a sensor 65
2.9 S4T: injection of a CEP rule 66

3.1 MCS: application scenario 72
3.2 MCSaaS: scenario 77
3.3 MCSaaS: stack 81
3.4 SAaaS: reference architecture 83
3.5 MCSaaS module 86
3.6 MCSaaS: initial platform setup AD 89
3.7 MCSaaS: app negotiation and platform bootstrap AD 91
3.8 MCSaaS: app deployment AD 93
3.9 MCSaaS-driven app: deployment scenario 96
3.10 MCS and MCSaaS emulation: contribution sampling . 104

4.1 MCS patterns: centralized and distributed 113
4.2 MoCSACO: activity diagram 117
4.3 Distance graph: originating road map 121
4.4 Markovian Agents: categories 126
4.5 Results: pheromone distributions 130
4.6 Results: traffic flow intensities 133

5.1 Functional diagram of WS-based reverse tunneling . . 146
5.2 Functional diagram of tunnel-based bridging over WS 147
5.3 Virtual networking use case: workflow 148

6.1 Cyber-Physical Systems 153
6.2 Cyber-City System function virtualization 155
6.3 Control logic: approaches 156
6.4 Software Defined City 157
6.5 SD City as closed-loop system 158

LIST OF TABLES

1.1 SAaaS results: contribution overhead 27
1.2 SAaaS results: individual operation performance . . . 28

2.1 IoTronic REST API 51

3.1 Taxonomy of MCS applications 74
3.2 MCSaaS scenario: actors 79

INTRODUCTION

In recent years, the Internet of Things (IoT) has emerged as one of the hottest trends in ICT, thanks to the proliferation of field-deployed, dispersed, and heterogeneous sensor- and actuator-hosting platforms. 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. Thus such a stimulating albeit challenging scenario calls for suitable approaches, technologies and solutions. In particular, on the verge of explosive growth in demand and adoption, vertical frameworks and closed siloes are to be considered unsustainable in the long term and are best avoided. Moreover, beyond interoperability and functional scope, the metric by which deployments and operations may be considered large is naturally going to be redefined according to the unprecedented sheer scale.

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. This basic level of service should not preclude focusing on more interesting usage patterns, such as opportunistic exploitation or on-demand custom wiring of a subset of the pool of globally available resources.

Structure of this Dissertation

In this work some approaches and solutions are presented to address the aforementioned challenges.

This dissertation is organised in six chapters, excluding the closing one and this introduction, as outlined in the following:

- Chapter 1 introduces the basic concepts behind *Sensing and Actuation as a Service* (SAaaS), a device-centric, service-oriented, Cloud-mediated approach to sensing and actuation, including a high-level architecture and some interaction models. It also describes a proof of concept.

- Chapter 2 presents *Stack4Things* (S4T), an extension to the OpenStack framework for SAaaS, able to enable remote control and actuation, as well as data collection. In particular, the section includes a description of S4T architecture, a subset of its REST APIs as well as outlining some use cases.

- Chapter 3 explores the feasibility of supporting and exposing a platform for Mobile CrowdSensing (MCS) applications on top of SAaaS-enabled infrastructure, a so-called *MCS as a Service* (MCSaaS), proposing a two-layered approach to node-side deployment as a solution to address contributor-owned devices' churn, also presenting platform-relevant interactions, two case studies as well as a preliminary testing and evaluation effort.

- Chapter 4 proposes a *crowd-cooperative strategy* which is expected in the future to leverage crowd-powered sensing-oriented platforms such as MCSaaS. Here the approach is specifically geared toward vehicular traffic within the context of Intelligent Transportation Systems, featuring a real-world example, a model and a preliminary evaluation.

- Chapter 5 describes remoting, tunneling as well as layering models and mechanisms of *Cloud-enabled network virtualization for IoT*, implemented within S4T and meant to support and simplify the management of wide-area heterogeneous sensor-/actuator-hosting nodes. A real-world example and some considerations on the overhead are outlined.

- Chapter 6 introduces the idea behind an elastic model for Smart Cities, ultimately leading to the perspective of a *Software-Defined City* as instance of a bundle of urban-scale, on-demand, hierarchical feedback loops based on (possibly virtualized) Cyber-Physical Systems. Requirements for such a scenario are there enumerated, as well as some use cases described.

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

This dissertation interpolates material from several published papers by the author, based on [1], [2] and [3], respectively for Chapters 1 and 3. Instead Chapters 2, 5 and 6 include material which has already been accepted, yet which publication is still pending, even if some results have already been disseminated in [4] and [5]; whereas Chapter 4 material still under review for publication, albeit already presented in a peer-reviewed workshop.

Acknowledgements

A subset of the research efforts described in this dissertation originate from activities partially funded by the *S.I.Mon.E.* project under Sicilian POR programme, *Smart Health* (04a2_C, cluster OSDH-SMART FSE-STAYWELL) and *SIGMA* (01_00683) projects under Italian PON programme, and *BEACON* project under European Union's Horizon 2020 Research and Innovation programme (grant agreement n. 644048).

Requirements, specifications and deliverables of any kind for the relevant projects are available on the respective project websites.

CHAPTER

ONE

SENSING AND ACTUATION AS A SERVICE

1.1 Introduction

The current ICT scenario is dominated by large and complex systems, paving the way to a ZettaBytes (BigData) [6] landscape made up of billion/trillion objects and devices (Internet of Things - IoT) [7]. Devices usually equipped by a wide range of sensing resources and advanced computing, storage and communication capabilities, often referred to as *smart*, populate this scenario, thus highlighting the need for facilities for their management. Efforts categorized under the IoT umbrella term take some steps in this direction but mainly from a networking perspective, where the communication among heterogeneous things is crucial. However, other important aspects related to sensing, computing and data resource management have to be addressed in these contexts.

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.

Taking into account the massive amounts of data this kind of 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 storage, centralized treatment, analytics, inference, etc. What is currently to be investigated more thoroughly is BigData at the lowest level, i.e., closer to the source. In order to understand well the issues and requirements in terms of the data-originating infrastructure, a deeper look at the underlying distributed systems is called for, possibly re-evaluating approaches, assumptions and theoretical frameworks where needed.

According to the above considerations, here follows a conceptual framework for processing duties to be pushed as close as possible to data for sensing-originated BigData loads, thus leading to a *device-centric* approach to distributed sensing systems, at the other side of the spectrum where current *data-centric* approaches are positioned. The device-centric approach, in line with the BigData principle of bringing computing near to data, proposes to inject intelligence on sensing devices in order to collect, preprocess, aggregate and mine sensed data at the source, before forwarding something useful. To this purpose a Cloud/IaaS approach is here proposed for adoption, since it allows to easily customize sensing resources through abstraction and virtualization solutions thus enabling the device-centric view.

This way, in the chapter a conjecture is put forward about how a sensing IaaS infrastructure can support a sensing-enabled BigData scenario exploiting the device-centric approach, under the guise of a Sensing IaaS stack, as a platform enabler for BigData approaches, providing basic services for developing data processing facilities and

APIs on top of the them, in an everything as a service (XaaS) [8, 9] philosophy.

The implementation of the core modules of this device-centric approach to sensing BigData management has been done for Android-based smartphones. The functional testing of such an implementation has been also performed through a surveillance app in order to evaluate the feasibility of this approach, also gauging certain metrics of interest according to different viewpoints (end user, contributors and providers).

This effort is now contextualized and extended into a BigData scenario, proposing a new device-centric approach. Based on this perspective, an architecture is henceforth provided while also developing the core building blocks of a BigData sensing IaaS stack starting from the Cloud-enabling artifacts and modules. Although different solutions for the management of resources and data in sensing infrastructures have been specified so far [10, 11], here progress beyond the state of the art consists in investigating the problem from a different, BigData angle, by adopting a device-centric approach rather than a more typical data-centric one.

This way, the overarching vision about the BigData problem is specified when it relates to (geographically dispersed) sensing data collection nodes, describing the main concepts behind the device-centric approach. A generalization of the Von Neumann architecture to distributed systems of this kind is put forward.

1.2 Overview: vision, approach

1.2.1 The big picture

From a high-level, abstract perspective, a distributed system can be considered as a generalization of the Von Neumann architecture at

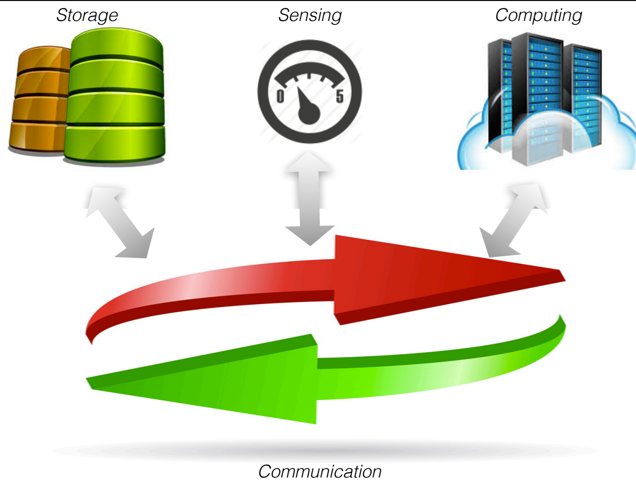


Figure 1.1: *A generalized Von Neumann architecture*

larger scale than a single computing system, as the one shown in Figure 1.1. In this architecture, computing subsystems provide processing facilities, storage subsystems make up the memory hierarchy, while I/O operations are performed by sensing and actuation subsystems. The communication subsystem, i.e., one or more buses per the traditional architecture, mainly recognizable under the guise of networking infrastructure, allows the interconnection among members of each class of the distributed system, as well as different instances of the same subsystem.

In this architecture, differently from the original Von Neumann one, all the elements, except the communication one, could be considered as optional. This way, a wide range of distributed computing (Cluster, Grid, Cloud, etc.), storage (NAS, SAN, etc.), sensing (WSN, M2M, IoT, etc.) infrastructure and solutions can be represented through the high level model shown in Figure 1.1. Furthermore, hybrid, hierarchical, multi-tiered architectures, including distributed sensing, storage and/or computing systems, may be comprised in this model. From the sensing angle, this architecture can properly repre-

sent a sensing system, where sensor data can be collected in specific storage subsystems and processed by computing subsystems, all interconnected through communication subsystems such as networks. Also sensor networks are included in the model, considering as communication subsystem an ad-hoc network, often wireless, connecting sensors to the base station that could store and process data.

The main bottleneck of the architecture of Figure 1.1 is the communication system, which also has significant impact on the overall dependability. As a result, it is necessary to minimize the usage of the communication subsystem in a distributed sensing system.

1.2.2 Device-centric paradigms

Putting sensing-related duties into context, a way to manage effectively huge collections of incoming data (Big Data) consists in minimizing the communication overhead by bringing computation closer to data, and not the other way around, as already discussed. The approaches usually adopted in sensing resource management [9, 12, 13] can be framed into the *data-centric* category since the only operations provided are data manipulation ones, considering sensing devices as just mere source of data. A step forward, in accordance with the BigData locality principle, is to provide to user actual, even if virtual, sensing resources instead of the data they generate adopting a *device-centric* paradigm.

Although the data-centric approach can be successfully applied to different contexts, the device-centric one manifests several benefits in comparison to the former:
decentralized control - distributed policies may be deployed on virtual sensors and actuators through customization features allowing to deploy on them user-customized software; *on-board data pre-filtering and processing* - data are filtered and/or processed on the device; *reduced*

number of data transfer - a direct link between the user and the sensing resource is established thus requiring just one data transfer, while in the data-centric approach at least two transfers are required since data are first stored into a database that provides them to the user; *composition, repurposing* - it allows to aggregate, compose and/or repurpose sensing resources; *higher security* - than the data-centric approach since it is possible to shift the burden from the resource to the network and vice-versa according to the required level of security and the device capabilities; *information dissemination* - data are disseminated through the distributed sensing infrastructure thus allowing to implement distributed data delivery algorithms optimized on the topology improving the transfer performance.

Exploring such perspective, in the following this Von Neumann architecture (VNA) generalization described above will be detailed by stretching a variety of modern patterns and models into this theoretical framework. Taking BigData as target domain, its whole point is about "reliably processing unbounded streams of data". Even recently developed advanced solutions such as, e.g., real-time streaming BigData systems, while being more tuned to the scenario under consideration, i.e., less batch-oriented and more node topology-dependent, are anyway wholly about spearheading once more the approach of *bringing computation closer to data*, ideally for local (or near-local) processing.

A requirement lies in the ability to inject at runtime custom logic for node-local computation, possibly including variants enabled for (explicit) node cooperation. In this sense, an equivalent for VNA would be coprocessor-based acceleration (possibly even in cooperative mode for multi-adapter configuration) and/or other CPU offloading engine, such as those normally available in high-end NICs. In turn this kind of scenario, apart from the availability of dedicated peripherals, e.g., sensor-hosting boards in this case, calls for the ability for

such subsystems to be reprogrammable: typically the role of transient processing logic, such as, e.g., shaders when talking about GPU pipelines and their corresponding engines (so called *shader units*).

Such resources need to be managed, thus whereas the control unit of a CPU performs this function in terms of VNA, a centralized management subsystem, such as an IaaS solution, would take on a similar role in the aforementioned generalization. Sensor-hosting environment virtualization may be considered a valuable addition, where multiplexing virtual sensors over physical ones, and composing instances (resource *slices*) into more complex ones, would be advanced mechanisms, in line with the aforementioned Cloud-oriented vision. In VNA terms, virtual instances of sensors play the role of I/O virtualization.

To complete this overview, another paradigm worth mentioning is that of Software Defined Networking (SDN) [14], or more in general SD* (Software Defined Radio, etc.), in this case declined as Software Defined Sensing (SDS). Indeed at the core of the SD* paradigm is both a separation between a layer dedicated to coordination of policies (so called *control plane*) and another dealing with mere execution (*data plane*, also called, e.g., forwarding plane, for SDNs). Whereas control in an SDN refers to the ability to deploy and switch to, e.g., a routing protocol, on a set of devices, the SDN forwarding fabric, typically hard-wired, just has to push packets around according to the policy such a choice of protocol encodes. When it comes to SDS, the reconfiguration is for a generalized control plane, made up of sensing, storage, processing and transmission user-defined group-wide policies, in comparison to the corresponding data plane, i.e. the aforementioned subsystems working according to such tunables, of a swarm of boards/slices, really a sort of cross-category policy engine.

Getting back to the sensing-related IaaS-like Cloud exposed above, the elastic on-demand allocation of resources it brings about may be considered as a mechanism to centrally manage and coordinate

slices making up a control plane, by leaving those available to (coordinated if needed) mass injection of custom logic. This means imposing whichever patterns or criteria match such separation of duties and concerns.

Where all these paradigms get pulled together, the overarching BigData problem may be reduced to a use case for control plane separation, in particular as a global policy, i.e., a set of directives to migrate computation as close as possible to the data to be processed.

1.3 A device-centric stack

From a device-centric perspective there is the need to offer a stack that provides readily available functionalities in the areas of sensor and actuator virtualization, and service-oriented provisioning to have (virtual) sensing resources available as endpoints, e.g., registered and enumerable, as well as actionable items, e.g., prone to code injection if needed. This is just a small set of Cloud-enabled functionalities, yet it is a first step toward the kind of device-centric architecture for BigData, ultimately leading to a generalized Von Neumann architecture, discussed in previous sections. More specifically, the following issues should be taken into account, and related functionalities provided, by the stack: *Sensor and actuator abstraction:* sensing and actuation resources have to be abstracted, providing a homogeneous view of heterogeneous sensors and actuators hosted by both mobiles and SNs; *Virtualization:* resources are also going to be virtualized, either restricting capabilities (thus exporting subsets), *repurposing* them or making compositions to obtain complex virtual sensors, or actuators on offer starting from the physical ones; Virtualization has to also provide adequate mechanisms for *customizing* the virtual resource provided, as well as isolation and similar security-related issues; *Node management:* adequate mechanisms and tools have to be pro-

vided to manage a sensing node, handling Cloud provider subscription, and implementing and enforcing policies merging device owner and Cloud provider objectives; *Service oriented/Cloud provisioning approach:* allowing end users (service providers, other infrastructure providers, organizations, etc.) to submit both functional and non-functional requests with requirements and specifications, without any knowledge of the system resources deployment. The provisioning of actual, even if virtual, sensing resource has to also ensure to end user the total control on the provided device. To this purpose a direct connection between end users and devices has to be established and facilities to support this mechanism have to be implemented. From the BigData perspective, this avoids most mediated interactions in data management thus minimizing roundtrips.

1.3.1 High-level IaaS architecture

To address all such requirements through a stack implementing the aforementioned functionalities the proposal here is to adopt and extend an IaaS approach to the sensing domain, at the core of the device-centric model. Through a sensing infrastructure Cloud stack physical (or even virtual) sensing and actuation resources can be provided on-demand, elastically, in an IaaS fashion for Cloud computing providing virtual machines.

By approaching sensor-based services from an IaaS perspective there are many other advantages to be reaped, such as the customization and virtualization possibilities this approach enables, including the ability to multiplex requests over (available) endpoints and deal elastically with hardware resource scarcity, in typical Cloud fashion.

This way, resources get to be fully exposed as infrastructure for the Cloud, while still preserving alternative mechanisms of interaction, such as data-driven mining-related services, which may simply

run in parallel, keeping one more avenue open for exploitation. Indeed other relevant standards, such as the IoT-A model [15], may be considered for extension by projecting the architectural considerations under discussion into these.

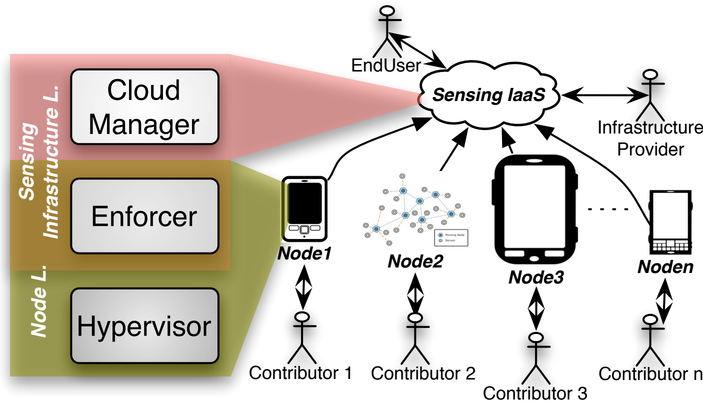


Figure 1.2: *Device-centric stack: architecture and deployment*

This way, starting from [16], the layered architecture shown in Figure 1.2 has been defined for the enablement of core mechanisms needed in a device-centric stack such as custom code upload to sensor boards, i.e., resource customization. It is composed of three modules: the Hypervisor, the Enforcer and the Cloud Manager ones, spanning across the node and the sensing infrastructure levels. The node level mainly provides functionalities for locally managing a node without any specific knowledge of being part of a sensing IaaS. At the management level the Cloud view is implemented by providing facilities for handling sensing resources as elements of the infrastructure, or ephemeral instances thereof, depending on whether the resource is a tangible, physical one, or a virtualized subset.

The *Hypervisor*, operating at node level, implements management, virtualization and customization mechanisms for sensing (and actuation) resources enrolled either from mobiles or SN nodes. At higher level, the *Cloud Manager* and the *Enforcer* deal with interaction mech-

anisms among nodes in the Cloud. The former is in charge of having the virtual sensing resources exposed via Web service interfaces and indexing resources. The latter implements enforcement mechanisms of global and local Cloud policies, a subscription management subsystem, and provides facilities for cooperation on overlay instantiation in particular, per the SDS approach.

As shown in Figure 1.2, the Hypervisor and the Enforcer are deployed into contributing nodes while the Cloud Manager is deployed into the Sensing IaaS Provider servers. From a device-centric perspective, the main basic functionalities characterizing the approach are the sensing resources abstraction, virtualization and customization. This is the reason why, in the following, the focus rests mainly on the Hypervisor core modules and related functionalities.

1.3.2 Core modules

Here the aim is to provide details about basic services and abstractions, as well as the corresponding components that are key enablers for the device-centric paradigm, i.e. the ones provided by the Hypervisor. It can be viewed as an essential component of this device-centric approach to sensing platforms, tasked with the management of sensor (and actuator) resources, as well as embedded processing and storage ones, by introducing virtualization and customization functionalities.

A modular, high-level layout of the Hypervisor architecture consists of four main building blocks: the Virtualization Unit, the Abstraction Unit, the Node Manager and the Adapter, as shown in Figure 1.3. The top layer of the Hypervisor is populated by the *Virtualization Unit*, whose core task is slicing, i.e., generating compatible partitioning schemes for a cluster of resources provided by the lower level Abstraction Unit or Node Manager, when there is no requirement calling for aggregation of resources, e.g., as the case of a slicing that

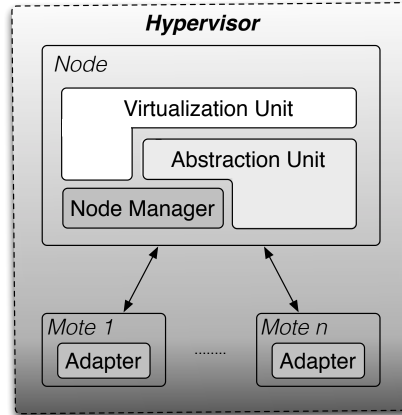
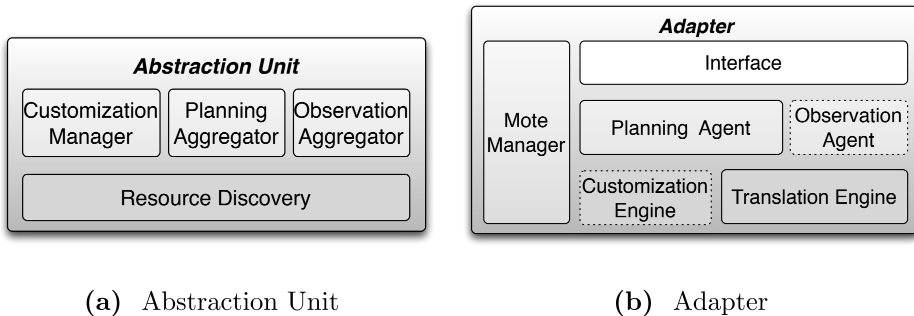


Figure 1.3: *SAaaS Hypervisor: architecture*



(a) Abstraction Unit **(b)** Adapter

Figure 1.4: *SAaaS Hypervisor: modules*

degenerates to mere mapping.

Below the Virtualization Unit lies an *Abstraction Unit*. As may be noticed comparing Figure 1.4a against Figure 1.4b, the Abstraction Unit replicates planning and observation facilities, in turn modelled after those featured in the Adapter, this time operating on a node-wide scale, thus combining the pool of resources of a WSN as a whole. In particular, the *Observation Aggregator* exports all node resources, while the *Planning Aggregator* helps in managing this set, by sending commands' combinations and tracking return codes, reacting to (partial) failures by timely triggering of adjustments. The *Resource*

Discovery module offers an interface to motes, actively gathering descriptors of underlying resources, then forwarding results to the Aggregator modules. A *Customization Manager* acts as orchestrator for customization engines deployed on motes.

Virtualization and Abstraction Units work over a *Node Manager*, which acts only at node level and is in charge of operations of sensing resources and of mandating policies, by cooperating with the Mote Manager, in turn inside the Adapter, replicating functionalities at SN mote level. These functionalities and roles of both modules collapse into the Mote Manager for standalone device. The Virtualization Unit is L-shaped in Figure 1.3 since it may work over an Agent-hosting Adapter, with no other interposed components, at least in selected cases, e.g., as in dealing with degenerate WSNs, i.e., a set of a single SN mote.

The lowest component of an Hypervisor is the *Adapter*, which is shown in Figure 1.4b, and plays three distinct roles: first exposing a customer-friendly and standards-compliant *Interface* toward on-board resources. It is in charge also of requests, retrievals, and eventually pre-processing stages for measurements, by means of the *Observation Agent*. The *Planning Agent* (PA) sends requests encoding actions (*tasks*) for the device. The two aforementioned agents rely on the availability of a platform-specific *Translation Engine*, in charge of the conversion of high-level directives to native commands.

The Hypervisor, through an Adapter, may also process requests to (re)configure a resource, leveraging an (optional) *Customization Engine*, i.e., an interpreter able to execute code needed to tailor sensing activities to requirements, per customer demand. An autonomic approach may be adopted by delegating specific management tasks of the Adapter to a *Mote Manager*, running mote-side, including operations such as power-driven self-optimization to be carried out with help from the Node Manager.

Furthermore, the main problem this layer-spanning module has to tackle lies in providing adequate mechanisms for a customer to be able to establish an out-of-band channel toward the system, i.e., one that is not Agent- or Interface-mediated. This is meant for direct interaction with either the resources or low-level modules, identifying it as a basic module of the overall stack. In the former case, e.g., for Agent-agnostic collection of observations, in the latter such as, e.g., the Customization Engine.

Planning Agent

The Planning Agent is one of the main modules of the Sensing IaaS system, implementing very basic facilities for management of sensing (and other) resources, including functionality reservation, parameter tuning, and observation scheduling. These allow to manage operating parameters for sensors, such as, e.g., sampling frequency, duty cycle, etc., dispatching as well platform-specific commands for the reservation of relevant processing and storage quotas.

The PA runs side-by-side with the Observation Agent, and complements its features. Unlike the latter, which is engaged in providing the upper layers with XML-encoded samples (*observations*), measured while driving sensing resources, including feedback on relevant system-wide metrics about the status of node-side processing and storage resources, the former is devoted mainly to the tuning of sampling parameters according to user-defined preferences, to be still interfaced with through standard-compliant and extensible encoding of requests for *tasks*, as well as corresponding responses. Beyond tuning, tasks for the scheduling of observations may be consumed by the PA: either following a predefined schedule, if a specific event occurs, or simply as a request from a client. The main goal here consists in exposing all underlying knobs to have those available to be operated on transparently by customers. Even though providing standardized

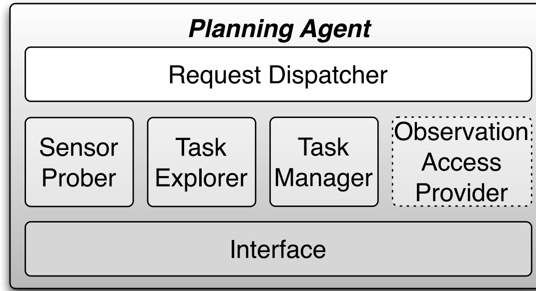


Figure 1.5: *SAaaS Planning Agent: architecture*

and useful mechanisms, an Observation Agent is not mandatory in order to let customers tap into sensing infrastructure. Indeed a PA may be enough to handle virtual or physical resources, once a bidirectional communication channel is established between the mote, or sensor-hosting mobile, and the client. Such facility then would be enough to let customers get and store observations, working synchronously over the channel if needed.

To meet the above mentioned requirements, an architecture, which comprises the six modules depicted in Figure 1.5, has been conceived: Request Dispatcher, Sensors Prober, Task Explorer, Task Manager, Observation Access Provider and Interface.

The *Request Dispatcher* manages a request, demultiplexing it accordingly to the modules below. The *Interface* needs to interact with Adapter services, that are: Customization Engine, Translation Engine and Node Manager.

The *Sensor Prober* enumerates all sensors and actuators within a platform, however complex and rich, by means of platform-specific low-level system probing. These sensors get identified therefore according to: type, observation facilities, sampling specifications, nominal features and commercial information (brand, model, etc.).

The *Task Explorer* enumerates available tasks, by probing available sensors and the underlying platform. Tasks related to the tuning

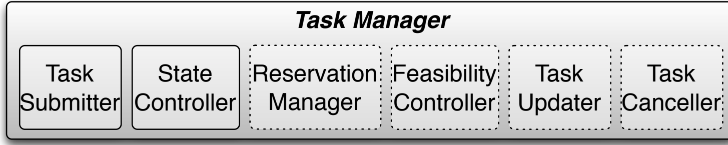


Figure 1.6: *SAaaS Task Manager: architecture*

of parameters for sensors may be logically separated according to sensor technology and type, thus it is possible to, e.g., plan the access to temperature measurements from a thermometer, as soon as a threshold has been exceeded, modify the focal length and relative position of a camera, or simply schedule periodic retrieval of observations with a predefined frequency, etc. Moreover, to evaluate whether a task is feasible or else, among the ones available for selection, it is required to query the relevant subsystem (sensing, computing, storage) to provide (runtime) confirmation of availability for servicing, or reservation thereof, otherwise returning a denial response. It is then a prerogative of the party querying the resource to decide what to do after an assessment of feasibility for the considered task.

The *Task Manager* manages the lifecycle of tasks, starting from the assessment of feasibility, through reservation and submission stages, following then up, and acting upon, the progress of a running task. Due to the number of operations involved, the Task Manager is organized into the six modules reported in Figure 1.6. Only two of them are mandatory.

Mandatory functionalities are provided by *Task Submitter* and *State Controller* modules. Respectively, their roles are enabling users to set mandatory parameters for a task before submission to a resource, and submit it when ready, following up processing stages of the task, alerting whichever agent querying about availability for task execution after submission, about its status, i.e., busy, until completion. Conversely, optional modules are: Reservation Manager, Feasibility

Controller, Task Updater, Task Canceller. These provide further facilities for management of running tasks to process, or ones yet to be scheduled.

If needed, a task may be reserved by a user for a period of time, during which exclusive access is granted to the provided sensing resource and no other user is allowed to submit or reserve it. The task will be performed once the user confirms, in order to the processing stages begin. It is up to the *Reservation Manager* both the reservation of tasks, and the corresponding confirmation. The *Feasibility Controller* checks if a task is serviceable, as detailed above. Such status depends on the availability of resources(s) which are essential for servicing the task, e.g., when not still allocated according to a previous request.

The *Task Updater* refreshes configuration parameters for a task, whenever there are modifications to be pushed after tasks have already entered processing stages. The *Task Canceller* allows users to stop and remove a task, which is already submitted or under reservation anyway.

Once a task has been serviced, resulting observations get stored. Any such measurement would then be accessible only through an Observation Agent. In terms of observations, the only duty left to the PA consists in the ability of the *Observation Access Provider* to provide endpoints as a way to access observations. As it depends on an Observation Agent, that makes it an optional component, which is required only when the latter is available in the Abstraction Unit.

1.4 Basic device interactions

Once described the device-centric stack building block architecture, in this section the focus is on the interactions among them and the end users. An end user-system interaction is composed of three sequential

macro-steps:

Sensors & Tasks Acquisition, *Sensor Use / Interaction*, and *Observation Access*.

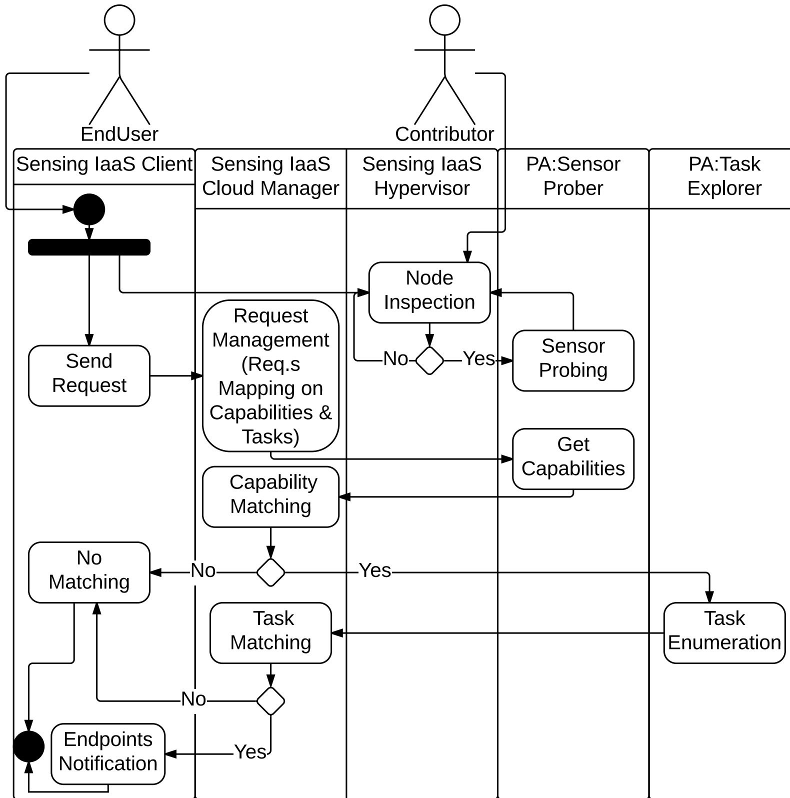


Figure 1.7: *SAaaS: resource acquisition AD*

In the Sensors & Tasks Acquisition macro-step, the tasks available over the full set of (on-board) sensing resources are counted and acquired. The activity diagram (AD) of Figure 1.7 describes the required activities for acquisition: the end user first sends a request with requirements and preferences on the needed resources and tasks to the Sensing IaaS server through the corresponding local client. On the contributor side, the Sensing IaaS stack has to probe the contributing nodes to find any exploitable resource available. The core resource (information) is then obtained in two steps: a capability search is first

performed and then, a discovery of the tasks provided by the resources is run on the device with matching capabilities. Once the matching tasks are identified, a list with corresponding endpoints (void otherwise) is sent to the end user.

When a sensing resource is acquired by the user, she can manage and configure it through the Sensor Use / Interaction step, which is split into Submission (Figure 1.8) and Management (Figure 1.9) substeps.

The former details how to submit, process, execute a specific task on the sensor. The latter describes the task management operations a user can perform during its processing.

Specifically, in the Submission the user selects a task among the available alternatives (if present) and configures it. Three options are available for submitting a task:

direct submission, *submission by reservation* (if deferred in time), or *feasibility checking*.

Through the Management a user that submitted a task can query or manage options related to its submission.

Three kinds of requests are available: *status checking*, *updating* and *cancelling*, mainly handled by the Task Manager modules.

Finally, through the Observation Access step a user can access to the sensed data results, measurements or observations. The diagram shown in Figure 1.10 details the operations required by this step that, as discussed in Section 1.3.2, usually involves the Observation Agent. This is triggered by the user through a “Describe Result Access” request, implying a query to both Observation Agent and Planning Agent.

1.5 Proof of concept

In this section the implementation of a prototype for some core functionalities of the sensing IaaS stack as discussed in Sections 1.3.2 and 1.4, in order to demonstrate the feasibility of the device-centric approach.

A first prototype implementation of the basic functionalities of the stack has been developed, targeting mobiles equipped with the Android OS, leveraging the NDK, a set of native libraries and APIs for developers.

As a reference standard, the implementation starts from the Sensor Planning Service (SPS) [17] of the Sensor Web Enablement (SWE) suite.

The implementation has been then tailored to a case study on a surveillance application through smartphones.

1.5.1 Case Study

The main idea for Sensing IaaS stack case study is to monitor a given area of interest using smartphone cameras, when available and able to capture frames on the area of interest.

To this purpose the main functionalities on which a user is interested in are those allowing to manipulate the camera, and especially to zoom in/out the camera on a specific point of interest. This way a mock Sensing Cloud provider have been implemented on just one contributing node, an Android 4.0 smartphone with a 1Ghz dual-core processor, 1GB RAM, 4GB intern memory, a 5Mpx VGA camera and a 2150 mAh battery. The Enforcer and Hypervisor modules of the stack have been deployed into the smartphone, while the end user interacts with the Sensing Cloud through a client deployed on an x86-backed running instance of the Android platform.

Following the phases and the algorithms described in Section 1.4,

in the following the end user-contributor/device interactions are highlighted in the case study at hand.

At the beginning, the user needs to select the kind of sensor according to preference, e.g., a camera, defining the area of interest and any operations and tasks to be performed (e.g., zoom-in/out).

The corresponding request is then forwarded to the Sensing IaaS that performs the discovery for the devices, sensors and tasks matching the user requirements.

The resource acquisition process is triggered by the end user that sends a request specifying the needed sensing resource (a camera) and a list of parameters and tasks (zoom in/out) it has to provide.

The Sensing IaaS server thus performs a resource discovery with such requirements among its contributing nodes following the workflow of Figure 1.7.

This way, a device matching the requirements is found in the testbed, i.e. the Android 4.0 smartphone initially enrolled by the Sensing IaaS provider, and the related message is delivered to the client (an Android-based emulated device).

On the other hand, if the system is not able to find any device matching the requirements, the system sends back a negative feedback to the end user client.

Assuming a successful discovery of the camera, it gets acquired by the end user to which a list of all the available task for the camera is notified.

This way, once the smartphone camera is acquired, a direct connection between the end user client and the device is established. This is exploited by the user to submit and manage task requests to the camera. In the former case the algorithm of Figure 1.8 is followed.

It is assumed the user wants to configure the camera parameters in the submission, specifically she wants to set 2 recording timers, by specifying the start and stop times and duration. The request is

submitted to the device in charge of enabling/disabling the targeted features of the camera.

Once the task is successfully submitted the user may query on its status through the management step, following the workflow of Figure 1.9 till the task is still running.

When the task is executed by the smartphone camera, any (remote, Cloud-side) user may just be interested in retrieving the collected data. One of the main benefit of the device-centric approach is that the sensed data may not just be collected, but also managed, aggregated and preprocessed at the source, limiting transfer and the overall impact on the network. This is possible by injecting some "intelligence" on the device, mainly into the Abstraction Unit and the Adapter Observation and Planning modules.

For instance, typical BigData treatment for such kind of raw data (e.g., video material) may entail leveraging real-time distributed streaming architectures such as Apache Storm [18] in order to map "topologies" (an alternative abstraction with respect to MapReduce "jobs") onto a set of nodes, including camera-hosting ones in the role of "spouts" (e.g., stream sources) plus one or more local "bolts" (intermediate processing and forwarding), as a way to build *pipelines* in place of centralized storage and multiple roundtrips.

A very simple use case may involve a generic camera-hosting node running a spout which provides an unprocessed, full-frame raw stream, and a couple of node-local bolts meant to extract different kinds of features (e.g., edges, shapes, etc.) by means of unrelated machine learning algorithms, to be fed to a variety of pipelines, according to the needed subset of computer vision approaches to be implemented.

Anyway, as described in Figure 1.10, the retrieval of the video recorded through the camera is performed by the user sending an Observations Access task request to the smartphone device, which returns the corresponding Observation Access Descriptor.

1.5.2 Testing

In this section an evaluation of the impact and performance of the sensing IaaS solution is put forward, from two possibly diverging perspectives: the contributor and user ones.

On one hand, a contributor is mainly interested in quantifying the impact of the stack on local resources, namely battery power, network bandwidth, computing and memory utilization.

On the other hand, the performance of a request processing is the main metric of interest for a user, in terms of response time and similar measurements on a complete acquisition-interaction-observation access interaction workflow.

In the tests all such metrics, i.e. CPU, memory, bandwidth utilization and battery depletion for contributors, were therefore measured, whereas from the user perspective of interest were the response times for acquisition, submission, management and observation access operations.

The tests were repeated 1000 times thus providing, in the following, the mean value μ and the 95% confidence interval offset γ for all such measurements.

Contribution Overhead

Param./	Battery Depletion		Network I/O		CPU		MEM
Stat.	**Energy** (J)	**Power** (W)	**Data** (KB)	**BWidth** (KB/s)	**Time** (ms)	**Usage** %	**Cons.** (MB)
μ	2.837	0.089	63.18/60.95	2.046/1.97	820.75	2.66%	1.54
γ	0.087	0.00013	0.092/0.088	0.0032/0.0030	10.43	0.051%	0.012

Table 1.1: *SAaaS results: contribution overhead*

The contributor-side overhead of the Sensing IaaS stack has been measured through the surveillance service running on the above described configuration. Inspired by literature [19, 20], the focus here was mainly on computing and memory utilization, network bandwidth

Parameter/ Statistics		**User** **Roundtrip time per request** (ms)	**Contributing Node** **Service time** (ms)	**SAaaS Framework/Network** **Overhead** (ms)
Acquisition	μ	383.35	294.13	89.17
	γ	18.67	14.32	3.33
Configuration	μ	381.57	296.97	84.53
	γ	17.84	15.35	2.88
Submission	μ	586.33	229.64	356.66
	γ	29.13	12.56	16.93

Table 1.2: *SAaaS results: individual operation performance*

and battery power for the smartphone contributing device. The values obtained by testing for all the considered metrics are summarized in Table 1.1 and described in the following.

Starting with device battery power, in order to evaluate the impact of the stack and the surveillance app on the smartphone, the PowerTutor tool has been used, a power monitor for Android-based mobile platforms [21]. The first column of Table 1.1 reports the corresponding values obtained by the power tests, specifically related to the smart surveillance app interactions against sensing IaaS, on the contributing node. These are quantifiable in 2.837 J in terms of energy or 89 mW expressed in power unit. The impact on the smartphone power, assuming a battery of 2150 mAh, can be estimated in about 1%.

With regard to the network bandwidth, the second column of Table 1.1 reports the statistics obtained by the testing on the surveillance app. In particular, the traffic incoming to/outgoing from the contributing node smartphone, and specifically the KB of data received/-transmitted and the corresponding bandwidth, is measured through the Eclipse ADT Plugin DDMS tool.

These values highlight that a very low and balanced traffic in the two directions is generated ($\sim$ 60 KB, bandwidth $\sim$ 2KB/s), thus sustainable in a 3G or higher technology smartphone. This parameter is of strategic importance to also demonstrate the benefit of a device-centric approach. Indeed, a very low traffic is experienced since data are stored locally instead of being immediately transferred once ac-

quired. The access to the data may be performed only once required, also selecting chunks of interest or even preprocessing, filtering and/or aggregating them. Even more important, the access to data for the end user is direct and not mediated by any server/service, avoiding further steps and waste of bandwidth and time.

The CPU impact is quantified in the third column of Table 1.1.

In particular the time the surveillance app and the sensing IaaS stack spend in the CPU is measured.

The Android debug class has been used to this purpose, adding specific log instructions in the app and the stack code, then evaluating the trace file through Traceview.

The results thus obtained allow to quantify the impact of the surveillance app in 2.66% on the overall CPU utilization.

The memory impact of the surveillance app over the Sensing IaaS stack deployed on the contributing smartphone has been quantified in the last column of Table 1.1.

As above measurements have been performed by employing the DDMS tool of the Eclipse ADT Plugin, and specifically invoking the System Information module.

The memory used by the app is in the average of 1.54 MB, about the 0.15% of memory utilization out of the 1GB available in the smartphone.

Performance

From a provider/end user viewpoint it is important to evaluate the temporal behaviour of the system to mainly obtain the app time to response or similar performance indices.

This way, another testing cycle has been prepared, this time focusing on the Sensing IaaS stack overhead in the contributing node and on the end to end surveillance app performance, respectively.

In particular the response times of the acquisition, configuration

and submission requests have been measured.

The measurements have been obtained through timestamps generated by the app and stack client-server, which source codes have been enriched with proper debug instructions placed at the beginning and at the end of each interaction or request processing. This way the duration of each operation performed by the overall system may be evaluated.

The results of this testing phase are reported in Table 1.2, showing that a user request is always processed in less than 1 s, and specifically in a range between 586.33 ms for submission to 381.57 ms for configuration, in the average. A value similar to the configuration one characterized the acquisition end user-smartphone interaction, 383.35 ms.

In order to evaluate the overhead of the stack, the time for processing a request have been measured on the contributing node, the smartphone, thus obtaining 229.64 ms for the submission processing, 296.97 ms and 294.13 ms for the acquisition and the configuration ones, respectively.

This way the overall overhead on a request may be evaluated, taking into account both the network and the stack contributions. The corresponding values are shown in the third column of Table 1.2. These highlight a high overhead of submission operations, while in the other kinds of operations the overhead is quite similar proportionally. It is due to the fact that a submission operation is a more complex operation than the others considered, since it often requires to perform a *GetFeasibility* and a *Reservation* operation before the *Submission* one as shown in the algorithm of Figure 1.8.

1.6 Related work

With regard to current research sensor boards and other constrained (nodal) environments are typically not considered in terms of their computation and storage capabilities albeit tiny, thus while an already sizeable literature output is available on BigData, it mostly focuses on the data treatment process in a dedicated cluster, i.e., what happens after data collection from external sources.

Even when the collection stages are to be considered into the investigation, the focus is usually on the minimization of network traffic, thus taking the form of communication protocol optimizations, or at most employing mechanisms for network throughput maximization by combining received packets together for transmission instead of just relaying those around. Examples are erasure coding [22], random linear coding [23], and network coding [24]. Erasure codes, like [10], may decrease network traffic at a small cost in processing power. Network coding, theoretically, may achieve optimal network throughput [11], while random linear coding - as a kind of rate-less erasure coding - may achieve this with no control overhead. It's interesting to note that in this case there's already the notion of non negligible computing power and storage space, to be put at the service of such techniques, thus trading off local capabilities in exchange for more efficient network utilization.

Following the service oriented computing trend and its generalization to the everything as a service (XaaS) approach [8], in literature several works adopted and applied this paradigm in distributed sensing and Cloud infrastructures to deal with data management issues. Most of them just consider the Cloud as an extended application domain from which data are retrieved and pushed according to a data-centric approach, providing services for their management. Typical examples in such direction are provided by [12], [13], [9] and [25], the latter

of which are mainly focused on smart city contexts, also supporting data (BigData) management through data-centric approaches for the domain at hand.

Similarly, some works explored possible intersections between IoT and Cloud, such as [26], mainly at high, semantic level, mapping physical IoT things into virtual environments in the Cloud. At infrastructure level, the BETaaS project [27] proposes a platform for the execution of M2M applications on top of services deployed in a Cloud of gateways.

In heterogeneous sensing environments, interoperability and communication issues call for specific software abstraction layers [28], enabling the dynamic reconfiguration of sensor nodes [29].

Abstraction and virtualisation are the solution proposed in [30] to implement seamless scalability and interoperability among sensing resources and things.

Some solutions aim at building up networks of mobiles, such as [31], performing measurements in SWE-compliant format also resorting on the mobile processing and storage resources.

A smartphone infrastructure is built up in [32] to opportunistically monitor physical actions performed by their owners, such as walking, jumping, running.

The key to understand the difference between the aforementioned literature and the proposal put forward here thus lies in intentionally lumping collection and treatment stages together. To be more precise it means getting rid of a full-blown collection step, as whichever data has to be gathered, it has to already be the output of local (or at least proximal) processing. This approach embodies the idea that, as long as some kind of trade-off is achievable or even already in place, computing and storage capabilities should be leveraged to a greater extent.

Still efforts should also be about trying to optimize against the

overall problem and not just limiting the engineering effort toward networking issues alone. As data treatment logic gets loaded and executed at the source, it has to be more and more similar to BigData techniques (e.g., MapReduce) typically running on a cluster.

This is also dynamic in nature, considering jobs are to be dispatched to sensing nodes as they are generated, i.e., at runtime. This is also not comparable to typical preprocessing duties for embedded nodes, such as simple statistics over a certain time period or sequence, e.g., averaging, which is another standard way to cope with scalability for transmission purposes.

A specific solution implementing the device-driven approach is strongly required to attack the problem from different perspectives, and in particular from the low-level, infrastructure one, as proposed here.

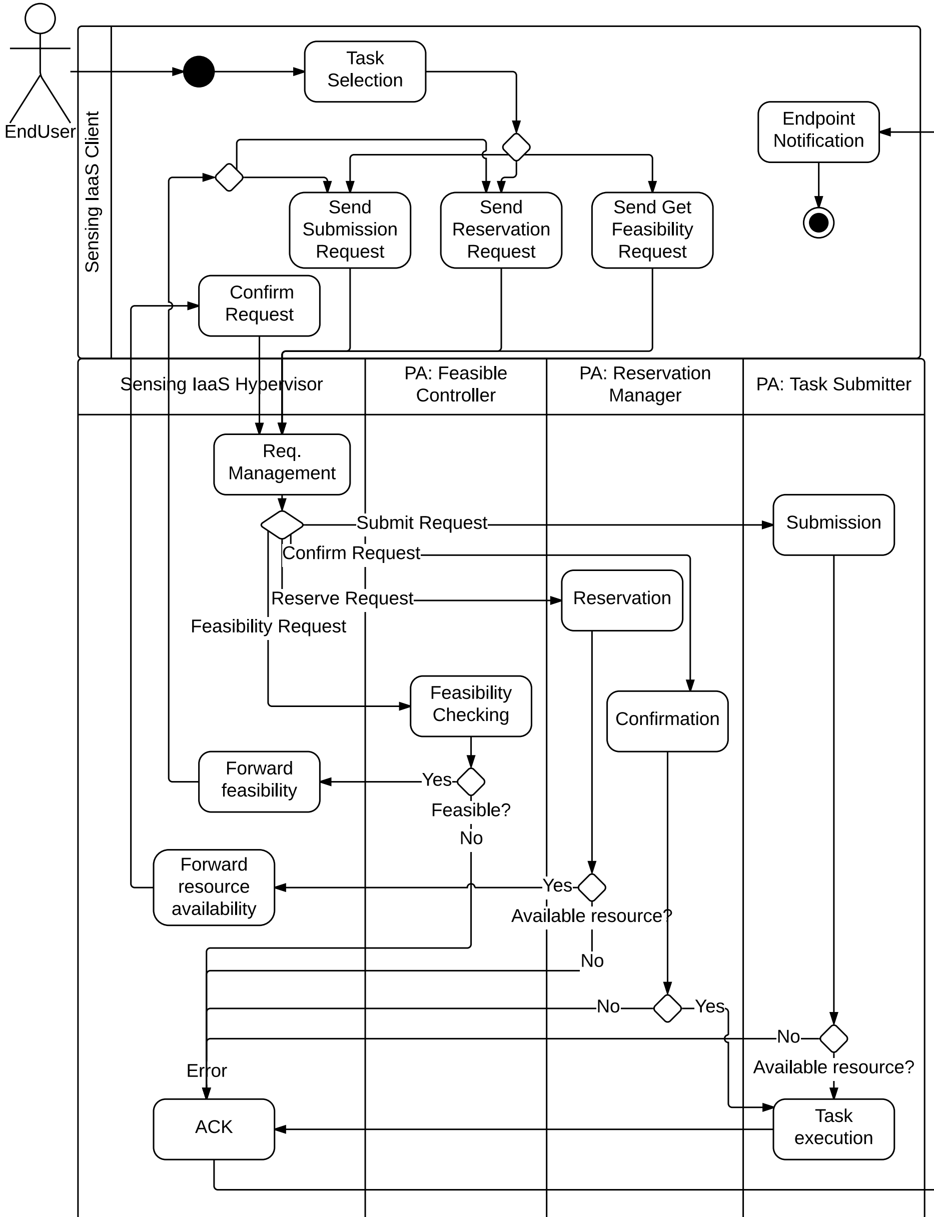


Figure 1.8: *SAaaS: request submission AD*

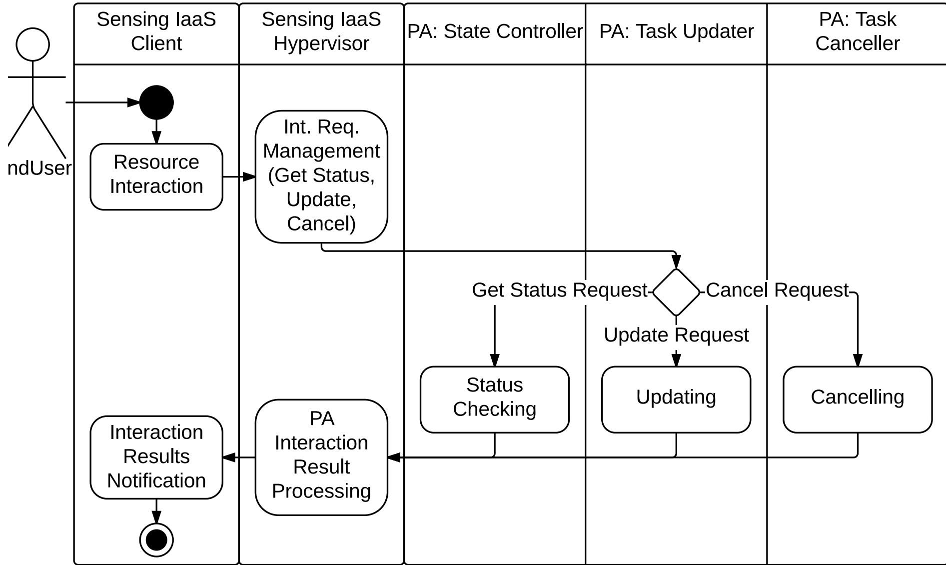


Figure 1.9: *SAaaS: submission management AD*

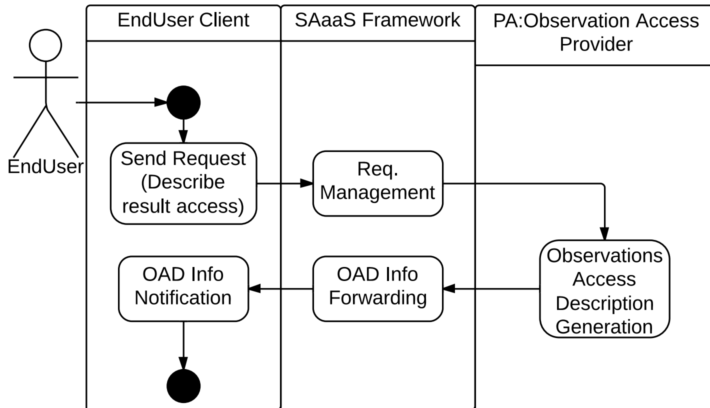


Figure 1.10: *SAaaS: observation access AD*

CHAPTER

TWO

STACK4THINGS: A FRAMEWORK FOR SAAAS

2.1 Introduction

Several solutions are already present in the literature for enabling the so-called Internet of Things, mainly focusing on lower (communication) layer and in particular on how to interconnect (among themselves and to the Internet) any network-enabled *thing* [33]. However, in order to realize the Sensing-and-Actuation-as-a-Service vision [16], other aspects have to be also taken into account such as solutions for creating and managing a dynamic infrastructure of sensing and actuation resources. In fact, in order to effectively control devices, sensors, and things, several mechanisms are strongly needed, e.g., management, organization, and coordination. Then, a middleware devoted to management of both sensor- and actuator-hosting resources may help in the establishment of higher-level services.

In this direction, the integration between IoT and Cloud is one of the most effective solutions even if up to now efforts revolve around managing heterogeneous devices by resorting to legacy protocols and

vertical solutions out of necessity, and integrating the whole ecosystem by means of ad-hoc approaches [34]. According to this vision, the Cloud may play a role both as a paradigm, and as one or more ready-made solutions for a (virtual) infrastructure manager (VIM), to be extended to IoT infrastructure. In particular, the proposal is to extend a well known framework for the management of Cloud computing resources, OpenStack [35], to manage sensing and actuation ones, by presenting Stack4Things[1] an OpenStack-based framework implementing the Sensing-and-Actuation-as-a-Service paradigm. Thanks to such a framework, it is possible to manage in an easily way fleets of sensor- and actuator-hosting boards regardless of their geographical position or their networking configuration.

Preliminary details of the Stack4Things [2] architecture have been presented in [37]. In this chapter, starting from a detailed requirement analysis the whole Stack4Things architecture is described by focusing on both Cloud and board components. In design stages a bottom-up approach has been followed, consisting of a mixture of relevant technologies, frameworks, and protocols. In addition to the already cited OpenStack, WebSocket technology [38] is taken advantage of, basing the communication framework on the Web Application Messaging Protocol (WAMP) [38].

2.2 Sensing-and-Actuation-as-a-Service

In this section, the scenario under investigation is first described by taking into consideration the main actors and entities involved. Then, requirements are reported upon which focus is given during the design of the Stack4Things framework, both from the functional and non-

[1] All the corresponding software is being developed as Open Source, making it freely available through the Web [36]

[2] From now on, Stack4Things is sometimes abbreviated as s4t.

functional points of view.

Figure 2.1 represents the scenario under consideration.

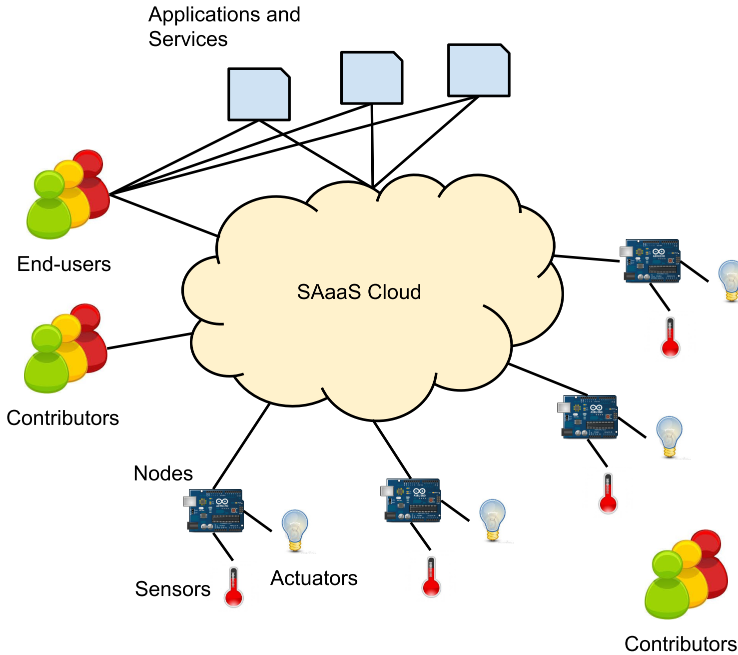


Figure 2.1: *Stack4Things: reference scenario*

In this scenario, Cloud computing facilities, implementing a service-oriented approach in the provisioning and management of sensing and actuation resources, are exploited to create a SAaaS Cloud. In fact, in the SAaaS vision, sensing and actuation devices should be handled along the same lines as computing and storage abstractions in traditional Clouds, i.e., virtualized and multiplexed over (possibly scarce) hardware resources. Thus, sensing and actuation devices have to be part of the Cloud infrastructure and have to be managed by following the consolidated Cloud approach, i.e., through a set of APIs ensuring remote control of software and hardware resources despite their geographical position. In other words, the idea is quite appealing and challenging since in such scenario an user could ask for handles

on (physical world) items to be manipulated through the user interfaces of the Cloud framework. Services related to the (sensing and actuation) infrastructure provisioning should be provided on-demand in an elastic and QoS-guaranteed way. In this way, on top of such services, other application level services can be easily implemented and provided to final users.

The main actors in the scenario are *contributors* and *end users*. Contributors provide sensing and actuation resources building up the SAaaS infrastructure. Examples of contributors are sensor networks owners, device owners, and people offering their PDAs as a source of data in a crowdsourcing model [39]. End users control and manage the resources provided by contributors. In particular, end-users may behave as infrastructure administrators and/or service providers, managing the SAaaS infrastructure and implementing applications and services on top of it. It is assumed that the sensing and actuation resources are provided to the infrastructure via a number of hardware-constrained units, which are henceforth referred to as *nodes*. Nodes host sensing and actuation resources and act as mediators in relation to the Cloud infrastructure. They need to have connectivity to the Internet in order for this approach to be applicable.

2.3 Background

The convergence of Cloud and IoT, and in particular the solutions to scale up IoT applications and to support real-time analytics, have been thoroughly investigated during the last years. A significant attempt in this direction, from a paradigmatic viewpoint, is *fog computing* [40] where both IoT and Cloud computing technologies are merged to provide new location-aware, reduced latency and improved QoS pervasive and ubiquitous services. Furthermore, based on this idea, several academic prototypes [41, 42, 43, 44] and commercial offerings such as

xively [45], ThingWorx [46] or SmartThings Open Cloud [47] are already available.

Some works focus on the implementation of an IoT/sensing Cloud [48, 49, 50], mainly dealing with sensing resource virtualization and management through a Cloud provisioning model. An interesting idea is to adopt some kind of hierarchical approach do improve network performance, adding nodes in between the device and the Cloud, as done through Cloudlets [51]. Another remarkable approach is the software defined one, successfully adopted in networking and data center management and thus applied in IoT-Cloud systems. Indeed, in [52], a first definition and a conceptual model of software defined things is provided, mainly implemented into the Cloud abstracting and encapsulating the underlying resource capabilities.

At higher level, in [53] a Cloud semantic overlay on top of physical sensing resources is proposed, specifying an IoT ontology able to provide semantic interoperability among heterogeneous devices and data formats. Based on semantic Web and CoAP technologies, the solution proposed in [54] mainly provide IoT service composition in a Cloud fashion.

All these efforts are manly focused on a data-centric perspective, mainly aiming at managing (IoT sensed) data by the Cloud. In [16] a different approach is adopted, where the goal is to provide actual sensing and actuation resources that could be handled by their (Cloud) users, as computing and storage resources in IaaS or DaaS Clouds, i.e. virtualized and multiplexed over (scarce) hardware resources. In other words, the proposed approach aims at adopting the service-oriented/Cloud paradigm in the management of sensing resources and things, according to a device-centric perspective, instead of considering the Cloud as just a complementary technology.

To this purpose, while designing the solution, efforts have been based upon Open Source technologies and standards. The latest *Ar-*

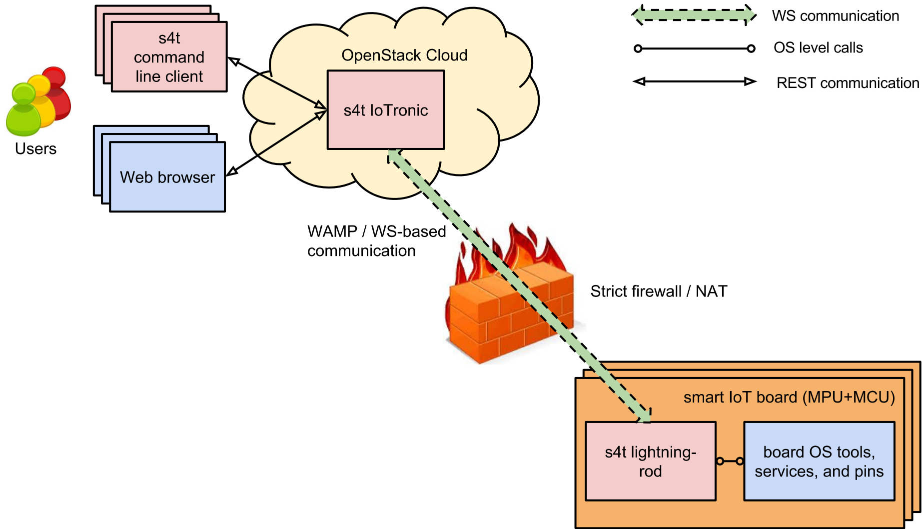


Figure 2.2: *Stack4Things: distributed system*

duino YUN [55]-like boards represent here a reference in terms of IoT nodes. Such a kind of devices is usually equipped with (low power) micro-controller (MCU) and micro-processor (MPU) units. They can interact with the physical world through a set of digital/analog I/O pins while connection to the Internet is assured by Ethernet and Wifi network interfaces. A Linux distribution (usually derived from the OpenWRT project) can run on the MPU. Recently, the use of *BaTHOS* [56] on the MCU side has become more mainstream, thus enabling the digital/analog I/O pins to be directly accessed from the MPU.

With respect to network connectivity, presence, and reachability, *WebSocket* [38] is the leading technology. WebSocket is a standard HTTP-based protocol providing a full-duplex TCP communication channel over a single HTTP-based persistent connection. WebSocket allows the HTTP server to send content to the browser without being solicited: messages can be passed back and forth while keeping the

connection open creating a two-way (bi-directional) ongoing conversation between a browser and the server. One of the main advantages of WebSocket is that communications are performed over TCP port number 80. This is of benefit for those environments which block non-Web Internet connections using a firewall. For this reason, several application-level protocols started to rely on this Web-based transport protocol for communication - see for example the use of eXtensible Messaging and Presence Protocol
(XMPP) over WebSocket - also in the IoT field.

Web Application Messaging Protocol (WAMP) [38] is a sub-protocol of WebSocket, specifying a communication semantic for messages sent over WebSocket. Differently from other application-level messaging protocols, e.g., XMPP, Advanced Message Queuing Protocol (AMQP), ZeroMQ, WAMP is natively based on WebSocket and provides both publish/subscribe (pub/sub) and (routed) remote procedure call (RPC) mechanisms. In WAMP, a router is responsible of brokering pub/sub messages and routing remote calls, together with results/errors.

As already mentioned, with respect to the virtual infrastructure manager, OpenStack [35] is the technology of reference. 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. The main goal in this chapter is to propose an extension of OpenStack for the management of sensing and actuation resources.

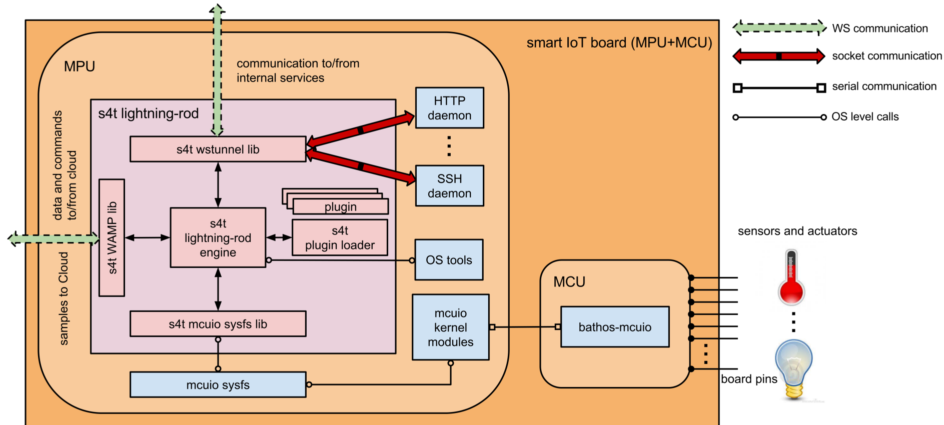


Figure 2.3: *Stack4Things: board-side architecture*

2.4 Stack4Things architecture

Figure 2.2 depicts at a high level the Stack4Things distributed system, focusing on communication between end users and sensor- and actuator-hosting nodes. It is assumed that each of such nodes is an Arduino YUN-like smart board. On the board side, the *Stack4Things lightning-rod* runs on the MPU and interacts with the OS tools and services of the board, and with sensing and actuation resources through I/O pins. It represents the point of contact with the Cloud infrastructure allowing the end users to manage the board resources even if they are behind a NAT or a strict firewall. This is ensured by a WAMP and WebSocket-based communication between the Stack4Things lightning-rod and its Cloud counterpart, namely the *Stack4Things IoTronic service*. The Stack4Things IoTronic service is implemented as an OpenStack service providing end users with the possibility to manage one or more smart boards, remotely. This can happen both via a command-line based client, namely *Stack4Things command line client*, and a Web browser though a set of REST APIs provided by the Stack4Things IoTronic service.

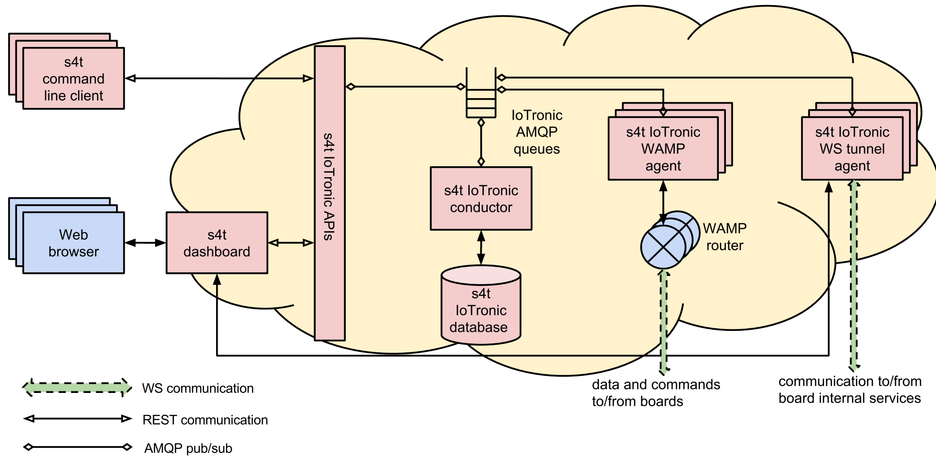


Figure 2.4: *Stack4Things: Cloud-side architecture*

2.4.1 Board-side

Figure 2.3 shows the Stack4Things architecture with more focus on the board side. It is assumed that *BaTHOS* runs on the board MCU while a Linux OpenWRT-like distribution runs on the MPU. BaTHOS is equipped with a set of extensions (from now on indicated as MCUIO extensions) that expose the board digital/analog I/O pins to the Linux kernel. The communication is carried out over a serial bus. The Linux kernel running on the MPU is compiled with built-in host-side *MCUIO modules*. In particular, functionalities provided by the MCUIO kernel modules include enumeration of the pins and exporting corresponding handlers for I/O in the form of i-nodes of the Linux sysfs virtual filesystem. Upwards the sysfs abstraction, which is compliant with common assumptions on UNIX-like filesystems, there is the need to mediate access by means of a set of MCUIO-inspired libraries, namely *Stack4Things MCUIO sysfs libraries*. Such libraries represent the interface with respect to the MCUIO sysfs filesystem dealing with read and write requests in terms of concurrency. This is done at the right level of semantic abstraction, i.e., locking and releasing resources ac-

cording to bookings and in a way that is dependent upon requirements deriving from the typical behavior of general purpose I/O pins and other requirements that are specific to the sensing and actuating resources.

The *Stack4Things lightning-rod engine* represents the core of the board-side software architecture. The engine interacts with the Cloud by connecting to a specific WAMP router (see also Figure 2.4) through a WebSocket full-duplex channel, sending and receiving data to/from the Cloud and executing commands provided by the users via the Cloud. Such commands can be related to the communication with the board digital/analog I/O pins and thus with the connected sensing and actuation resources (through the Stack4Things MCUIO sysfs library) and to the interactions with OS tools and/or resources (e.g., filesystem, services and daemons, package manager). The communication with the Cloud is assured by a set of libraries implementing the client-side functionalities of the WAMP protocol (*Stack4Things WAMP libraries*). Moreover, a set of WebSocket libraries (*Stack4Things wstunnel libraries*) allows the engine to act as a WebSocket reverse tunneling server, connecting to a specific WebSocket server running in the Cloud. This allows internal services to be directly accessed by external users through the WebSocket tunnel whose incoming traffic is automatically forwarded to the internal daemon (e.g., SSH, HTTP, Telnet) under consideration. Outgoing traffic is redirected to the WebSocket tunnel and eventually reaches the end user that connects to the WebSocket server running in the Cloud in order to interact with the board service.

The Stack4Things lightning-rod engine also implements a plugin loader. Custom plugins can be injected from the Cloud and run on top of the plugin loader in order to implement specific user-defined commands, possibly including system-level interactions, such as, e.g., with a package manager and/or the init/runlevels subsystem.

In this sense future efforts may resume previous results [3] related to runtime customization for further enhancements to the architecture.

New REST resources are automatically created exposing the user-defined commands on the Cloud side. As soon as such resources are invoked the corresponding code is executed on top of the smart board.

2.4.2 Cloud-side - control and actuation

The Stack4Things Cloud-side architecture (see Figure 2.4) consists of an OpenStack service that has been called IoTronic. The main goals of IoTronic lie in extending the OpenStack architecture towards the management of sensing and actuation resources, i.e., to be an implementation of the SAaaS paradigm. IoTronic is characterized by the standard architecture of an OpenStack service. The *Stack4Things IoTronic conductor* represents the core of the service, managing the *Stack4Things IoTronic*
database that stores all the necessary information, e.g., board-unique identifiers, association with users and tenants, board properties and hardware/software characteristics as well as dispatching remote procedure calls among other components. The *Stack4Things IoTronic APIs* exposes a REST interface for the end users that may interact with the service both via a custom client (*Stack4Things IoTronic command line client*) and via a Web browser. In fact, the OpenStack Horizon dashboard has been enhanced with a *Stack4Things dashboard* exposing all the functionalities provided by the Stack4Things IoTronic service and other software components. In particular, the dashboard also deals with the access to board-internal services, redirecting the user to the *Stack4Things IoTronic WS tunnel agent*. This piece of software is a wrapper and a controller for the WebSocket server to which the boards connect through the use of Stack4Things wstunnel libraries.

Similarly, the *Stack4Things IoTronic WAMP agent* controls the

WAMP router and acts as a bridge between other components and the boards. It translates Advanced Message Queuing Protocol (AMQP) messages into WAMP messages and vice-versa. AMQP is an open standard application layer protocol for message-oriented middleware, a bus featuring message orientation, queueing, routing (including point-to-point and
publish-subscribe), reliability and security. Following the standard OpenStack philosophy all the communication among the IoTronic components is performed over the network via an AMQP queue. This allows the whole architecture to be as scalable as possible given that all the components can be deployed on different machines without affecting the service functionalities, as well as the fact that more than one *Stack4Things IoTronic WS tunnel agent* and more than one *Stack4Things IoTronic WAMP agent* can be instantiated, each of them dealing with a sub-set of the IoT devices. In this way, redundancy and high availability are also guaranteed. As already mentioned in Section 2.3, a prominent reason for choosing WAMP as the protocol for node-related interactions, apart from possibly leaner implementations and smoother porting, lies in WAMP being a WebSocket subprotocol and supporting two application messaging patterns, Publish & Subscribe and Remote Procedure Calls, the latter being not available in AMQP.

2.4.3 Cloud-side - sensing data collection

The OpenStack service that collects monitoring data and events from the infrastructure (mainly for billing and elasticity purposes) is Ceilometer. Building on top of it, in order to allow collection of metrics coming from the smart boards, in particular a *Stack4Things Ceilometer agent* is provided, to which smart boards that need to send metrics can connect. Such an agent translates the WAMP messages

received by the boards to AMQP messages in the form of OpenStack notifications. Such notifications are then translated by the Ceilometer framework in samples that are collected by the Ceilometer collector and then stored in a non-SQL database (usually MongoDB). Metrics and events can be accessed through the Ceilometer APIs. The Stack4Things dashboard and the Stack4Things command line client are also able to interact with such APIs in order to obtain/visualize real-time and historical data. The Stack4Things framework also provide complex event processor (CEP) functionalities through the *Stack4Things CEP engine*. This engine can be programmed in order to detect specific situations of interest that can then be signaled to the Stack4Things IoTronic conductor which, in turn, can send commands to the smart boards in order to react to the situation by actuating actions or changing their behavior.

A prototype of the architecture so far described has been implemented and source code is freely available through the Web [36].

2.5 Stack4Things REST API

Table 2.1 reports an extract of the Stack4Things IoTronic RESTful API with exploited HTTP methods, URLs, semantics, input parameters, and return types. We focus on API methods that relate to nodes, corresponding pins, services that can be accessed on the nodes, jobs that can be scheduled to send sensor readings to the Cloud, and injection of CEP statements with specific reactions.

API calls listed under the Nodes section provide a list of the nodes currently registered to the Cloud (NodeCollection JSON data type provided in the body of the response) and, if necessary, detailed information about each node (Node JSON data type). An example of NodeCollection JSON response is the following:

```
{
 "nodes": [
  {
```

```
   "description": "Sample node",
    "links": [
     {
      "href": "http://s4t.org/v0.1/nodes/eaaca217-
           e7d8-47b4-bb41-3f99f20eed89",
      "rel": "self"
     },
     {
      "href": "http://s4t.org/nodes/eaaca217-e7d8
           -47b4-bb41-3f99f20eed89",
      "rel": "bookmark"
     }
    ],
    "uuid": "eaaca217-e7d8-47b4-bb41-3f99f20eed89"
  }
 ]
}
```


while the following is an example of Node JSON response:


```
{
 "created_at": "2000-01-01T12:00:00",
 "description": "Sample node",
 "extra": {},
 "links": [
  {
   "href": "http://s4t.org/v0.1/nodes/eaaca217-
        e7d8-47b4-bb41-3f99f20eed89",
   "rel": "self"
  },
  {
   "href": "http://s4t.org/nodes/eaaca217-e7d8-47
        b4-bb41-3f99f20eed89",
   "rel": "bookmark"
  }
 ],
 "updated_at": "2000-01-01T12:00:00",
 "uuid": "eaaca217-e7d8-47b4-bb41-3f99f20eed89"
}
```


API calls listed under the Pins section provide the interface to access pins on nodes. In particular, it is possible to retrieve a node layout in terms of pins and their modes and it is possible to set/unset modes on a pin. Finally, it is possible to set/read a value from a pin. The RESTful interface hides the complexity

#	Method	URL	Semantics	Parameters	Return Type
	Nodes				
1	GET	/v0.1/nodes	Retrieve a list of nodes.		NodeCollection
2	GET	/v0.1/nodes/{node_uuid}	Retrieve information about the given node.	node_uuid (uuid)	Node
	Pins				
3	GET	/v0.1/nodes/{node_uuid}/pins	Retrieve a list of pins on a node.	node_uuid (uuid)	NodeLayout
4	GET	/v0.1/nodes/{node_uuid}/pins/{pin_name}/mode	Retrieve the mode of a pin on a node.	node_uuid (uuid) pin_name (unicode)	PinMode
5	POST	/v0.1/nodes/{node_uuid}/pins/{pin_name}/mode	Set the mode of a pin on a node.	node_uuid (uuid) pin_name (unicode) PinMode (json in body)	-
6	DELETE	/v0.1/nodes/{node_uuid}/pins/{pin_name}/mode	Clear mode setting for a pin on a node.	node_uuid (uuid) pin_name (unicode)	-
7	GET	/v0.1/nodes/{node_uuid}/pins/{pin_name}/value	Retrieve the value of a pin on a node.	node_uuid (uuid) pin_name (unicode)	PinValue
8	POST	/v0.1/nodes/{node_uuid}/pins/{pin_name}/value	Set the value of a pin on a node.	node_uuid (uuid) pin_name (unicode) PinValue (json in body)	-
Services					
9	GET	/v0.1/nodes/{node_uuid}/services	Retrieve a list of services on a node	node_uuid (uuid)	ServiceCollection
10	POST	/v0.1/nodes/{node_uuid}/services	Activate a new service on a node	node_uuid (uuid) Service (json in body)	-
11	DELETE	/v0.1/nodes/{node_uuid}/services	Delete a service on a node	node_uuid (uuid) Service (json in body)	-
Jobs					
13	GET	/v0.1/nodes/{node_uuid}/jobs	Retrieve a list of jobs on a node	node_uuid (uuid)	JobCollection
14	POST	/v0.1/nodes/{node_uuid}/jobs	Activate a new job on a node	node_uuid (uuid) Job (json in body)	-
15	DELETE	/v0.1/nodes/{node_uuid}/jobs	Delete a job on a node	node_uuid (uuid) Job (json in body)	-
CEP statements					
16	GET	/v0.1/cep	Retrieve a list of cep statements on the cloud	-	CepStatementWReactionCollection
17	POST	/v0.1/cep	Activate a new cep statements on the cloud	CepStatementWReaction	-
18	DELETE	/v0.1/cep	Delete a cep statements on the cloud	CepStatementWReaction	-

Table 2.1: *IoTronic REST API*

2.6 Use cases

In this section, we propose some specific use cases that have been implemented and tested in the Stack4Things middleware. In particular, we show how the architecture components interact among themselves to fulfill certain objectives. Each use case corresponds to a specific call in the Stack4Things REST APIs.

2.6.1 Use case: provide the list of nodes registered to the Cloud

This use case is the basic one, consisting in a listing of all the nodes currently registered to the Cloud. It corresponds to call #1 in Table 2.1. The listing is useful to retrieve the unique identifiers of the different nodes that can later be used to directly interact which each of them. As a use case prerequisite, we assume that one or more nodes are already registered to the Cloud. The following operations are then performed (see Figure 2.5).

1. The user asks for the list of the nodes registered to the Cloud through the s4t dashboard (or alternatively through the s4t command line client).

2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST (specifically, call #1). The call pushes a new message into an AMQP IoTronic queue.

3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it performs a query to the s4t IoTronic database retrieving the list of nodes registered to the Cloud.

4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.

5. The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.

6. The s4t dashboard provides the user with the list of boards registered to the Cloud.

In this use case, no interaction with any board is necessary given that all the information is stored on the s4t IoTronic database. However, if desired, the connectivity status of the boards that are currently registered to the Cloud can be retrieved from the WAMP router using the presence mechanisms that it provides natively. In particular, such an information can be collected on demand or periodically. In the first case, when the call for retrieving the list of nodes is issued, after querying the s4t IoTronic database, the s4t IoTronic conductor can contact the s4t IoTronic WAMP agents to which the boards are registered obtaining news about their connectivity status. In the second case, the s4t IoTronic WAMP agents can periodically store such an information on the s4t IoTronic database in a proactive way so that the s4t IoTronic conductor is able to find it when necessary. Of course, a tread-off between freshness of the information (first case) and performance of the call (second case) arises.

2.6.2 Use case: retrieve the current value of a pin on a specific board

This use case is slightly more complex than the first one as it requires an interaction with a specific board. It corresponds to call #7 in Table 2.1. As a use case prerequisite, we assume that one or more nodes are already registered to the Cloud and that the user knows both the unique identifier of the desired board (maybe retrieved by issuing call #1 in Table 2.1) and the name of the pin from which he/she wants to read the current value (maybe retrieved by issuing

call #3 in Table 2.1 to get the list of pins of a specific node). The following operations are then performed (see Figure 2.6).

1. The user asks for the current value of a pin on a specific board through the s4t dashboard (or alternatively through the s4t command line client).

2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST (specifically, call #7). The call pushes a new message into a specific AMQP IoTronic queue.

3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it performs a query to the s4t IoTronic database. In particular, it checks if the board is already registered to the Cloud and if the required pin actually exists. Finally, it queries for the s4t IoTronic WAMP agent to which the board is registered.

4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.

5. 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.

6. Through the s4t WAMP lib the s4t lightning-rod engine receives the message by the WAMP router.

7. The s4t lightning-rod engine uses the s4t mcuio sysfs lib to read the value of the specified pin and through the s4t WAMP lib replies to the s4t IoTronic WAMP agent by writing a message into a specific topic on the corresponding WAMP router.

8. The s4t IoTronic WAMP agent receives the message from the WAMP router and publishes a new message into a specific

AMQP IoTronic queue with the value that has been read from the pin on the specified board.

9. The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.

10. The s4t dashboard provides the user with the value that has been read from the pin on the specified board.

As already mentioned, scalability in this kind of use cases is assured by instantiating more than one s4t IoTronic WAMP agent so that each one of them can deal with a subset of the boards connected to the Cloud infrastructure.

2.6.3 Use case: create an SSH connection toward a node

This use case is a common one in classical Cloud scenarios, i.e., SSH access into a virtualized resource. In our case, we consider the creation of an SSH connection toward a node through the help of the Cloud management system. The use case corresponds to call #10 in Table 2.1 that the user has to issue specifying the standard SSH port, i.e., port number 22. Given the assumption that nodes are behind a firewall/NAT a more complex interaction flow is needed in order to fulfill the goal with respect to a standard Cloud scenario in which compute nodes hosting virtual machines and frontend nodes through which the connection is guaranteed are usually within the same local area network. In fact, in order for a connection to be created with a service hosted on a board, a board-initiated tunnel needs to be created to the Cloud. As a use case prerequisite, we assume the node of interest is already registered to the Cloud and the SSH daemon is

already listening on its standard port[3]. The following operations are then performed (see Figure 2.7):

1. The user asks for a connection to the SSH daemon running on a specific board through the s4t dashboard (or alternatively through the s4t command line client).
2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST. The call pushes a new message into a specific AMQP IoTronic queue.
3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it performs a query to the s4t IoTronic database. In particular, it checks if the board is already registered to the Cloud and queries for the s4t IoTronic WAMP agent to which the board is registered. Finally, it decides the s4t IoTronic WS tunnel agent to which the user can be redirected and randomly generates a free TCP port.
4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.
5. 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.
6. Through the s4t WAMP lib the s4t lightning-rod engine receives the message by the WAMP router.
7. The s4t lightning-rod engine opens a reverse WebSocket tunnel to the s4t IoTronic WS tunnel agent specified by the s4t IoTronic

[3]Note that start/stop/restart commands for standard services running on the boards registered to the Cloud has been implemented through the RPC functionalities provided by the WAMP mechanisms but we do note report the corresponding execution flows for a matter of space.

conductor also providing the TCP port through the s4t wstunnel lib. It also opens a TCP connection to the internal SSH daemon and pipes the socket to the tunnel.

8. The s4t IoTronic WS tunnel agent opens a TCP server on the specified port. Then, it publishes a new message into a specific AMQP IoTronic queue confirming that the operation has been correctly executed.

9. The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.

10. The s4t dashboard provides the user with the IP address and TCP port that he/she can use to connect to reach the SSH service on the board.

11. The user connects to the specified IP address and TCP port via an SSH client and the connection is tunneled to the board.

2.6.4 Use case: store readings from a sensor in the Cloud

This use case (see Figure 2.8) envisions the usage of the Cloud infrastructure not only for the management and control of the nodes but also as a storing platform for sensing data. In fact, the readings coming from a specific sensor attached to a certain board can be stored in the Ceilometer database and could be potentially used to take decisions and react to predefined situations (see the next use case). The use case corresponds to call #14 in Table 2.1. Other storage systems could be used different from the Ceilometer one. For example, we also implemented mechanisms for the storing of sensing data in external MongoDB databases and or Open Data-compliant CKAN-enabled storages such as the one provided by the FI-WARE [57] infrastructure.

As a use case prerequisite, we suppose a sensor is attached to a specific pin of a certain board. We also suppose the board is already registered to the Cloud.

1. The user asks for the readings from a sensor attached to a specific pin of a certain board to be stored in the Cloud with a sampling time of choice[4] through the s4t dashboard (or alternatively through the s4t command line client).

2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST. The call pushes a new message into a specific AMQP IoTronic queue.

3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it performs a query to the s4t IoTronic database. In particular, it checks if the board is already registered to the Cloud and queries for the s4t IoTronic WAMP agent to which the board is registered. Finally, it decides the s4t ceilometer agent to which the readings from the sensor should be sent.

4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.

5. The s4t IoTronic WAMP agent to which the board is registered pulls the message from the queue and pushes a new message into a specific topic on the corresponding WAMP router.

6. Through the s4t WAMP lib the s4t lightning-rod engine receives the message by the WAMP router.

[4]Other mechanisms have been implemented in which the sending of the sensor readings is triggered by a change or by the overpassing of a specified threshold but we do not report them here for a matter of space.

7. The s4t lightning-rod engine connects to the WAMP router of the specified s4t ceilometer agent through the s4t WAMP lib. Then, the s4t lightning-rod engine periodically reads from the specified pin through the s4t mcuio sysfs lib and pushes a message into a specific topic on the WAMP router.

8. The s4t ceilometer agent pulls each message from the WAMP router and pushes a corresponding message into the Ceilometer AMQP queue.

9. The Ceilometer collector collects the messages from the Ceilometer AMQP queue and stores the contained metrics into the Ceilometer non-SQL database. It also sends them to the s4t CEP engine via REST for further semi real-time analysis.

Besides periodic sampling of the readings of a sensor, the user is also allowed to program the system to send samples when specific situations are detected, e.g., a threshold is passed, a positive/negative change in the value occurs. Mixed data dispatching modes are also available, e.g., allowing users to program the system to send samples periodically and each time a threshold is passed.

2.6.5 Use case: inject a CEP rule and set a reaction

This use case (see Figure 2.9) shows how real-time analysis capabilities can be injected in the Cloud to react to specific situations of interest. In fact, the readings coming from all the sensors attached to the boards registered to the Cloud that have been configured to be stored in the Ceilometer database can be also redirected to the CEP engine that can be programmed to detect user-specified data patterns. The user can specify both the pattern of interest (in the form of a ESPER

statement) and the reaction that he/she desires the system to have as soon as a detection is performed. We designed the system so that reactions can be in the form of REST calls to the IoTronic interface so that, e.g., actuation commands can be send to the pins of specific boards, the configuration of the system can be changed injecting CEP rules, and so on. The use case corresponds to call #17 in Table 2.1. As a use case prerequisite, we suppose that a set of metrics have already be programmed so that they are sent to the Cloud.

1. The user asks a CEP rule to be injected in the system together with a corresponding reaction through the s4t dashboard (or alternatively through the s4t command line client).

2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST. The call pushes a new message into a specific AMQP IoTronic queue.

3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it stores in the s4t IoTronic database the reaction that it founds in the message so that it will be able to retrieve it if the s4t CEP engine signals that the situation of interest occurs.

4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.

5. The s4t CEP engine pulls the message from the queue and dynamically load the CEP rule so that it can be continuously checked.

6. If a CEP rule is triggered the s4t CEP engine pushes a new message into a specific AMQP IoTronic queue.

7. The s4t IoTronic conductor pulls the message from the queue and it queries the database for the reaction that has been associated to that rule.

8. The s4t IoTronic conductor issues the call specified in the reaction to the s4t IoTronic API actuating the reaction.

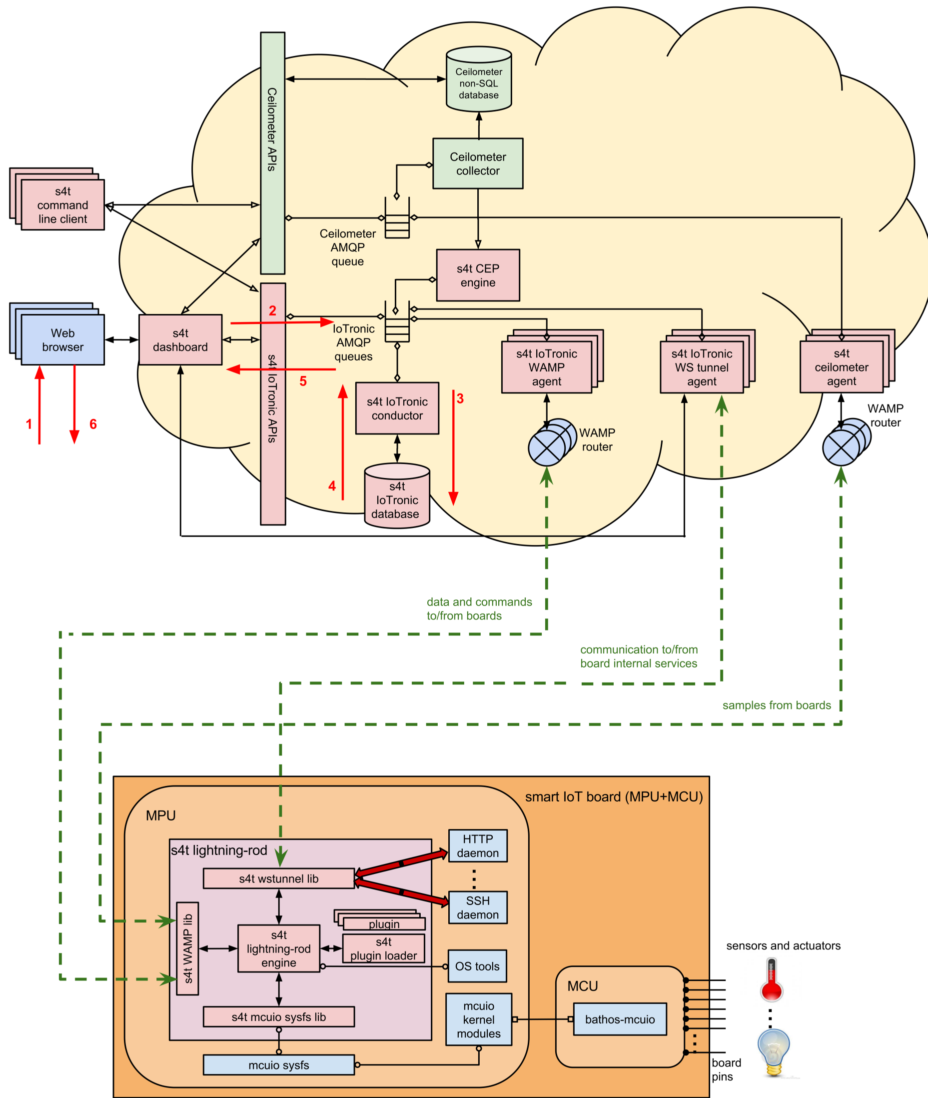


Figure 2.5: *S4T: listing of registered nodes*

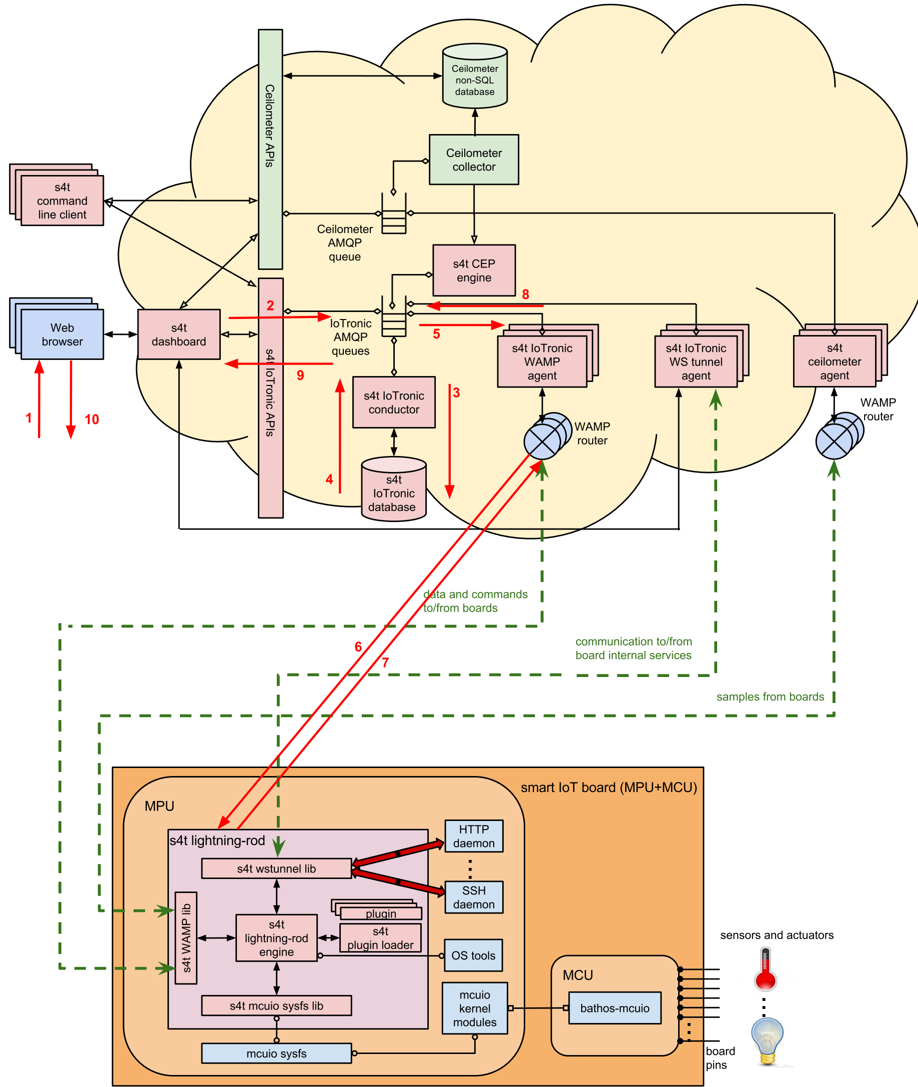


__Figure 2.6:__ S4T: retrieving current value of a pin

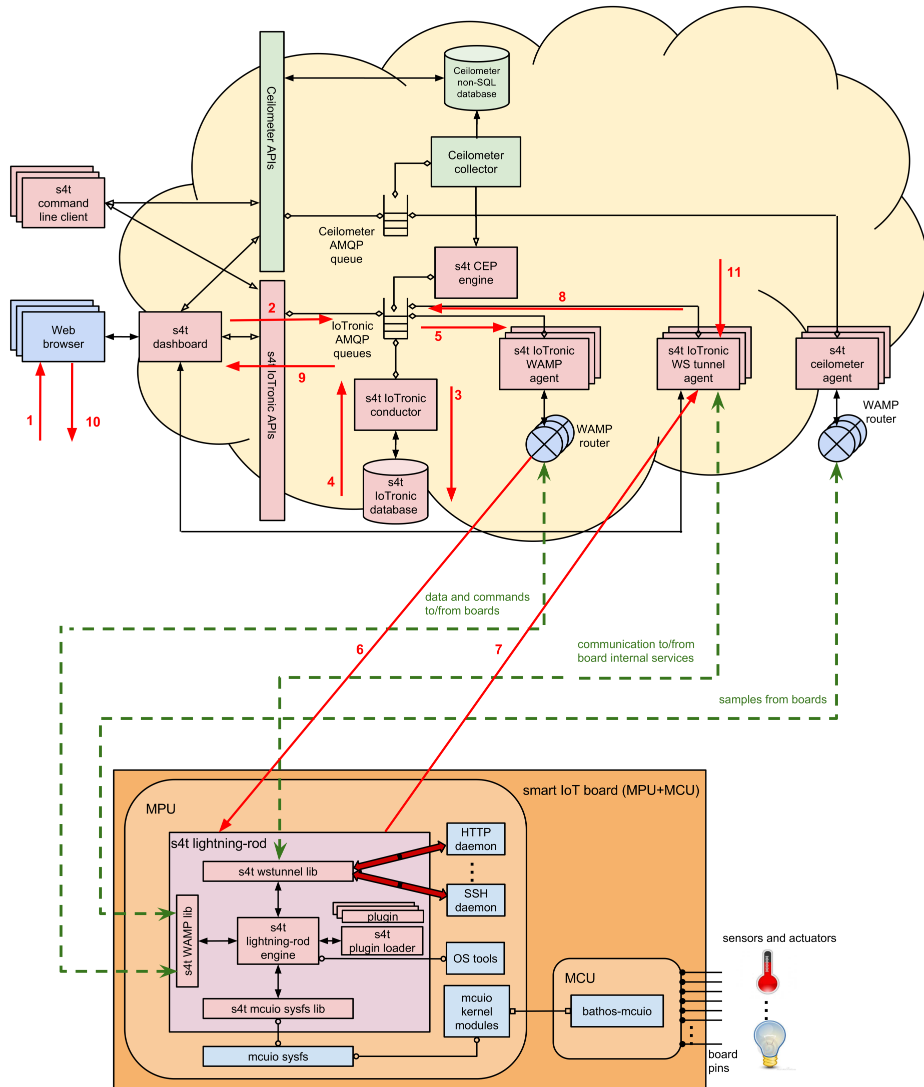


Figure 2.7: S4T: creation of an SSH connection

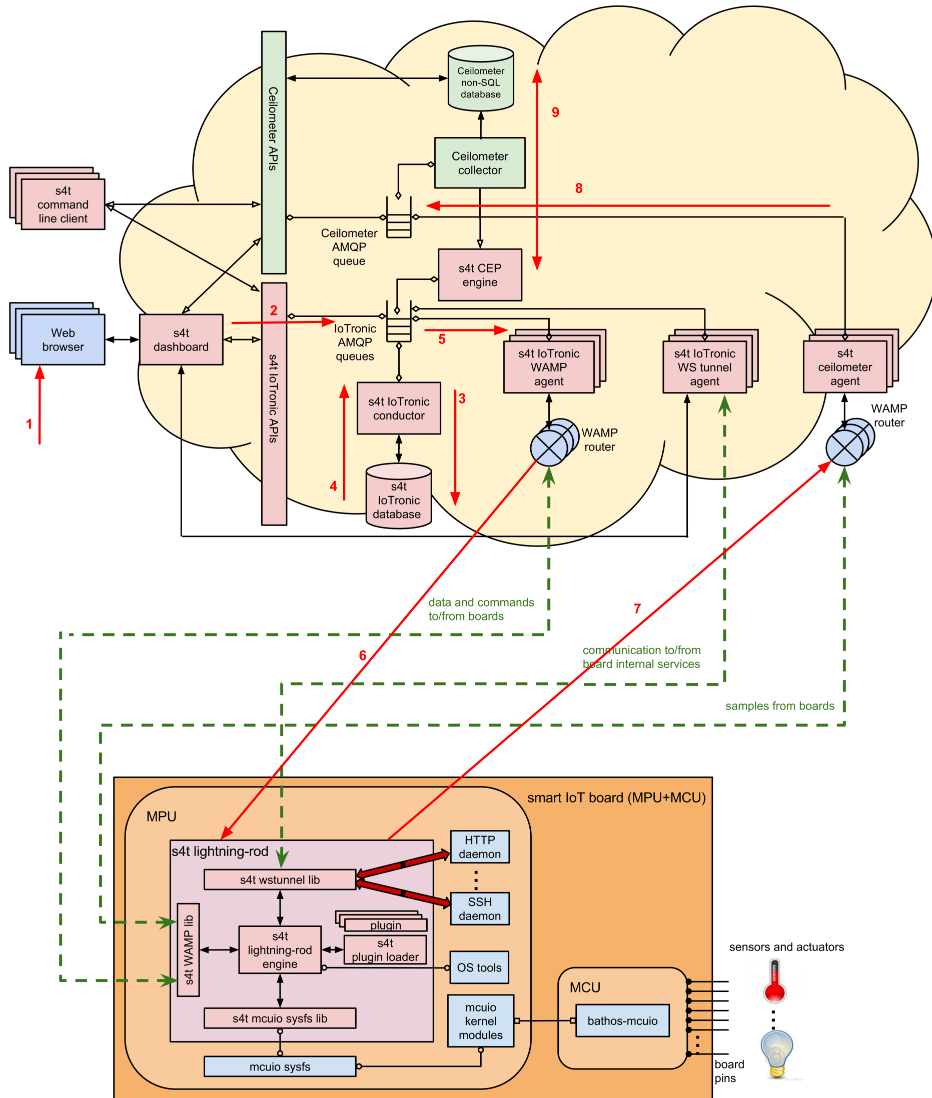


Figure 2.8: *S4T: storing readings from a sensor*

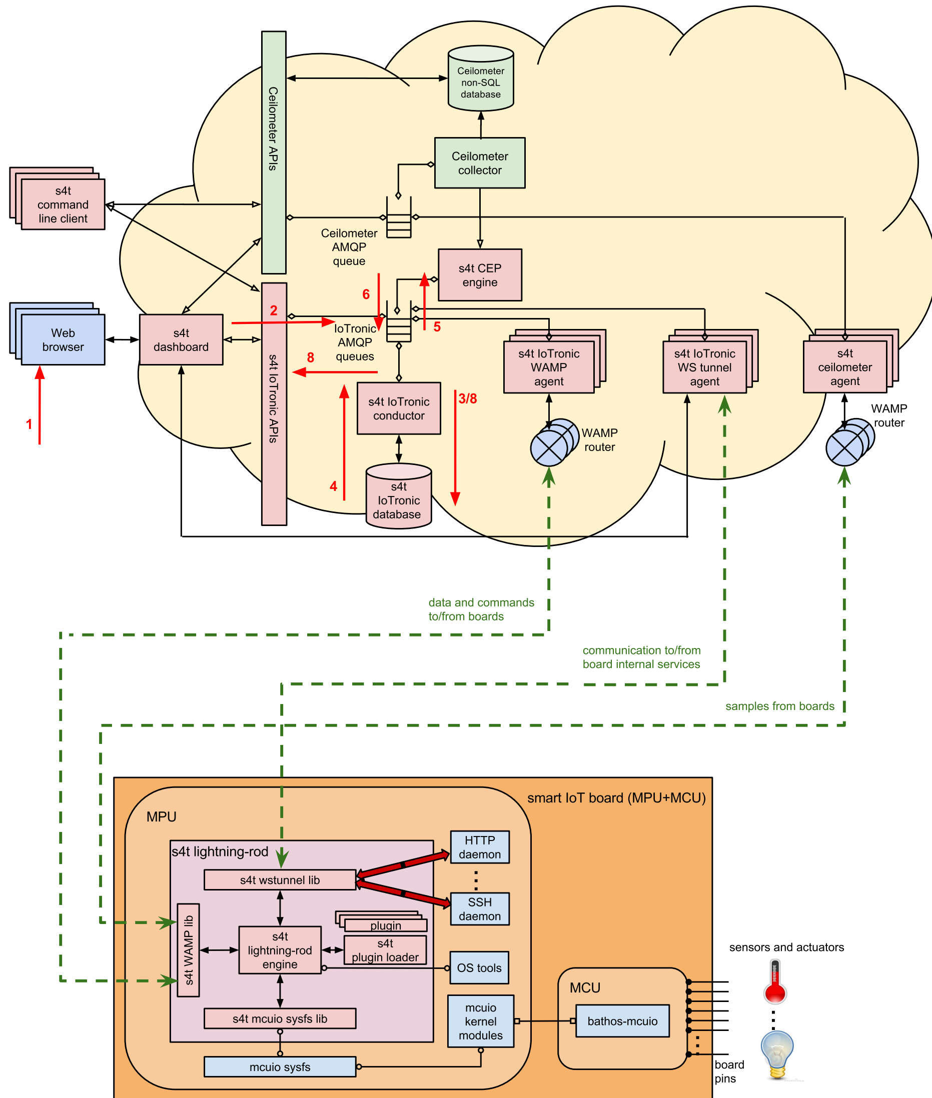


Figure 2.9: *S4T: injection of a CEP rule*

CHAPTER

THREE

MOBILE CROWDSENSING AS A SERVICE: A PLATFORM FOR OPPORTUNISTIC SENSING

3.1 Introduction and motivations

Current trends, with specific regard to cyber physical systems and Internet of Things (IoT), suggest that one of the most interesting thrusts towards pervasive services comes from opportunistic and participatory sensing paradigms, such as Mobile Crowd Sensing (MCS). MCS leverages the pervasiveness of smartphones and other portable devices, enabling users and community groups to collectively share data from onboard sensing resources so as to measure phenomena of common interest, exploiting social dynamics. The contributor has also the possibility to augment raw data with context as metadata. This community-driven sensing trend is brought about by machine interactions at different levels, including data communications, collection, processing and mining. Commencing crowd-sourcing and sensing duties from mobiles, involving device owners as volunteering partici-

pants, potentially renders end users both contributors and consumers of large volumes of (curated) data.

However, there are several key issues that need to be addressed for the MCS paradigm to experience widespread adoption [58, 59]. Firstly, a unified architecture for supporting MCS applications is required to enable reusability of software components, facilitating shorter time to market cycles. Existing MCS applications are built as stand-alone ones, while common challenges, e.g., related to resource engagement, get independently revisited each time, or are not addressed at all. The heterogeneity of mobile Operating Systems (OS) and sensor hardware further amplifies the problem. As a result sensing and processing activities usually result in carrying out similar tasks (i) within a single device (contributing node) for different MCS applications, resulting in energy starvation and (ii) across multiple, neighboring devices, leading to spikes in bandwidth usage and processing requirements at the back end. Both cases highlight a non-scalable deployment model.

Furthermore, MCS applications rely on every single contributor for local deployment duties. While most MCS apps require a critical mass of contributors to be deemed useful, app adoption is bounded by the rate at which users keep track of, and install, newly introduced ones. A substantially low rate as, according to recent reports, e.g., in 2014 nearly 80% of the 1.2 million apps available at the Apple App Store had hardly any downloads at all[1]. Providing resources on a volunteer basis is one of the foremost limitations to the large scale exploitation of the MCS paradigm, as it naturally bears constraints related to contribution churn. In terms of MCS applications this sets the need for "marketing" strategies aimed at increasing the number of subscriptions, stimulating, retaining and rewarding potential contributors through incentives. However, even if the enrollment activities (or even mechanisms, such as credit-reward systems) may be highly

[1] https://www.adjust.com/assets/downloads/AppleAppStore_Report2014.pdf

effective, the subscription process is usually characterized by long and smooth dynamics. Therefore a significant time delay may be experienced before getting a significant stream of sensing data. This issue may strongly confine the scope of the MCS paradigm to a shortlist of applications featuring broad, long-established supporting communities of potential contributors.

From a different perspective, another significant trend moves IoT towards service-oriented and Cloud computing paradigms [60, 61, 62]. In this view, the Cloud is not just a technology to support the archiving of sensed data coming from pervasive IoT infrastructure, but also a model and a paradigm to adopt in the management of the underlying resources and things.

Following this IoT-Cloud research trajectory, in this chapter, a novel platform for opportunistic (mass) exploitation of contributed resources for MCS apps is presented to get more insights as to whether the MCS paradigm may indeed be applied at large-scale in the IoT context. The proposed approach overcomes several of the aforementioned hurdles, by facilitating what is essentially the most difficult endeavor for prospective MCS entrepreneurs, i.e., offering a level playground, with homogeneous access to wildly different underlying ecosystems. To this end, two classes of concerns are identified with regard to (unassisted) dissemination of MCS apps; (i) *infrastructure*-related, mostly focused on mechanisms to enroll and manage voluntarily contributing nodes, as well as to abstract sensing resources and enable uniform access, and (ii) *application*-related, mainly devoted to mechanisms for interfacing with enabled infrastructure, asking for the required sensing resources and, once obtained, deploying the MCS app onto the resource-hosting nodes. This way infrastructure resources (*supply*) may be decoupled from application requirements (*demand*) as in Cloud contexts, making development, deployment and operation fully independent.

For delivering this novel MCS approach, the Sensing and Actuation as a Service (SAaaS) framework, proposed in [16], is adopted as the lower-level (infrastructure domain) enabler. SAaaS is based on the service-oriented (and Cloud-inspired) approach of elastically providing (virtual) sensing and actuation resources on demand, gathered from underlying (contributed) physical nodes. The study at hand extends and adapts the SAaaS paradigm for enabling rapid MCS app deployment and streamlining their operation, actually providing an *MCS as a Service* (MCSaaS) platform.

There are a number of advantages in dealing with MCS from the (SA)aaS angle. Virtualizing and customizing sensing resources, starting from the capabilities provided by the SAaaS model, allows for concurrent exploitation of pools of devices by several platform/application providers. Delivering MCSaaS may further simplify the provisioning of sensing and processing activities within a device or across a pool of devices. Moreover, decoupling the MCS application from the infrastructure promotes the roles of a sensing infrastructure provider that enrolls and manages contributing node(s), below, and of a platform provider on top of that, acting as a broker between (i) the former and (ii) the MCS application provider, enabling the latter to focus on the application and leave concerns and enforcement about requirements (type of resources, availability, etc.) to the platform.

In this chapter a high level overview of the proposed MCSaaS approach is provided complemented by a description of the distinct architectural elements and their interactions. Moreover the benefits of the proposed framework are here highlighted as opposed to the conventional MCS approach, by means of (i) a prototype implementation of the MCSaaS framework and the emulation of a real-world application deployment; and by (ii) modeling with generalized stochastic Petri nets (GSPN) [63].

3.2 Preliminary concepts and related work

3.2.1 Mobile Crowd-Sensing

MCS [58] is an emerging trend, lying at the intersection of volunteer and crowd-based computing, IoT, and sensing paradigms. It refers to a broad range of community-powered sensing approaches belonging to either *participatory* [64] and/or *opportunistic* [65] categories, aiming at involving large population of contributors [59, 66].

A broad range of applications, from mining of urban dynamics [67], to public safety [68], traffic and environment monitoring [69, 70], smart lighting [71] and smart cities [72], just to name a few, may be implemented adopting the MCS paradigm. Existing MCS applications are comprised of two main components [58]:

i) device-specific ones, for data collection, execution of local analytics if needed, and data dissemination,

ii) the backend, for extensive data analysis, storage and visualization duties.

A simple and generic MCS application scenario is shown in Figure 3.1, where the main elements and stakeholders involved in the MCS system are identified based on [58]. These include the contributing *nodes*, e.g., smartphones, tablets and, PDAs, shared by their *Owners* or *Contributors* to build up the *sensing infrastructure environment*, as well as the *Application Service Provider* (MCS ASP), that manages and supervises the whole process, gathering and processing sensed data through the application server, which also interfaces with the *End User* of the MCS application.

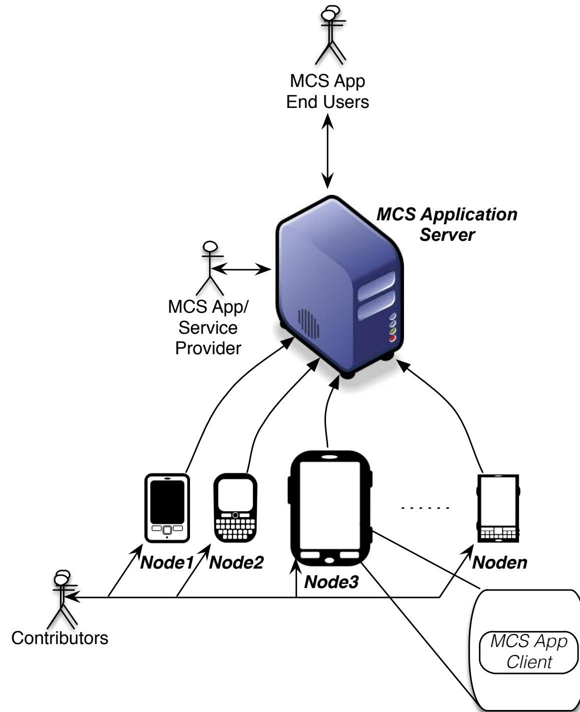


Figure 3.1: *MCS: application scenario*

3.2.2 MCS taxonomy

One of the main categorizations of MCS spans the participatory-opportunistic spectrum. On one hand, participatory sensing may be defined as any crowd-sourced sensing activity where each member of the crowd is actively involved, giving feedback when asked or otherwise tagging measurements on a voluntary basis. Conversely, an opportunistic perspective sensing is essentially *unmanned*: MCS would tap into mobile devices just because people carry those around in their pockets all day long anyway. Thus also the device *owners* may be included in the data feeding process. Their mobility and situation awareness may be leveraged, in an opportunistic and participatory

fashion, to support the collection of fine grained information and semantically tagged data.

Without proper incentives, the owners may not be willing to contribute with their resources. For the MCS success appropriate *incentive mechanisms* are required to recruit, engage and retain human participants [73]. In this sense, a centralized credit system, assigning and managing credits and rewards, is usually adopted as incentive mechanism in a participatory strategy, while, in opportunistic scenarios, the gamification approach [74] is preferred. As a more subtle distinction of the two approaches, *users* benefiting from the crowd-sourcing may be identified. In participatory MCS systems, and services that may derive from it, the community, or the public at large is the main target of the benefits. Whereas in opportunistic MCS systems it is single individuals mainly taking advantage.

Still according to the end-user perspective, another aspect to be considered is how information produced by an MCS system is consumed, or made relevant to the situation under which *fruition* would occur. A typical proactive, participatory pattern for users may consist in merely consulting an MCS-derived knowledge base, thus leveraging information as-is, i.e., non-contextually and in a pull fashion. Conversely, an opportunistic fruition mode would be based implemented by push-based notifications. Moreover, also in terms of *interactions*, at least first-time enrollment requires input on the side of owner on a client-server model basis. Yet even opportunistic schemes, featuring distributed behavior and cooperative strategies, may be considered, dependent on the underlying topology, as is the case for mesh-like ones in device-dense environments.

A synthesis of the approaches and categories of MCS applications is presented in Table 3.1. This way, a wide range of possibilities for MCS application paradigms, from pure participatory to wholly opportunistic ones, may be identified, also including hybrid solutions.

Table 3.1: *Taxonomy of MCS applications*

***MCS* categorized (by)**	**Approach**	
	Participatory	*Opportunistic*
Owner involvement	Active, human-assisted sensing / tagging	Background, unmanned data collection
Incentive mechanism	Credit systems (bank)	Credit collection race
User benefit	Public interest	Individual utility
Fruition modality	Pull / non-contextual	Push / contextual
Interaction model	Centralized (client-server)	Distributed (mesh)

3.2.3 Related work

So far, several MCS applications [75, 76, 69, 77] have been developed in different contexts, demonstrating the MCS paradigm is useful in applications directly and indirectly involving different stakeholders and huge populations of users. This has drawn the attention of both the academic and business communities, which, on one hand, started developing new middlewares implementing mechanisms and tools for MCS system management [78, 59] while, on the other, are investigating potential exploitations of the MCS approach both in scientific applications and in commercial ones. Therefore, the current state of affairs highlights the need for suitable methodologies and techniques able to bring order in the MCS field, adopting engineering practices and tools to explore the untapped potential of the paradigm.

With specific regard to the issues raised in the previous section, several frameworks have been proposed to support expedite MCS application development and deployment. AnonySense [79] is a privacy-aware framework for opportunistic and participatory sensing. Applications specify the sensing task behavior using a Domain Specific Language (AnonyTL) and then submit it to the AnonySense components and mobile nodes. A polling model is used for task distribution. Downloaded tasks are matched to the nodes based on their context.

Medusa [78] is a programming framework that specifies the workflow of sensing tasks to be executed in smartphones and onto the Cloud. It defines the Med-Script programming language that pro-

vides high-level abstractions for the various stages in crowd-sensing tasks as well as for flow control. Moreover it provides a distributed runtime between the Cloud and smartphones.

Pogo [80] proposes a middleware for building large scale sensing testbeds using mobile phones. Both researchers and device owners, each category running the middleware differently, depending on their role. Pogo relies on the XMPP protocol for communication between nodes. Experiments are written in JavaScript.

Vita [81] supports the development, deployment and management of multiple MCS applications/tasks. It consists of both a mobile and a Cloud platform. The former provides the appropriate services (e.g., map and location) that enable the execution of sensing tasks. It optimizes the allocation of a task to a group of users or Cloud servers by using Genetic Algorithms and K-means clustering. The Cloud platform streamlines the development and deployment of MCS applications, integrates and stores uploaded sensing data, as well as metrics related to system operation.

PRISM [82] is also a platform for community-sensing applications that allows the deployment of binaries ready for execution onto mobiles. Method call interposition is used to sandbox-untrusted applications to ensure security and privacy. PRISM follows a push-based model for the automatic deployment of applications to an appropriate set of phones. Efficient tracking of mobiles is implemented mitigating privacy risks and reducing communication overhead.

Most of the aforementioned frameworks propose custom languages for hardware and OS abstraction of the contributing devices or specify ad-hoc sandboxed environments for secure execution of applications. Although convenient, the fact that only pre-programmed functions and software modules are provided to developers results in a less flexible and not as powerful application development environment. In fact, only PRISM [82] allows for native MCS applications deployment

(although sensors can only be accessed through a sandbox), while also providing a mechanism for selection of contributing devices. By not spinning a service-oriented model for (sensing) infrastructure out from the platform that is meant to exploit it, sandboxing and access to hardware resources need to be crafted explicitly for the platform itself, instead of delegating such duties to an IaaS-level framework.

With regard to the deployment of sensing applications and tasks, the above frameworks allow for rapid installation and execution on all contributing devices indiscriminately. Additionally, Medusa provides specific rewards and incentives to stimulate user participation while it allows smartphone owners to specify limits on usage of system components. However, most of these frameworks do not provide any advanced mechanisms for targeted selection of contributing devices, despite that such a process could significantly increase efficiency in the exploitation of available resources, as well as minimize impact on existing MCS-dependent services running on the same devices. This is a consequence of the lack of infrastructure ready to be exploited in a controlled way for MCS, as resources are still to be enrolled with ad-hoc mechanisms as provided by the aforementioned platforms.

In closing, it is notable that none of the above frameworks tackles support to contribution churn as a platform-provided feature, as a way to enable on-demand expansion or shrinkage of the user base of MCS apps, according to the needs of application developers and providers.

3.3 MCSaaS paradigm

3.3.1 Vision

The MCSaaS framework attempts to address the basic limitations of existing MCS applications, such as explicit and time-consuming developer efforts in the engagement of resources, while instead support-

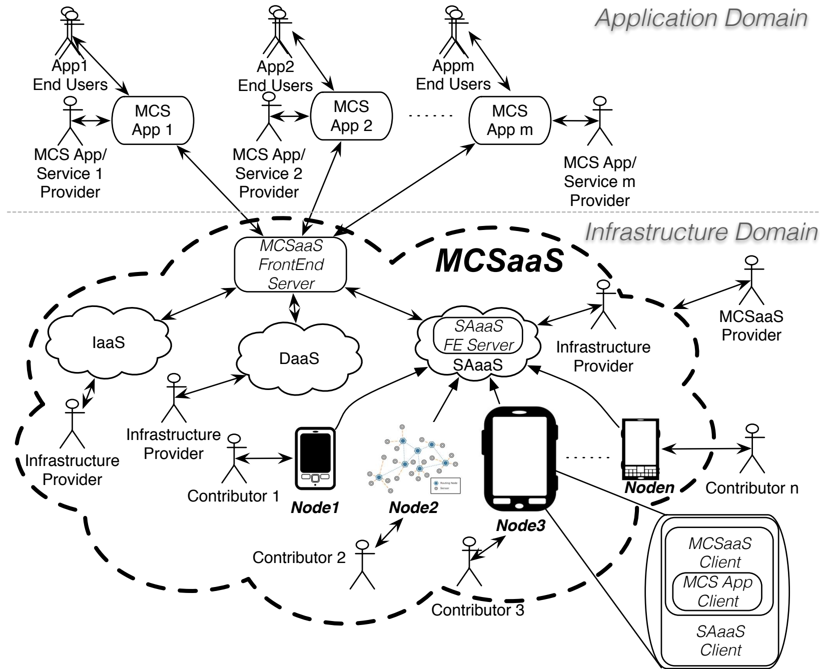


Figure 3.2: *MCSaaS: scenario*

ing contributor recruitment by MCS ASPs. Specifically, in MCSaaS these issues are addressed by making infrastructure available on demand. This way, MCS application/service providers can immediately deploy and run their applications and services on promptly available resources, ready to use once configured or customized for deploying the MCS application. Another important benefit of this approach lies in having real-time requirements fulfilled mostly for free, by design due to the paradigm shift, since QoS requirements can be satisfied by providing a certain guaranteed number of devices in face of loss.

From a high level point of view, this approach can be implemented by functionally and administratively splitting the MCS application/service deployment into two domains: i) the *infrastructure* domain, which includes (embedded) sensing devices, providing ser-

vices for resource management (brokering, interoperability, etc.) and facilities for customizing and deploying the MCS application; and ii) the *application* domain, hosting the frontend and backend (analytics) services for the (filtered and pre-processed) data provided by the infrastructure modules, exploiting infrastructure/low level *application deployment* and *data/node management* facilities to implement the MCS application or service.

This strongly impacts on the MCS paradigm, significantly changing the scenario by introducing new stakeholders, and enabling further avenues for exploitation, not only in terms of research but also from a business perspective, such as an open market of MCS(aaS) sensing resources and services. More specifically, as can be seen in Figure 3.2, different application service providers (MCS ASPs) may leverage the facilities an MCSaaS platform provider has on offer, including traditional enterprise-level infrastructure providers such as the incumbent players for both processing and storage resources, i.e., Infrastructure as a Service (IaaS) and Data as a Service (DaaS), respectively. Under the sensing Cloud a few kinds of infrastructure contributors are depicted: as we are talking about SAaaS here, the owners / admins are contributors sharing resources under their control, such as mobiles, PDAs or even Wireless Sensor Networks (WSNs).

The differences between the "traditional" MCS scenario and the proposed MCSaaS one are clear by comparing Figure 3.1 and Figure 3.2, and have been summarized in Table 3.2, where the main MCSaaS actors with their duties, as well as the pros and cons of their roles, are described. More specifically, within the "plain" MCS domain (Figure 3.1), a potential contributor directly interacts with an MCS ASP to be engaged in an MCS activity. According to the MCSaaS scenario in Figure 3.2, contributors are node owners/administrators that are sharing resources under their control via registration to a sensing Infrastructure Provider (SaaS), possibly retained by means of suitable

Actor	*Description*	
	Pros	**Cons**
Contributor	*Volunteering contributor of sensing resources, stimulated by appropriate incentives.*	
	- Possibility to earn credits, rewards. - Free or remunerated. - Multiple MCS application contributions through a single subscription (delegation pros). - Possibility to be both user and contributor. - Privacy and security due to two layers of isolation.	- The node contribution is initially managed by a third party broker (delegation cons).
InP	*A sensing infrastructure provider enrolls and manages contributors according to specific service level agreement.*	
	- Enlarge the business customer portfolio (due to mashups).	- Third party brokering and monitoring.
MCSaaS-P	*The MCSaaS-P provides resources mashup and brokerage services to MCS ASPs, along with customization service for the engaged resources and MCS application deployment service.*	
	- New business opportunities. - Increased resource utilization and throughput. - To reach a wide audience. - Big data - analytics. - Involving sensor networks. - Actuation.	- Resource management. - Duties on security, privacy and SLA to both sides.
MCS ASP	*Application provider that delivers a single or multiple MCS applications/services to MCS End Users. The MCSaaS ASP deploys the MCS application(s) utilizing the MCSaaS framework.*	
	- Wider application domains involving mobile and/or static (SN) sensors and actuators. - No problems of enrollment and management of sensing resources. - QoS-guaranteed resource provisioning. - Real-time applications suitability. - Increased application reliability, availability and performance. - Capillarity, worldwide coverage, # of contributors. - Heterogeneous resources (computing, storage, sensing) provided. - Facilities for the application deployment (configuration, customization, setup, analytics tools). - Customizability of resources (pre-processing, filtering, client-side functionalities, reduced overhead, bandwidth-local processing trade-off). - Abstracted/homogeneous access (APIs) where resources are what is customized to meet the needed abstraction (SAaaS-unique). - Resources handling capabilities due to the device/resource-centric approach vs the data-centric one, enabling enhanced features and new applications. - New applications involving actuation resources.	- May be charged.
MCS EU	*The consumer of the MCS services provided by the MCS ASP.*	
	- High performance, low delays. - The role of an MCS EU may naturally coincide with the role of Contributor.	- May be charged.

Table 3.2: *MCSaaS scenario: actors*

incentives. The contribution is therefore managed at a lower level, thus lending a degree of freedom to resource sharing (multiple MCS app contributions, contribution profiles, credits, money, etc.) but this requires to delegate resource control to the infrastructure provider. The MCSaaS scenario is thus enriched by new stakeholders, such as the Infrastructure Providers (InP) and the MCSaaS Provider (MCSaaS-

P) in between the MCS ASP and the Contributors. This brings about several benefits and is advantageous in particular for MCS ASPs, who can rapidly develop and deploy their apps onto MCSaaS-enabled infrastructure.

A service-oriented provisioning model is the best way for the MCSaaS-P to provide the required resources to MCS ASP, enabling customization facilities while ensuring the required service levels for provisioning. The MCSaaS-P can provide support to single or multiple MCS applications/services to be delivered transparently to MCS End Users (EU), as in the traditional MCS scenario. In order for the MCSaaS-P to provide the requested sensing resources to the MCS ASP, the two parties have to negotiate the set of required resources and then, upon agreement, the ASP deploys the MCS app to the corresponding set of registered contributing nodes provided by the InP through the MCSaaS platform. Details on this process are reported in Section 3.6.

3.3.2 Stack

The main idea we propose in this work is therefore to adopt a Cloud and service-oriented approach for the on-demand provisioning of MCS resources and services. This way, the SAaaS sensing Cloud becomes a pillar of the MCSaaS infrastructure. Furthermore, a higher platform-like layer to provide services for the resource management (mashup, brokering, interoperation, etc.) as well as for deploying and customizing the app on top of such infrastructure is mandatory. Sensing and actuation resources are involved in the Cloud not as simple endpoints, as in current mobile Cloud trends, but rather in the same way as computing, storage, and network resources usually are in more traditional Cloud stacks: abstracted, virtualized and grouped in Clouds, to unlock new value-added services by mixing the potential of the Cloud

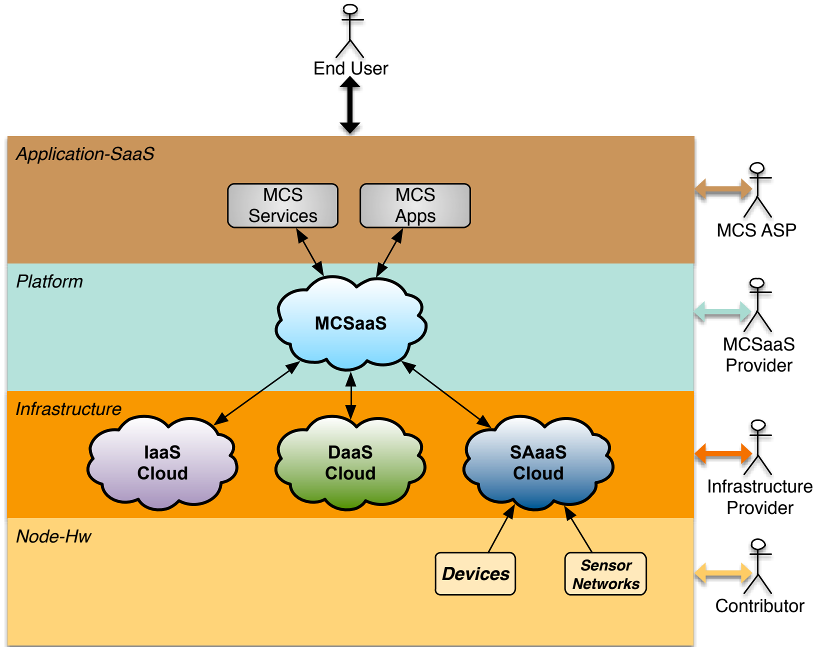


Figure 3.3: *MCSaaS: stack*

with that of the IoT.

Our goal is therefore to provide a conceptual framework, and correspondingly a software stack, able to deal with such issues, while aiming at the MCSaaS vision as the whole. To this purpose, we adopt a multi-tiered approach as depicted in Figure 3.3, where a layered scheme comprising the node and the infrastructure Clouds below the platform (PaaS) and the application-Software as a Service (SaaS) on top, is proposed.

At a lower level of the MCSaaS stack we have contributors sharing their nodes with infrastructure providers that enroll them to provide (virtual) sensing and actuation resources as a service (SAaaS). Optionally also computing (IaaS) and storage (DaaS) providers could be included at this level. On top of the basic infrastructure mechanisms,

services mainly related to the specific configuration, customization, and management of virtual resources for MCS applications are provided by the MCSaaS Cloud platform. Facilities to expose resources provided by different categories (i.e., IaaS, DaaS, SAaaS) of infrastructure providers may be implemented by the MCSaaS platform. At a higher level, an MCS application employs such infrastructure and platform facilities to eventually enable and provide SaaS services. However, an MCS ASP may just deploy an application, collecting and/or displaying data, without necessarily building up a Web service out of it, or otherwise becoming an (MCS-powered) Software as a Service provider.

3.4 Infrastructure

As seen in the previous section, the MCSaaS scenario may involve third parties to provide IaaS/DaaS resources where needed for developers of MCS applications, but here the focus is on the main building block of any kind of MCS-centered facility: (mobile) sensing infrastructure, under the guise of service-oriented Cloud-enabled resources, a so-called Sensing and Actuation as a Service (SAaaS).

SAaaS is a paradigm aimed at developing a sensing infrastructure based on sensors and actuators from both mobile devices and SNs, providing virtual sensing and actuation resources in a Cloud-like fashion. More specifically, it delivers the basic mechanisms and tools for enabling a Cloud of sensors and actuators, which have to be suitably extended and customized by providers to implement enhanced services and provisioning models. To this end, the main issues to be addressed include: abstraction of sensing and actuation resources, virtualization, customization, monitoring, SLA and QoS management, subscription, churn and policy management, enrollment, indexing and discovery, security and fault tolerance.

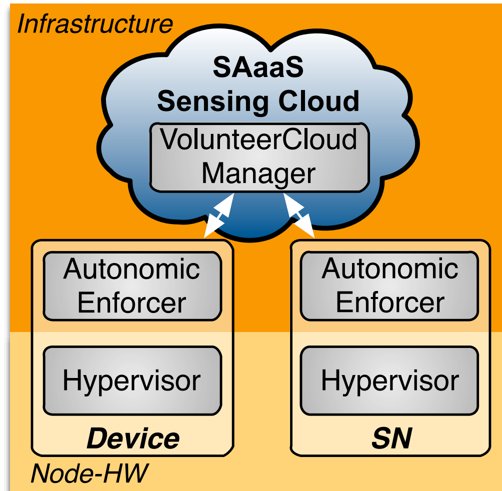


Figure 3.4: *SAaaS: reference architecture*

This fosters the design of a software stack that implements the following main functionalities: (i) involvement of SNs, smartphones or other devices endowed with sensors and/or actuators, and their enablement for interoperation in a Cloud environment; (ii) distributed mechanisms and tools for self-management, configuration and adaptation of nodes; (iii) functions and interfaces for enabling and managing contributing resources.

To implement such ambitious idea, i.e., a Cloud of sensors based on the SAaaS paradigm, in [16] the whole stack was introduced with a high-level blueprint of the architectural modules, the three main components of which are shown in Figure 3.4, bottom-up: Hypervisor, Autonomic Enforcer and VolunteerCloud Manager.

The SAaaS stack and modules span to the two lower layers of Figure 3.3: from the Cloud Infrastructure one, providing support to the MCSaaS PaaS and MCS SaaS software applications and services, to the node layer, covering *edge* devices. Through the SAaaS stack,

any device, either mobile or static, may be engaged in crowdsensing activities, as well as any sensor network, regardless of the software environment and operating system. This way, the term SAaaS *node* is used in the following to indicate a smart device equipped with sensors, such as a smartphone, or a frontend to a possibly large number of smaller sensing devices (i.e., motes), such as the gateway of an SN.

Furthermore, since the SAaaS implements a *device-centric* approach, providing actual sensing and actuation resources, even if multiplexed and/or virtualized, it allows for customization and configuration capabilities typically unexposed higher up to MCS ASPs, that may in turn explore previously neglected application domains.

The lowest block, the Hypervisor, operates at the level of a single node, where it abstracts the available sensors. The node can be a standalone resource-rich device, such as a smartphone, or it can be an embedded system which belongs to a network (such as a WSN). The main duties of the Hypervisor are: communications and networking virtualization primitives, abstract description of devices and capabilities according to the relevant information model, virtualization of sensing resources, customization, isolation, semantic labeling.

The Cloud modules, under the guise of an Autonomic Enforcer and a VolunteerCloud Manager, deal with issues related to the interaction among nodes. The former is responsible of the node-internal enforcement of Cloud policies, local vs. global policy tie-breaking, cooperation on overlay instantiation, enrollment initiation and subscription(s) bookkeeping. Whenever suitable it operates according to autonomic principles. The latter is instead in charge of the following functionalities: exposing the Cloud of sensors via Web Service interfaces, indexing of subscribers and contributed resources, monitoring of Service Level Agreements (SLAs) and Quality of Service (QoS) metrics. These layers thus form a coupled, two-level Cloud stack, where many mechanisms are split in both modules, dealing with node-wise

actions and self-organizing properties in the lower one, and centrally managed Cloud-wise methods in the upper one.

Indeed, taking as an example device enrollment, the lower Cloud module is in charge of node-side initiation of the enrollment process, including one-time interaction with the contributor, e.g., for excluding or limiting access to certain device-hosted resources, by pushing the description of enumerated resources to the Cloud, as well as of bookkeeping of any Cloud instance the node has successfully subscribed to. The centralized module is instead in charge of Cloud-side acceptance or rejection of the enrollment request as well as indexing of resources (and the corresponding nodes) to provide querying capabilities to the end user of the Cloud.

At this level, having communication among the Cloud modules layered on top of ubiquitously available protocols and services for IoT and M2M such as WebSockets [38] means getting access to lots of available gadgets and personal devices that would otherwise go untapped, unless major revisions in terms of their firmware, and communication stacks in particular, get planned. This choice directly reverberates on the widening possibilities a MCS app developer may experience as a result of the potential expansion of the pool of resources.

3.5 Platform: MCSaaS module

For the design of the MCSaaS stack, the need comes up to define what an MCSaaS platform should offer. PaaS is an intermediate service model for Cloud computing, between IaaS and SaaS, where the consumer creates software using tools, libraries, etc., available from the provider, also controlling software deployment and configuration settings. The PaaS provider usually hosts lots of ancillary facilities to the main infrastructure under consideration (in this case SAaaS mainly), e.g., networks, servers, storage, sensors.

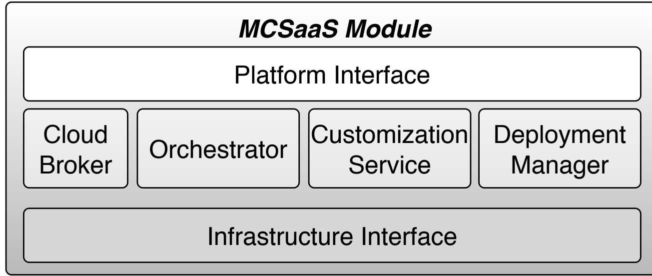


Figure 3.5: *MCSaaS module*

A platform like MCSaaS must cater for domain-specific APIs for MCS, such as pre-filtering and pre-processing mechanisms to be leveraged both at the node endpoint (e.g., the mobile) and at the Cloud, while considering the trade-off between communication overhead and local computation. Other APIs would include general-purpose analytics frameworks, in this case to be leveraged only at IaaS/DaaS level. Issues related to federation, inter- or multi-Cloud setups, privacy, security and trust, which would ideally fall under the scope of MCSaaS, have not been investigated for this specific effort. Thus application hosting and a deployment environment are the main focus in this context. Moreover a PaaS typically also includes facilities for application design, application development and testing as well as typical services for developers and integrators, such as team collaboration, Web service integration, database-driven persistence, state management, application versioning, application instrumentation / profiling and facilities for community nurturing.

All aforementioned functionalities are synthesized in the MCSaaS module implementing a PaaS service at the platform layer (see Figure 3.5). The *Infrastructure Interface* provides the means to interact with the underlying sensing Clouds. Standard interfaces such as OCCI [83] and/or specific techniques following the multi-Cloud approach can be adopted. On top of the Infrastructure Interface, four MCSaaS core

sub-modules are defined, namely Cloud Broker, Orchestrator, Customization Service and Deployment Manager.

An incoming MCSaaS request is forwarded by the *Platform Interface* to these sub-modules. Such interface should be RESTful and expose as entities useful abstractions, e.g., application bundle objects. This is initially managed by the *Orchestrator*, which identifies the resources required by the MCS application extracting requirements and dependencies from the request encoding the high level workflow of the application. This kind of service may conform to available standards, e.g., TOSCA [84], in terms of exposed functionalities and relevant APIs. It therefore iteratively interacts with the *Cloud Broker* to reserve resources according to the aforementioned requirements provided by the MCS ASP, the Broker in turn planning in advance the engagement of sensing devices as needed through SAaaS InPs. Following a successful negotiation and the allocation of required resources to the MCS application the application setup phase is launched by both the Customization Service and the Deployment Manager. The *Customization Service* mainly focuses on customizing the resources and (virtual) devices, setting specific low-level parameters such as duty cycle, sampling frequency and scale range for sensing resources, as well as high level configurations of the software environment hosting the injected code of the MCS application. This is achieved by translating high-level directives in terms of uniform low-level primitives, still expressed in a generic form, as those are ultimately relayed to the SAaaS for further processing. The *Deployment Manager* instead is aimed at deploying the required modules on the available resources, adapting them to the hosting environment. Pre-configured packages or bundles may be provided, where the payload may contain a whole application environment or parts of it, or even extra tools that implement advanced features such as analytics, interfaces, domain-related plugins and add-ons. As the latter modules are here the main focus

for the initial implementation, in Section 3.6 the workflows involved are described and in Section 3.7 more details are given in terms of the corresponding software design.

3.6 Setup and deployment of MCS applications

Within the proposed paradigm, the actions and interactions of the main stakeholders (depicted in Figure 3.2) through the blocks and the modules identified above, are of high practical importance. Thus, the main activities related to MCS application setup, deployment and management into the MCSaaS framework, are described through the following Activity Diagrams (AD). More specifically, two main perspectives associated with the MCSaaS-P and the MCS ASP are taken into account in the description of these main activities, which include (i) setting up an MCSaaS platform (Section 3.6.1), followed by (ii) the configuration and deployment of a specific MCS application on it (Section 3.6.2), respectively.

3.6.1 MCSaaS platform setup

With regard to enablement of PaaS over potentially exploitable infrastructure (SAaaS), there is the need to first describe a bootstrap scenario, where mandatory MCSaaS components are pushed to the mobiles in a PaaS-agnostic way. In more detail, the following kind of interaction is envisioned: the MCSaaS provider needs to choose the appropriate infrastructure, given a set of available InPs, to offer MCS as a service.

Afterwards a phase of bootstrap ensues, where the MCSaaS-P must enable every mobile, booked through the SAaaS InP, for easy MCSaaS-assisted deployment of MCS applications. For this purpose SAaaS-

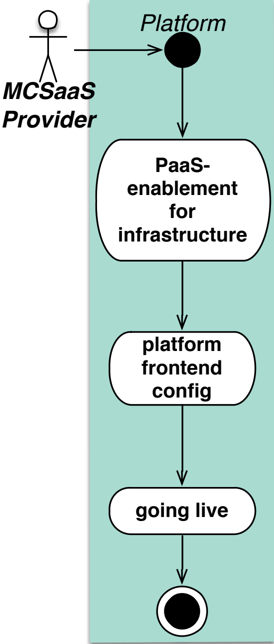


Figure 3.6: *MCSaaS: initial platform setup AD*

provided services, powered by the Hypervisor, are employed to deploy the node-side MCSaaS module.

The MCSaaS module enables contributors to choose their level of involvement and resource sharing for MCS apps, i.e., not only sensing resources as in pure SAaaS scenarios, but also CPU time, memory and/or on-board storage. Moreover, this module is in charge of duties related to MCS app deployment, i.e., activating endpoints for, and then cooperating with, the Deployment Manager in Figure 3.5, also enforcing the aforementioned choices in terms of local computing and storage resources when receiving and deploying the next MCS app.

The aforementioned approach, apart from avoiding layering (IaaS / PaaS) violations, also enables the implementation of the mobile-side SAaaS components to cope just with sensing infrastructure concerns and SAaaS scenarios. This focus then carries over also in terms of sandboxing and other security-related mechanisms, e.g. dealing with user-mandated restrictions, only with regard to SAaaS-relevant sensing resource usage and interactions.

In the same fashion, any node-side operations dictated by the Customization Service, would just be dealt with by the bootstrapped environment, thus leading to the development of certain security-related mechanisms, such as checking permissions of MCS apps in terms of, e.g., access to any kind of user data, to be implemented only at this level, i.e., in the MCSaaS bootstrapped component itself.

Figure 3.6 depicts MCSaaS-P activities related to the platform setup, starting with a macro-step including all actions involved in *PaaS-enablement for infrastructure*, such as identifying and selecting potential InPs, exposing relevant APIs and service endpoints. Afterwards a phase of *platform frontend configuration* follows: this entails the setup of a Software as a Service instance, i.e., a Rapid Application Development (RAD) environment available to application developers. As soon as the frontend is ready, the MCSaaS-P can "go live" and

start servicing customers, e.g. MCS ASPs.

3.6.2 MCS application configuration and deployment

The MCS application configuration / deployment is performed by the MCS ASP using the services provided by an MCSaaS-P and is broken down into the (i) application negotiation and platform bootstrap stages and (ii) the actual app deployment, detailed in Section 3.6.2 and Section 3.6.2 respectively.

Negotiation and Bootstrap

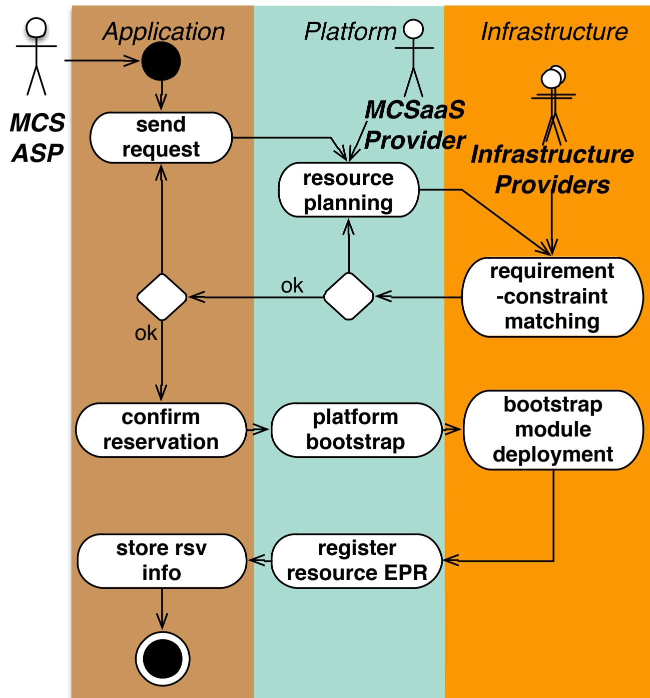


Figure 3.7: *MCSaaS: app negotiation and platform bootstrap AD*

Figure 3.7 describes the (PaaS-mediated) app negotiation and platform bootstrap stages. The former entails identifying the required building blocks for the app, in terms of IaaS/DaaS/SAaaS resources, with a phase where the developer (or MCS ASP) pushes the corresponding plan to the platform frontend (*resource planning*). It is followed by a negotiation of the requirements to be matched, coordinated by the Cloud Broker, possibly with iterations involving the MCS-ASP still into the planning stages (*requirement-constraint matching*). Upon agreement and following the MCS ASP confirmation (*confirm reservation*), the platform modules have to be deployed into the provided sensing resources, through the interaction between the MCSaaS-P (*platform bootstrap*) and the InP (*bootstrap module deployment*). Thus, the end-point references (EPR) of the reserved sensing resources are fed back to the MCSaaS-P and stored (*register resource EPR*) and the reservation data forwarded to the MCS ASP (*store rsv info*).

Deployment

Field deployment of an MCS app, as detailed in Figure 3.8, starts with the MCS ASP choosing among available APIs (*choice of API*), for basic as well as advanced operations (e.g. pre-filtering, analytics). This is followed by the platform's *artifact contextualization* of the MCS application and the configuration of the infrastructure sensing resources on the specific application domain (*app domain configuration*), translating at the same time customer-driven constraints on resources and APIs by wiring up infrastructure endpoints accordingly (*service endpoint wiring*).

Then, an initial set of requirements is submitted by the application (*requirement push*), which is the "recipe" (including high-level code) obtained during the bootstrap phases as discussed in Section 3.6.2. This step kickstarts the *artifact deployment* (binaries, configurations,

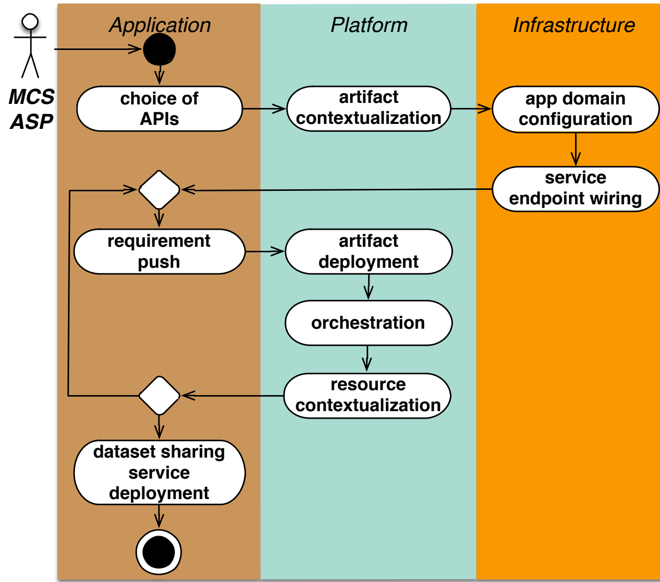


Figure 3.8: *MCSaaS: app deployment AD*

system images, etc.) to VMs, storage objects and, in the case of SAaaS, contributor-owned mobiles. Behind the scenes the deployment would be preceded by translation of recipes in artifacts, e.g. compilation/packaging, still up to the Deployment Manager. Activation of the endpoints, and subsequent mapping of the wiring (as shown in the corresponding AD of Figure 3.8) over allocated resources (*orchestration*), are up to the Orchestrator. For the operations of the Orchestrator to be effective, or even just feasible under most circumstances, platform-initiated activations and (re)wirings are required, thus leading to a need for an always-on (anytime) push-to-client messaging channel for each registered device, powered by environment-agnostic bi-directional asynchronous exchange primitives.

Afterwards, a phase of *resource contextualization*, by means of the Customization Service, is called for, either in terms of configuring or even customizing the underlying resources, e.g., by iteratively in-

volving the SAaaS services, where relevant. Mapping and, especially, customization could possibly lead to a mismatch between initial requirements and actual setup, thus requiring eventual iterations with MCS ASP to reach a convergence or at least a satisfactory trade-off before stopping the matchmaking process. Furthermore, the MCS ASP can start exposing a portal or useful Web services to share and visualize full datasets, or any compact representation thereof in the *dataset sharing service deployment*. These datasets may therefore be provided to third parties (also as OpenData) to let them develop applications centered around such datastores.

3.7 The MCSaaS implementation

In the following a description and some details regarding the MCSaaS stack implementation are provided.

With regard to the node-side runtime, there are the Sensing APIs and the environment-provided notification subsystem which are of particular relevance to this design. Whilst the former is opaque in terms of the MCSaaS, as its Customization Service has to relay sensor reconfiguration and tuning requests to the SAaaS device-side subsystem, the latter is useful at both Cloud layers, to minimally involve platform-provided Cloud-based mechanisms for push-based communication to devices, avoiding to tackle corner cases in terms of reachability, and tracking of networking conditions in general.

The server-side logic, available in the Orchestrator and the Cloud Broker, has been developed in Java and Python, resorting to servlets [85] in terms of the user-facing interaction model and RESTful endpoints as Cloud-private interfaces to the mechanisms implemented in Python.

Apache Tomcat [86] is used as the Java servlet container. For example, Google notification service, Google Cloud Messaging

(GCM) [87], is used for exchanging (MCSaaS-bootstrapping) asynchronous messages in Android-powered mobiles, as needed to support Cloud-initiated primitives and runtime customizable mechanisms, as well as for on-demand activation of a WebSocket-based subsystem. This preference towards a (partially) custom communication bus lies in the inherent limitations (and costs) borne by the use of GCM, which is simply not meant for big payloads (e.g., file transfers), but is specifically designed and marketed for push notifications and in this case employed also for signaling.

In terms of the main modules, the Broker keeps track of the reservations and transfers resource requests to the Orchestrator. The latter in the current implementation mainly deals with churn, which at MCSaaS level means reacting to fluctuations in the number of active contributors for any of the MCS apps hosted by the system. In particular, as soon as in a certain area such population falls below a threshold, according to parameters set by the MCS developer, the churn management routines running in the Orchestrator look up other platform-enabled nodes in the same area to push notifications through GCM, meant for triggering the nodes to retrieve the package, install and launch the corresponding payload, the MCS app. The whole MCSaaS population is continuously under tracking so any query is expected to be serviced in the order of fractions of a second. The client-side component of the Deployment Manager listens for incoming payloads, tagged as such by the server-side logic, to automate the installation of the app as soon as it is downloaded onto the device. This is to let the app be installed without any user intervention, albeit the process gets inevitably visible at times, as some windows would pop up anyway during installation stages, albeit for very short time intervals (e.g., less than a second for any window instance).

3.8 MCS app: case study

In this section examples of MCS applications deployed using the MC-SaaS platform are discussed, highlighting the benefits of MCSaaS. To compare the proposed approach against the traditional MCS one, two possible use cases in an IoT / Smart City scenario are considered, involving two different MCS applications: (i) a multi-purpose community-focused app for pothole mapping and (ii) a traffic monitoring app serving the Messina, Milan and Athens urban areas (Figure 3.9).

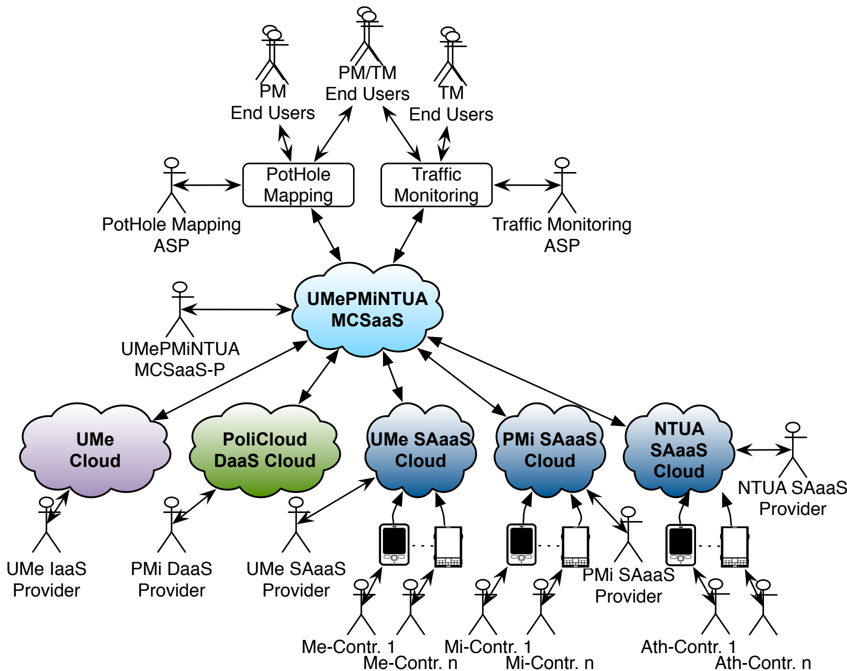


Figure 3.9: *MCSaaS-driven app: deployment scenario*

The UMe (University of Messina), PMi (Politecnico di Milano) and NTUA (National Technical University of Athens) SAaaS (InPs) are used in the deployment scenario, leveraging their private IaaS and DaaS providers if needed (such as the OpenStack-powered UMe and

PMi PoliCloud [88]). However, any private or public company/institution (e.g., mobile telcos) could take up the role of the InP. These SAaaS Clouds feature smartphone sensors from volunteering Contributors (roaming users). Smartphones are bound to be extremely useful in such scenarios, as they are equipped with several relevant sensors e.g., for positioning (GPS,GSM, WiFi) and motion detection (gyroscope), while supporting (always-on) Internet access. Roaming users become Contributors by discovering and (non-exclusively) subscribing to SAaaS Clouds.

The UMePMiNTUA MCSaaS-P is established according to the workflow depicted in Figure 3.6. After the selection of the appropriate InPs (subscribed to services, signed SLAs, etc.), UMePMiNTUA discovers, collects and exposes generic (templatized) WS endpoints (URIs), typically for RESTful consumption, and corresponding APIs documentation within an HTML5-powered RAD IDE.

On top of this platform, the pothole mapping and traffic monitoring apps are provided by two different MCS ASPs. Contributors may be engaged for both pothole mapping and traffic monitoring MCS apps concurrently, a desired outcome and one of the main drivers behind this effort. The MCS End User (EU) (e.g., a taxi driver), uses the corresponding mobile (or Web-based) apps for retrieving the information of interest. Often an MCS EU also acts as a Contributor.

In the following, it is initially discussed how to set up and deploy these two MCS apps (Sections 3.8.1 and 3.8.2) using the proposed MSCaaS platform. Then, in Section 3.8.3, (i) preliminary results of the prototype in the context of the traffic monitoring use case are reported, and (ii) benefits of the MCSaaS approach against the conventional MCS one highlighted through an analytic model.

3.8.1 Pothole mapping

A pothole mapping MCS application automatically collects road condition information once it is started, without human intervention. It requires just an accelerometer and a positioning system at the core, to be sampled for detection of abrupt vertical displacements and geolocalization. Data validation including the removal of outliers and elimination of false positives is facilitated by the crowdsensing approach collecting, analyzing and clustering input coming from a diverse range of users.

In the current MCS app deployment scenario, the pothole app requires just a database and a server for the Web UI. Therefore planning the requirements in terms of IaaS/DaaS is quite straightforward. On the other hand, the high-level constraints for the sensing infrastructure to be transmitted / negotiated include (i) sensing potential (accelerometers and positioning at least) (ii) geographical area and (iii) device mobility (vehicular, e.g., bikes and cars). The app design stage requires to leverage a set of libraries and ready-made routines, a typical process for a RAD environment, for, e.g., mobile-side outliers detection, as well as IaaS APIs to choose from (OGC- or M2M-compliant REST calls) etc., following the PaaS approach.

Once the app is released, the pothole ASP has two main alternatives: directly enrolling contributors supporting the pothole mapping service in a given area of interest (traditional MCS) or asking an MCSaaS-P, e.g., UMePMiNTUA, to provide the required (vehicular) sensing resources. As discussed above, in the former case, user enrollment could be slow or even not successful, while in the other case the MCS ASP has immediate access to the sensing infrastructure at the cost of the provided service (MCSaaS). Combining the two approaches in a kind of "cloudbursting" fashion, the pothole ASP may resort on-demand to third party sensing resources provided by the

UMePMiNTUA MCSaaS-P along the “owned” ones directly engaged by the MCS, when required, e.g. in the case accuracy in the mapping is required within a given time constraint.

The pothole ASP has to negotiate with the UMePMiNTUA MCSaaS-P for the resources, required by the pothole mapping app that needs to be deployed in the MCSaaS platform/infrastructure. In the case of successful negotiation, the sensing resources provided by the UMePMiNTUA MCSaaS-P have to be set up and configured to receive the pothole app code as shown in Figure 3.7.

The pothole app deployment begins as soon as the developer lets the output (*recipe*) of the design stage be consumed by the Deployment Manager, in this case just dealing with a limited subset of binaries (e.g., compatible with Android-powered mobiles), with regard to sensing resources (Figure 3.8). Finally, at instantiation time endpoints that are provided/consumed by the application get enabled and the (abstract) wiring in the recipe gets mapped to the infrastructure by the Orchestrator.

3.8.2 Traffic monitoring

Another application of the MCS paradigm may be traffic monitoring, which is a useful instance of service based on mobile contributors, still related to drivers also as a potential pool of consumers and/or producers, and in general in the same domain. In terms of MCS, while the sensing activities are similar to pothole mapping, e.g., still mainly enabled by positioning subsystems and accelerometers, the requirements diverge remarkably.

As overall traffic and specific metrics, such as current cruising speed and the rate of slowdowns, to name a few, need to be fed and updated in *real-time* for the service to be genuinely useful, it follows that there are constraints such as, e.g., the minimum number of con-

tributors per area on average, to ensure accuracy on traffic monitoring sampling. Most importantly the platform is meant for automatic deployment of multiple MCS apps if needed, ready to be executed side by side, so a scenario where both the pothole mapping application and the traffic monitoring ones are concurrently working in background is absolutely feasible and contemplated, while still addressing the specifics of the requirements of each application, e.g., real-time constraints on the availability of actively contributing resources in the case of traffic monitoring only.

The latter is really a use case for the orchestration capabilities of the platform, as exemplified by the management of node churn, which needs to be addressed on a continuous basis, to provide the expected quality of service, or at least degrading gracefully by informing the developer about the number of currently (or recently) involved contributors per area, in other words, under a simplified scheme, the confidence interval with respect to displayed metrics.

This way, resorting to the MCSaaS provider UMePMiNTUA is maybe the only effective solution to ensure a predefined level of accuracy in sampling for the traffic monitoring service. Also in this case a hybrid MCS-MCSaaS solution may be effective, exploiting the (vehicular) sensing resources provided by the UMePMiNTUA MCSaaS-P to complement and enrich the information gathered from the contributing ones engaged by the traffic monitoring ASP through a traditional MCS enrollment process. On the other hand, the enrollment of sensing resources from UMePMiNTUA follows the negotiation and deployment/configuration phases described in Figure 3.7 and Figure 3.8.

3.8.3 Testing and evaluation

To demonstrate the effectiveness of the MCSaaS model against the traditional MCS one, evaluations follow of (i) a preliminary imple-

mentation of the stack for the deployment of the traffic monitoring app and (ii) the corresponding GSPN models.

MCS Application Deployment: A Proof of Concept

The traffic monitoring application, denoted hereafter as MCS-Traffic app, falls under the category of *infrastructure* MCS apps [58]. The implementation of the MCS-Traffic application follows the same principles as similar community-based GPS-enabled applications for navigation (e.g., Waze [89]), using contributor's location and travel times via GPS to provide real-time traffic updates, as exemplified in Section 3.8.2. To show the benefits of using the proposed paradigm for the MCS ASP, as opposed to current MCS application deployment practices, an illustrative example related to the MCS-Traffic app is presented, specifically focusing on issues such as volunteer-based contribution and node churn.

Testing Scenario The MCS-Traffic app requires real-time information (via GPS tracking) to provide updates on (expected) travel times for an arterial road (service area). Specifically, the minimum number of actively probing vehicles (contributors) is crucial for establishing the reliability of estimations in terms of link speed [90]. A link is a section of road spanning a continuous segment with no intersections, while the link speed refers to the distance traveled by a vehicle in a unit of time.

The following testing scenario is considered: vehicles enter the service area according to a Poisson process and travel times are normally distributed [91]. It is assumed that data from at least 10 probes are required with a sampling period of 700s to establish the reliability of the estimate on link speed [90]. This way, the traffic monitoring app is able to correctly estimate and predict actual traffic in the area of interest.

In experiments an average population of 150 vehicles is assumed to be in the service area. Furthermore, it is assumed that just a part of them are also SAaaS contributors (about 8%), and similarly that just about 5% of vehicles are MCS contributors, directly engaged by the ASP. These values are arbitrary, however, since a conventional MCS application needs to recruit, engage and retain participants, such assumptions may be considered reasonable, possibly a bit conservative in the MCSaaS case.

Testing Environment The prototype implementation at MCSaaS and SaaS levels, described in Section 3.7, has been used to test the deployment and operation of the MCS-Traffic app. MCS EUs / contributors were emulated (running Android 4.2 / API level 8 [92]), due to the lack of a substantial number of physical devices, as required by the testing scenario. Android has been adopted, not only for its widespread availability and popularity, but also for some of the facilities the SDK and the runtime environment provide, e.g. the emulation subsystem (using Genymotion [93]) and the Debug Bridge (ADB) [94] for the management of emulated instances. Helper functions, using shell scripting, were employed for setting up the testing scenario (e.g., vehicle mobility, engagement etc.) along with ADB for provisioning and control of the emulated instances.

Experimentation Metrics The two different scenarios discussed in Section 3.3 are examined in experiments based upon the MCS-Traffic app: (i) the one envisioned under the conventional MCS paradigm and (ii) the one enabled by the proposed MCSaaS paradigm.

These two scenarios are compared based on the average number of vehicles within the service area as a function of time, serving as active probes (*Number of Contributors*). For this purpose, enumerate are the (i) MCS contributors (MCS-C), involved in the conventional

MCS scenario; and (ii) MCSaaS contributors (MCSaaS-C), contributing to the traffic monitoring app through the whole stack. MCSaaS contributors are a subset of users engaged by the SaaS provider (InP) that are denoted as SAaaS contributors (SAaaS-C).

In addition, the effect of *Resource Churn* is quantified, by measuring on average the time required by the framework to replace MCSaaS contributors that left the service area, restoring the MCSaaS pool to the number required by the MCS-Traffic app.

Results and Comparison In the MCSaaS scenario, it is up to the SAaaS provider (InP) to ensure a consistent high number of contributors engaged to a MCSaaS-P. Then it is up to the MCSaaS-P in its turn to reserve at least the minimum required amount of MCS contributors, according to the application requirements (in this case the MCS-Traffic app).

Number of Contributors: For the set of sampling periods, on average 5.8 contributors for the conventional MCS scenario and 9.2 contributors for the MCSaaS one, excluding a warm-up period of 500 s, have been estimated. Comparing the results obtained in the case of the MCSaaS scenario to the conventional one, it can be argued that the MCSaaS solution succeeds in engaging contributors within the service area, as it approximately meets the constraint posed by the MCS-Traffic app in terms of the number of active probes (10 probes are required).

The effectiveness of the proposed approach in addressing the requirements of the application, is also visible in Figure 3.10 that is a snapshot of the emulation within a sampling period, just reporting on a single testing trace. Following the 20th time unit there are two, almost simultaneous, SAaaS contributor departures that affect the MCSaaS provisioning service. However the framework is able to promptly react, restoring the MCSaaS pool to 10 contributors/probes

as requested by the MCS-Traffic app.

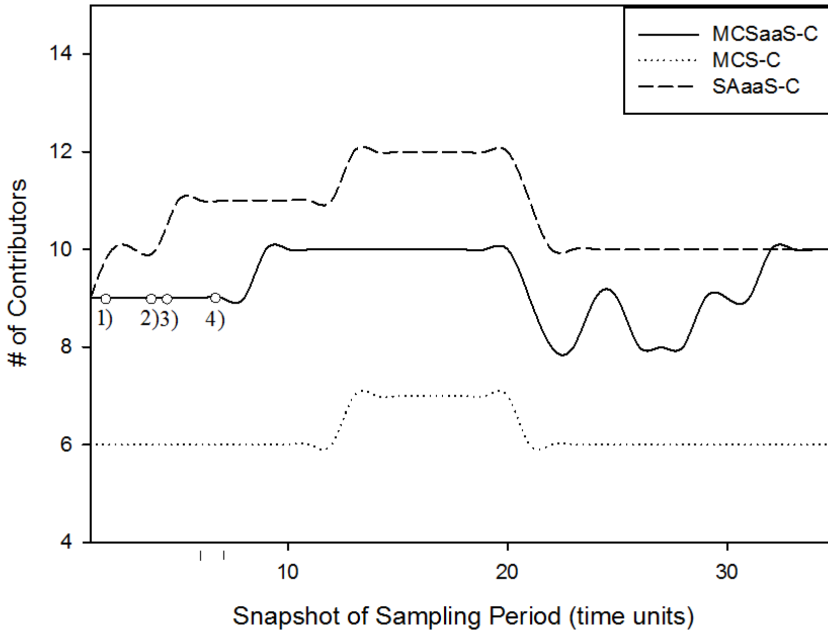


Figure 3.10: *MCS and MCSaaS emulation: contribution sampling*

Resource Churn: Through the experiments a lag of 6 time units (30 s) on average has been evaluated, due to the resource churn functionality, for the MCSaaS system to actually replace contributors (MCSaaS-C) that left the system. This lag is broken down into four intervals:

1) the time needed for the SAaaS server to identify the arrival of a new contributor and initiate the MCSaaS client installation process;

2) the MCSaaS client installation time;

3) the time needed for the MCSaaS server to identify the arrival of a new contributor and initiate the MCS-Traffic app installation;

4) the application installation time.

This delay is also captured in the MCSaaS curve of Figure 3.10, highlighted in the [0, 10] time unit interval.

In this evaluation the overhead introduced by the abstraction of sensor resources and registration of infrastructure services has been neglected, since in [95] metrics pertaining to the abstraction overhead of sensing resources, at the SAaaS level, was measured and found to be considered negligible for all purposes. With regard to registration, it can be prospected as a typically one-time (SAaaS-level) operation, so it is not so critical in the long term.

A quantitative, in-depth evaluation on the impact of the churn management in terms of performance cannot be performed due to the limits of the testbed and the preliminary implementation of the framework. Nonetheless, some general, qualitative observations, based on the experiments conducted, are reported in the following. The impact of resource churn (and subsequent updates of resources) on the performance of the service to be offered (i.e., the MCS application) is rather limited by design, as long as any replacement is application-transparent (as is the case in general for MCSaaS). The overall application behavior may depend on the smoothness of the curve depicting the ratio of available resources to the requested ones over time, as in the case of the traffic monitoring app. Reactivity is thus key mostly when dealing with loss of huge numbers of contributed devices. App deployment time, depending on bandwidth and concurrent (non-blocking) fan-out, is mostly constant, not proportional to the number of devices to be replaced.

CHAPTER

FOUR

A CROWD-COOPERATIVE APPROACH FOR INTELLIGENT TRANSPORTATION SYSTEMS

4.1 Introduction and motivations

Among the strategic services addressing societal challenges, many governments give priority to mobility and transportation, pushing for Intelligent Transportation Systems (ITS). Some interesting results in this direction come from ICT through crowdsourcing. If properly exploited the potential of new technologies and approaches is expected to provide a great contribution to the development of effective ITS solutions. With an ever growing availability of embedded, mostly personal and mobile computing devices for everyday tasks, there is an almost limitless potential for tapping onboard resources.

A promising way to exploit this untapped potential is Mobile CrowdSensing (MCS) [58]. It aims at gathering and harnessing the power and wisdom of the crowds to deal mainly with human-centric problems, typically in social, urban and citizen science applications. MCS comprises by definition applications where individuals carrying sensor-hosting embedded systems such as smartphones get collectively

engaged in information gathering and sharing efforts to monitor and georeference events which may be of interest for individuals and communities alike. It gets applied successfully in different, more or less intelligent, transportation systems and services, e.g., monitoring of traffic and road conditions, mapping of road network features and of cues for elements of interest.

One of the main advantages of MCS is the possibility to perform sample collection, data mining, etc., without accounting for the corresponding experiments in advance, just leveraging natural daily life patterns, arising from human activities, as they happen and leave behind breadcrumbs in form of samplings ready to be collected. Aim of this kind of enablement then is putting this power at the fingertips of developers or would-be entrepreneurs, ready to kickstart whichever effort in next to no time. In particular self-provisioning and autonomous cooperation are needed to avoid long setup times for experiments, disruptions beyond careful planning and sizing, as well as reducing the development of ad-hoc solutions. MCS is already establishing itself as a trendy paradigm, but most efforts go into the direction of easing *participatory* (e.g. manned) patterns. Apart from privacy and security issues, where anonymization and sandboxing respectively are key countermeasures, most engagement chances should be meant to be *opportunistic* to let MCS be truly widespread, inexpensive and wholly disruptive as a paradigm. In this context, problems lie foremost in enabling unassisted deployments, as well as accommodating for peer-oriented communication and distributed self-organization, mostly due to real-world settings and constraints, e.g. intermittently disconnected operation.

Most MCS applications typically feature a common, simplified architecture, made up of two main components, one running on the embedded device to collect and disseminate measurements, and a second one as backend. The main drawback of such siloed pattern lies

in missing exploitation of proximity or density in topologies. In particular this last point is crucial, as any kind of high-density scenario, especially when it includes real-time constraints as in ITS, needs a smart approach to proactively take advantage of proximal nodes and crowded areas instead of crumbling under the weight of such scale. Given the MCS paradigm and forthcoming use cases with specific regard to ITS, where mobility is really going to match crowds at scale, an opportunistic design pattern may be conceived. In particular, the MCS approach perfectly fits with traffic application requirements, asking for frequent, nearly real-time, updates in both monitoring and route planning decision making. But, due to the limitation of current (mainly participatory) MCS patterns, it has been effectively adopted just in traffic monitoring applications [69, 70]. In this chapter the aim is to fill this gap, demonstrating that the MCS paradigm, through novel opportunistic patterns, can also support and implement self organizing systems able to autonomously decide on the best route for a vehicle based on the traffic information gathered from neighbors. This way, the proposal is to use this new MCS development in route planning, to implement a new generation of ITS.

4.2 Background and related work

4.2.1 An overview of ITS

A directive by the European Union Commission [96] defines ITS as "systems in which information and communication technologies are applied in the field of road transport, including infrastructure, vehicles and users, and in traffic management and mobility management, as well as for interfaces with other modes of transport". From the ICT perspective ITS may be envisioned to embrace any advanced solution that integrates live data and other feedback from a number

of heterogeneous sources, such as parking guidance and information systems.

In particular, efforts related to ITS seem naturally poised to have as target high-population density areas and to consider multimodal systems of transportation comprising either personal vehicles or shared carriers for commuters, such as buses and trains. This way ITS naturally spans a wide range of technologies, starting from basic management systems such as navigation ones, possibly to be augmented in the future by systems where artificial "co-drivers" may assist humans during their duties [97].

Yet, there are many other examples of instances of subsystems prone to be enhanced through ICT, e.g., traffic signal control systems, which may leverage some kind of system-optimal routing algorithm for road networks as well, such as game-theory based ones [98]. Moreover, from a technological viewpoint, any delay in information dissemination for vehicle-to-vehicle communication networks [99], so called VANETs [100], considering a traffic-dense configuration as the relevant scenario, can be identified as one of the main challenges to be overcome for any coordination system to really work as expected.

Even established technologies, such as Wireless Sensor Networks, may find peculiar applications in ITS such as monitoring of conditions for railways [101], or be revised to meet novel requirements, such as challenging wireless communication scenarios in high-speed railway systems [102]. Some authors [103] have leveraged Deep Learning to predict traffic flows by dealing with Big Data sources. Such problems were also analyzed by model-based solutions: for instance in [104] a stochastic (hazard-based) model to evaluate the impact of a reliability-safety tradeoff on the travel-time is proposed.

4.2.2 MCS for ITS

Several success stories demonstrate the capability of the MCS paradigm to support the development of successful ITS applications. Among them, mapping activities such as OpenStreetMap [105] and BikeNet [76], aimed at creating an open geographic map of the road networks or bike routes worldwide, respectively, or more specific pothole detection and mapping application [69], gained consensus and large scale participation. Anyway, due to the MCS potential for providing near real-time information, the ITS-related killer application for this paradigm is traffic monitoring. Indeed, there are several solutions proposed in literature addressing this problem from different perspectives. First and foremost, CarTel [106] and NeriCell [69] pioneered tackling this problem by a sensing infrastructure deployed on-purpose, or mobiles, respectively. Then, more advanced solutions have combined both static and mobile sensors, involving street sensors such as cameras as well as smartphones or vehicle sensors. For instance in [107] a traffic monitoring system for a public transportation service is implemented, and similarly in [70] for vehicle traffic in general. A step further has been achieved in terms of integrating data crowdsourced from social networks as well, towards vehicular social networks (VSN) [108, 109], where a full layered stack, including a framework for developing context-aware applications for VSN thus established, is proposed [110]. Also large-scale, commercial solutions have developed traffic monitoring services (partially) based on mobile contributions, such as those included in Google Maps (Google traffic [111]), MapQuest and Baidu services.

These works demonstrate that, on one hand the MCS paradigm perfectly fits with traffic monitoring goals, but on the other hand it is not applied in applications requiring active involvement of end-users/contributors through a control loop or a feedback, such as, in

the ITS context, route planning. This is likely due to some limitations of the current wisdom about the MCS paradigm itself, mainly focused on participatory patterns instead of opportunistic ones. Indeed, in the development of ITS for traffic engineering and optimization of flows and transportation resources, two main solutions exist: the managed or the unsupervised (cooperative) approaches. The former means recurring to centralized systems, such as municipality-controlled street lights, timetables and relative information systems. The latter requires leveraging cooperation among traffic vectors, such as private vehicles or buses.

Furthermore, a traffic engineering system may be translated into a purely distributed, network mesh-dependent subsystem. Identifying *Internet* as the connection facility and *things* as vehicles leads to a straightforward mapping onto the Internet of Things (IoT) paradigm, sometimes also declined in literature [112] as *Internet of Vehicles*. Under such premise, this ITS characterization of MCS may just become a pattern under the IoT umbrella term, i.e. a specialization of the platform that an IoT would represent for sensing-related, mobility-enabled, crowd-sourced use cases.

4.3 A novel cooperative strategy

In this section an innovative distributed MCS scenario is going to be proposed, and therefore a stigmergic strategy for achieving optimization goals through distributed cooperation of autonomous MCS nodes.

4.3.1 A distributed MCS pattern

Framing the discussion in the aforementioned characterization, a scenario is briefly described, where cooperation among nodes becomes important, even essential, especially in the context of mobility and

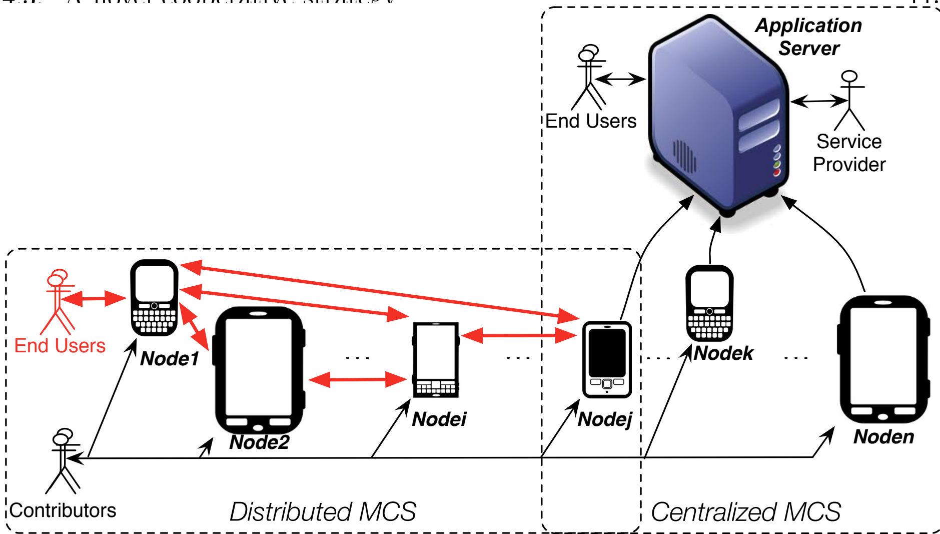


Figure 4.1: *MCS patterns: centralized and distributed*

transportation applications. This way, the focus is on MCS opportunistic patterns, mainly characterized by push fruition modality and distributed interaction model, according to the taxonomy in Table 3.1 (Section 3.2.2).

As shown on the right part of Figure 4.1, typical (*centralized*) MCS applications mainly implement a client-server interaction pattern where a *service provider* offers MCS-based services to *end users*, leveraging *contributors* willingness to provide their physical (sensing) resources. Data are therefore collected and processed by (backend and frontend) *application servers* to carry out analytics and feed back relevant results to end users. As discussed above this approach does not allow to properly exploit the power of the underlying resources at the edge of the IoT, restricting the applicability of the MCS paradigm to just client/server applications thus requiring a centralized coordination.

A way to fully exploit this unexpressed potential is by adopting a *distributed* approach. This aspect is depicted on the left of Figure 4.1, where the main differences between the decentralized versus

the traditional/centralized MCS patterns are highlighted in red. The main one is the opportunistic cooperation, a collaborative approach by which nodes may interact one another to aid local computations and perform distributed optimization on a small/medium scale. This way, end users may leverage an MCS application by just exploiting cooperation among nodes. In this scenario contributors usually also act as end users and viceversa, directly interacting through their nodes/devices, and there is no service provider. Furthermore, it is worth remarking that a contributor could be involved in different MCS applications, therefore, as highlighted in Figure 4.1, the same node may be involved in both centralized and distributed contribution patterns.

To the best of our knowledge, this is the first attempt at exploiting opportunistic, cooperative approaches in MCS contexts. Some work on opportunistic IoT and sensing environment is available in literature. For example, in [113] an opportunistic IoT framework is proposed, mainly extending opportunistic networking towards participatory sensing, enabling opportunistic information sharing among things to also support mobile social networking. Similarly, opportunistic mobile networking is the topic of [114], mainly focusing on low level data forwarding issues through a framework able to support and optimize opportunistic sensing. Differently from these approaches, here the work is placed at a high level, proposing an alternative or, to be more precise, a complementary approach to the centralized MCS paradigm, where backend-less operations is possible, as cooperation would work unimpaired anyway, at least in steady-state stages of execution. Existing solutions, as the aforementioned ones, mainly operate at network level and are surely of interest for the implementation of our ideas when dealing with such concerns, which anyway are out of the scope of the present chapter, focused on introducing and motivating an opportunistic-cooperative approach specifically geared for MCS.

4.3.2 Stigmergic approach

To demonstrate the suitability of the distributed MCS pattern, effective methods and tools enabling opportunistic features are required, in particular with a focus on an ITS scenario, e.g., vehicular traffic engineering in a urban setting.

Over such a set of (dynamic) meshes, such as a vehicular crowd, a stigmergic approach is proposed for cooperation and optimization. Let's first tackle swarm optimization alone.

Ant colony optimization

Ant colony optimization (ACO) [115] is a relatively recent metaheuristic based on the behavior of ants seeking a path between their colony and a source of food. In nature wandering ants have exhibited in this sense a provable capability to discover nearly optimal paths. The collective intelligence of the swarm derives from the indirect exchange of information among ants via the environment (the so-called *stigmergy*). While traveling to search for food, ants lay down pheromones on their way back to the nest (i.e., home colony) only when sources of food are found. As other colony members step into pheromone trails, they tend to stick to the beaten path accordingly. Moreover, the trace gets reinforced as more individuals follow the same trail, leaving pheromone of their own, in turn resulting increasingly attractive for other ants. For any complex problem which can be reduced to a search for optimal paths, ACO may work as a probabilistic solver, by emulating such naturally occurring behavior. The probability p_{ij}^k for an artificial ant k, placed in vertex i, to move toward node j is defined as follows:

$$p_{ij}^k = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{l \in \mathbf{N}_i^k} (\tau_{il}^{\alpha} \cdot \eta_{il}^{\beta})} \tag{4.1}$$

where τ_{ij} corresponds to the *quantity of pheromones* laid over arc a_{ij},

η_{ij} to *a-priori attractiveness* of the move, computed by some heuristic embedding the cost of choosing arc a_{ij} along the path that leads to the destination, and $\mathbf{N}_i^k$ is the *set of neighbors* in node i for ant k, i.e., the nodes directly reachable by the ant. Coefficients α and β are global parameters for the algorithm, and are typically both set equal to 1, so that pheromones and a-priori information have the same importance in the choice of the arc. According to typical ACO variants, ants bring food back home after being done with their movement. Denoting T^k as the *tour* of ant k and n as the number of elapsed rounds, C^k is defined as the *length* of T^k, and used to specify the amount of pheromones to be placed by ant k on each arc on the trail leading to the food source:

$$\Delta\tau_{ij}^k = \left\{ \begin{array}{lr} \frac{1}{C^k} & \text{if arc } (i,j) \text{ belongs to } T^k \\ 0 & \text{otherwise} \end{array} \right\} \tag{4.2}$$

$$\tau_{ij}(n+1) = \tau_{ij}(n) + \sum_{k=1}^{M} \Delta\tau_{ij}^k \tag{4.3}$$

where M is the total number of ants in the colony.

At the end of a round, after each ant has completed a move, the extent of pheromones laid over each arc gets reduced (e.g. evaporates), according to:

$$\tau_{ij}(n+1) = (1-\rho)\tau_{ij}(n) \tag{4.4}$$

where ρ is a global *evaporation parameter* as well, typically set in literature with values ranging around 0.5, i.e., halving the pheromone value at each iteration.

ACO algorithms yield their best performance when some form of local search algorithm is employed.

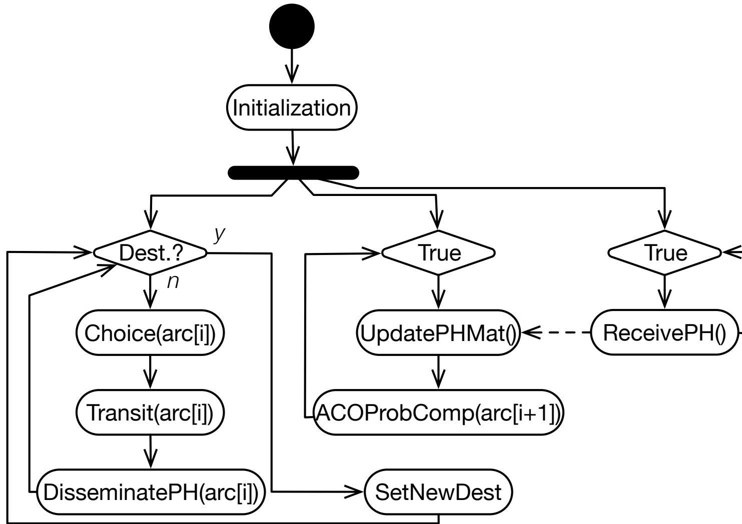


Figure 4.2: *MoCSACO: activity diagram*

ACO-based MCS

In order to adapt ACOs to the envisioned cooperative MCS-based application, an adaptation of an ACO, called *MoCSACO*, is proposed. Its behavior is described in Figure 4.2, where an ant corresponds to a (physical) mobile device, i.e., a *real-world agent*, such as a vehicle.

Before delving into the detailed description of Figure 4.2, the overall rationale of this adaptation needs to be outlined.

The general objective of finding the shortest path on a (weighted) graph is here being redefined in terms of leveraging common, state-of-the-art and readily available heuristics for path discovery. The A^* search algorithm is such a solution, allowing us to apply the stigmergic approach to arc choice and weighting only, i.e., the *admissible heuristic* function in case of A^*, where each arc has a cost defined by a certain metric.

The cost of choosing an arc a_{ij} is defined as the ratio between a certain property chosen to be used as a metric, h_{ij}, e.g., a distance or

length, and its weight, w_{ij}:

$$c_{ij} = h_{ij}/w_{ij} \tag{4.5}$$

In its turn, the *weight* of the arc w_{ij} is directly correlated to the quantity of pheromones:

$$w_{ij} = \gamma \cdot \tau_{ij} \tag{4.6}$$

where γ is a constant of proportionality (may be typically set to 1) and τ_{ij} represents the *amount of pheromones* placed on arc a_{ij}. In order to make the a-priori cost (i.e., of choosing an arc along the path towards destinations) explicit, let the following formula:

$$c_{i \xrightarrow{j} d} = c_{ij} + \min_{k \in \mathbf{N}_j} c_{j \xrightarrow{k} d} \tag{4.7}$$

define the *cost* $c_{i \xrightarrow{j} d}$ from node i towards destination d along a neighboring node j as the sum of the distance between i and j, c_{ij}, and that from j to destination along the choice of node k, belonging to the neighborhood of j $\mathbf{N}_j^k$, which minimizes this distance.

Given all the above, the value of the *a-priori gain*, $\eta_{i \xrightarrow{j} d}$, for a certain choice leading to destination d is computed according to the following formula:

$$\eta_{i \xrightarrow{j} d} = \delta / c_{i \xrightarrow{j} d} \tag{4.8}$$

where the relationship is inversely proportional with respect to the (weighted) distance, i.e., a cost, and δ is just a constant of proportionality, which may be set to 1 according to literature, when referring to such kind of formula, i.e., tying a-priori information to costs.

A further fix, also applicable to the standard ACO variant, would consist in relaxing the requirement that agents, i.e., vehicles, travel back home after finding food, in its stead leveraging the opportunistic

inter-node communication for near-instant swarm-wide dissemination of pheromone trails.

This way, probability p_{ij}^k of Equation 4.1 has to be adapted to any MoCSACO artificial ant, placed in vertex i, to move toward node j, along the path to destination d, as follows:

$$p^k_{i \xrightarrow{j} d} = \frac{\tau_{ij}^{\alpha} \cdot \eta^{\beta}_{i \xrightarrow{j} d}}{\sum_{l \in \mathbf{N}_i^k} (\tau_{il}^{\alpha} \cdot \eta^{\beta}_{i \xrightarrow{l} d})} \tag{4.9}$$

where τ_{ij} corresponds to the quantity of pheromones laid over arc a_{ij}, $\eta_{i \xrightarrow{j} d}$ to a-priori attractiveness of the choice, computed by some heuristic embedding the cost of choosing arc a_{ij} along the path that leads to the destination d, and $\mathbf{N}_i^k$ is the set of neighbors in node i for ant k, i.e., the admissible transitions for the ant. Even in this case, pheromone gets updated as defined in Equation 4.3.

In MoCSACO, there are as many objectives as destinations, choices are unpredictable (in the sense that an autonomous agent, e.g., a *driver*, may choose to disregard indications, or even just drop off the cooperative efforts by stopping its own instance of MoCSACO), and there cannot be a notion of rounds for such kind of agents.

It follows that there is not an applicable notion of convergence, and pheromone laid over each arc evaporates, still according to Equation 4.4, but on a time basis, by setting per-arc countdown timers (possibly preset to a default value), to be reset at each pheromone update. Moreover, A^* may then be considered a degenerate algorithm (for arc choice only), in the sense that a sparse density of agents, or just slow transitions, for whatsoever reason, may induce depletion of pheromones, which may be counter-acted upon by choosing a very low value for the evaporation parameter and/or the timer frequency. In turn depleted pheromones would lead to significant perturbations in the computation of probabilities, translating into unreliable estimates ultimately, to be accounted for by reverting arc choice to (determin-

istic) A^* computations each time $\sum_{l\ in \mathbf{N}_i^k}(\tau_{il})$ falls below a predefined threshold.

Getting back at Figure 4.2, there are depicted the activities pertaining to a single ant joining the distributed system, i.e., starting up and being connected.

An *Initialization* phase corresponds to downloading initial pheromone matrix P, possibly from a centralized MCS backend, with constantly available information about traffic, if global Internet connectivity is available, or reset to some predefined defaults with respect to the graph topology, such as an inverse proportionality with respect to the length of road segments. Afterwards there are three concurrent (infinite) loops. The rightmost constantly listening for, and collecting, updates (*ReceivePH*) from the mesh. The loop in the center of the diagram, periodically updating the pheromone matrix (*UpdatePHMat*) immediately before every computation of the probability for the neighboring arc(s) to be traversed (*ACOProbComp*), a function which gets triggered by edge visits. In the leftmost (nested) loop, unless the destination has been reached, the system waits for the next destination to be set in order to re-enter the inner loop, where dissemination (*DisseminatePH*) of an updated pheromone value is triggered by timers resetting and thus producing evaporation, or the (arc) *Transit* being over. In its turn, *Transit* is an (agent-performed) action that follows another one about the *Choice* of the arc to be traversed by randomly extracting one of the neighboring arcs according to the aforementioned probability.

4.4 ITS implementation

In this Section we apply the general MoCSACO approach proposed in Section 4.3 to the ITS application domain. In particular, we analyze an instance of a traffic shaping problem implemented through a partially

decentralized navigation system for automotive usage, where both the single user and the (global transportation) system must satisfy a given set of criteria. Such high-level requirement would translate into the unambiguous, lower-level goals of minimizing possibly diverging attributes, thus leading to a tradeoff between the number of vehicles per road segment and the duration of the path of each user.

4.4.1 Motivating example

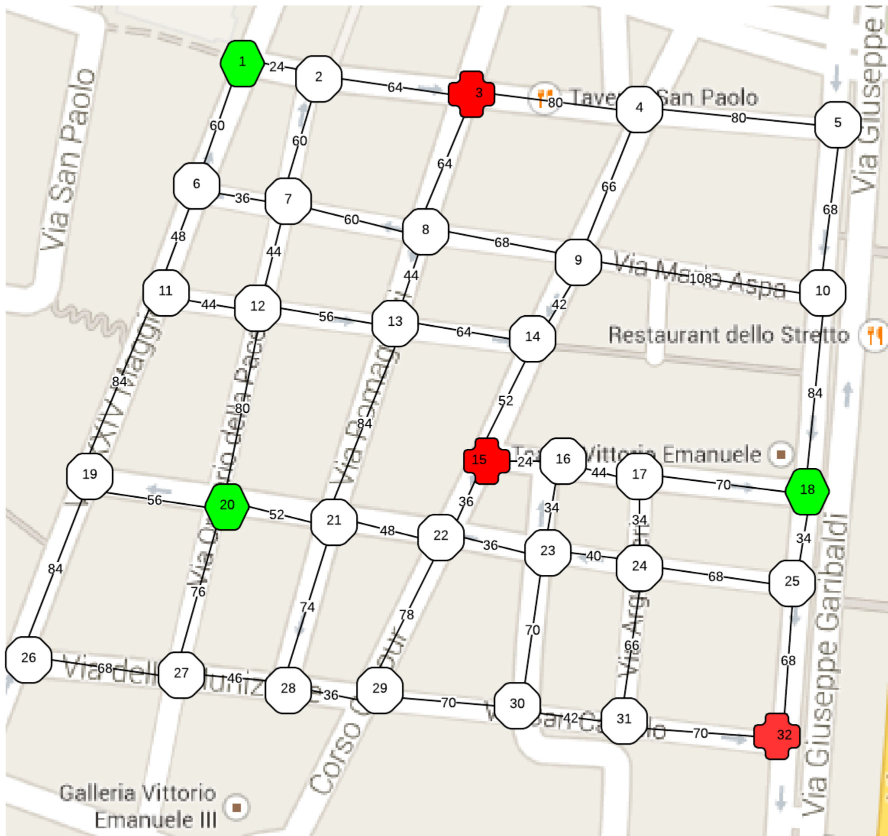


Figure 4.3: *Distance graph: originating road map*

The traffic of an urban area, near downtown the city of Messina, is examined here. From the road map in Figure 4.3, a graph $\mathcal{G}$ with differ-

ent types of nodes on the region of interest (RoI) is derived. Hexagonal vertices denote higher-order nodes, i.e. from these nodes the vehicles enter or leave the MCS cooperative system, e.g., through collector roads or parking lots. The cross-shaped vertices denote destination nodes, i.e., nodes of particular interest that have an high probability to be the final destination of the vehicles, for instance railway stations or schools in the morning. Both higher-order and destination nodes are randomly placed in the RoI. Finally, the circle vertices are just plain transit nodes. Arcs of the graph represent road segments and the corresponding weights represent the length l_{ij} of such segments.

Given such scenario, the objective of the decentralized navigation system is two-fold. On one hand, the single user aims to choose the quickest path to its destination, on the other the system aims as much as possible to preserve a low-intensity traffic in all roads, thus avoiding traffic jams. Notice that in such problem the individualistic solution of choosing the shortest path causes globally a traffic jam in such path. Instead a cooperative solution may be appealing since it allows to decrease the traffic intensity on the RoI, speeding-up the path traversal time for most of the users.

4.4.2 MoCSACO application to ITS

From an ITS perspective, the MCS nodes depicted in Figure 4.1 are embedded automotive devices, e.g., GPS-based navigation systems, installed on each vehicle, or available as detachable devices (e.g., mobiles). Through the mesh MCS network the pheromone-encoded (implicit) information about the traffic in the RoI is disseminated to all vehicles. In such context, due to the opportunistic nature of the communications [100], vehicles may have an approximated and partially incomplete view of the whole traffic situation in the area, albeit an aid could be within reach by planning a number of nodes available as fixed

infrastructure (e.g., totems) at predefined sites, helping at least with dissemination [116, 117, 118] duties by coping with uneven sparsity of agents.

In this sense, apart from the aforementioned application of plain A^* in each vehicle as degenerate function for path building, costs may always be adjusted according to a global view of the traffic provided by a centralized server, as shown by the black arrows in Figure 4.1 connecting a subset of Internet-connected nodes to the Application Server.

Each device-vehicle executes the algorithm of Figure 4.2 acting as an ant, where the matrix updates originate from the continuous exchange of information with other vehicles. The results of such process are the quantities of pheromones laid over the roads (τ_{ij}) and the probabilities to choose such roads given a specific destination ($p_{i \xrightarrow{j} d}$). These values are used to define the costs associated with the arcs that in turn define the heuristic function used by the A^* algorithm executed by each embedded device. In order to implement the proposed strategy in the traffic engineering application, we have to modify Equations (4.6)-(4.8). In particular, we adapt Equation 4.5 to the ITS domain at hand:

$$c_{ij} = l_{ij} \cdot w_{ij} \tag{4.10}$$

by replacing h_{ij} with l_{ij} as the (physical) length of the (road) segment. Being T^k the tour or *traveled path* of ant (i.e., vehicle) k, C^k of Equation 4.2 is redefined as:

$$C^k = \frac{t_e}{T_l^k} \tag{4.11}$$

where t_e is the time spent since the trip beginning, assuming vehicles continuously traveling, and T_l^k is the length of T^k. This way, the amount of pheromones to be placed by ant k on each arc is still

specified by Equation 4.2.

A further adjustment is needed to achieve a good trade-off between the objective of each vehicle, i.e., reach its destination in the minimum time, and the one of the transportation system as a whole, i.e., preserving low traffic intensity. A solution may come from literature, where the authors of [119] specifically tailored the algorithm to traffic routing, a "modified ACO". This version of the algorithm, geared towards traffic routing, consists of a straightforward adaptation of traditional ACOs. A probability threshold, t, is introduced to modify arc traversing probabilities as given by the Equation 4.1, to make arcs less desirable for ants. This threshold may be defined as function $t : [0, 1] \rightarrow [0, 1]$:

$$t(p_{ij}^k) = \frac{1 - p_{ij}^k}{2} \tag{4.12}$$

where p_{ij}^k is the probability as originally defined for ant k choosing arc a_{ij}. The relation for arc traversing probability is adapted by the presence of a threshold as follows:

$$\overline{p}_{ij}^k = \left\{ \begin{array}{lr} p_{ij}^k & \text{if } p_{ij}^k < \text{threshold } t \\ t(p_{ij}^k) & \text{otherwise} \end{array} \right\} \tag{4.13}$$

The threshold defined above is sensitive to both collective knowledge stored in the pheromone matrix as well as a-priori information. This property is important especially when a-priori information varies in time as traffic congestion does. Indeed, even using such a simplified thresholding function, the modified algorithm quickly reacts to changes in the environment, while defaulting to neglecting a good routing candidate when the path is very crowded, as soon as an efficient path becomes less crowded (i.e., the probability for an ant to choose it falls below t), the logic will switch back to privileging such path for routing. For such reasons a threshold-enabled ACO seems an ideal candidate to solve multi-path traffic routing problems.

Using Equations 4.7-4.5 we can compute for each destination d the cost $\eta_{i \xrightarrow{j} d}$ to take the arc (i, j) in the path towards node d without considering the road traffic. Providing these values as a metric to define the heuristic function of the A^* search algorithm, we can find the minimum length path, which can be very different from the optimal path considering the traffic.

4.5 Modeling and evaluation

To evaluate the MoCSACO approach a specific technique able to stochastically represent the ant colony interactions is required. The usual way to study ACO problems is through simulation, since classical analytical techniques, such as state space-based ones, are affected by the well-known state space explosion problem, due to the large number of involved elements (i.e., ants and roads). However, new stochastic entities, called Markovian Agents (*MAs*) [120], have been introduced to provide a flexible, powerful, and scalable technique for modeling complex systems of distributed interacting objects thus evaluated through feasible analytical and numerical solution algorithms. Moreover, *MAs* are suitable to represent systems able to self-organize their topology adapting to environmental changes [121] such as ant colonies.

4.5.1 MA model of the MoCSACO algorithm

An *MA* is an entity that can evolve autonomously according to its local behavior, but interacts with the environment and with the other agents. In particular, an *MA* is a finite-state continuous-time homogeneous Markov chain (CTMC) that evolves according to a given transition rate matrix and is located in a specific geographical position. The interaction among *MAs* is represented by the exchange of relational

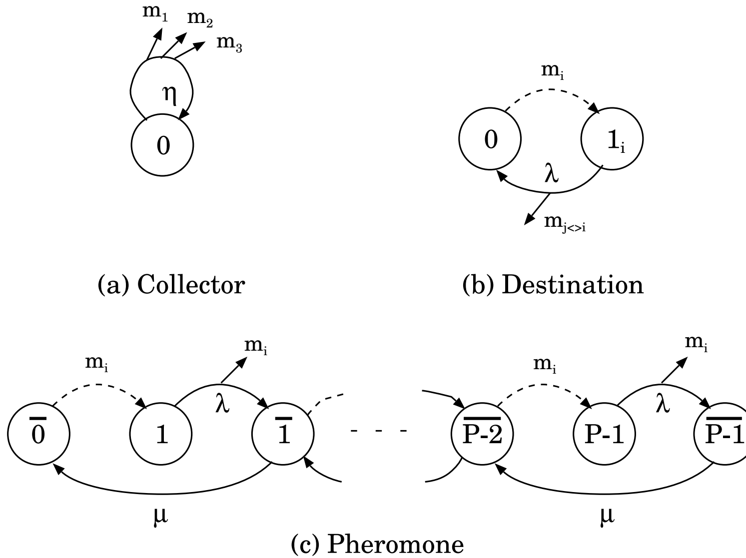


Figure 4.4: *Markovian Agents: categories*

entities, called messages, which are emitted by an *MA* and perceived by its neighbors influencing their dynamics. The specific interactions among agents are formalized through a *perception function* that rules the aptitude of receiving messages according to agent positions. To model heterogeneous systems, different classes of agents and type of messages are allowed: agents belonging to the same class behave in the same way (i.e., they have the same CMTC structure, but possibly with different rates), and when receiving a message they may react in different ways according to the type of message received.

In the model for the example in this Section the *MAs* of different classes are placed on the vertices of the graph $\mathcal{G}$ according to the type of node. A class h agent is placed in collector nodes, a class d is placed in destination nodes and a class p agent is positioned in the others. A vehicle, or ant, moving from vertex i to j is represented by a message emitted by an *MA* located in vertex i, and received by an *MA* in vertex j. Three different types of messages $\{m_1, m_2, m_3\}$ are required to represent the three possible vehicle destinations.

The behavior of an agent collector of class h located in vertex $\mathbf{v}$ (from now on called $MA^h(\mathbf{v})$) is shown in Figure 4.4(a). It is characterized by a single state with a self loop which rate of incoming vehicles is η. During its transitions it can emit with equal probability messages of type m_i (in Figure 4.4 they are shown as labeled little arrows starting from the self loop) representing the incoming of a vehicle with a probabilistic destination. Agent MA^d is depicted in Figure 4.4(b): in state 0 the agent waits for the arrival of messages; when a message m_i arrives, the agent moves to state 1_i (depicted as a dashed arrow in the figure) and then comes back to 0 retransmitting a new message $m_j \neq m_i$. This represents that, when a vehicles reaches its final destination, it decides to change its destination moving towards other nodes. Assuming that the mean time to traverse a node is equal to Tr, we can set $\lambda = 1/Tr$. Finally, the MA^p agent encodes in its state-space the amount of pheromone in the node. In our model, such value is discretized in P levels ranging from 0 to a maximum amount of $P-1$ units. Thus, state 0 of the MA^p agent represents a node without pheromone, whereas states $\overline{p}$ or p mean the presence of p units of pheromone. At the arrival of a vehicle with destination i (dashed arrow labeled m_i), the amount of pheromone gets increased by one unit, when the vehicle leaves the node moving to a neighbor node (continuous arrow with generation of message m_i) the amount of pheromone is preserved. Pheromone evaporation is represented by a local transition from a state $\overline{p}$ to a state $\overline{p-1}$, thus pheromone decrements by one unit at time with rate μ.

Let us denote the total density of agents of class c in position $\mathbf{v}$ with $\xi^c(\mathbf{v})$ and $\rho_i^c(t, \mathbf{v})$ the density of agents in state i and position $\mathbf{v}$ at time t. The state densities are collected into a vector $\boldsymbol{\rho}^c(t, \mathbf{v}) = [\rho_i^c(t, \mathbf{v})]$.

The routing of messages exchanged by MA^p agents is ruled by the perception function $u_m(\cdot)$ defined similarly to Equation 4.9. For each

destination m_i we have:

$$u_m(\mathbf{v},\mathbf{v}',t) = \frac{(E[\boldsymbol{\rho}^p(t,\mathbf{v})])^\alpha \cdot \eta^\beta_{\mathbf{v}'\overset{\mathbf{v}}{\to}m}}{\sum_{\mathbf{v}''\in Next(\mathbf{v}')}(E[\boldsymbol{\rho}^p(t,\mathbf{v}'')])^\alpha \eta^\beta_{\mathbf{v}'\overset{\mathbf{v}''}{\to}m}} \tag{4.14}$$

where the average amount of pheromone $E[\boldsymbol{\rho}^p(\tau,\mathbf{v})]$ is computed considering the pheromone level corresponding to the states of the agents MA^p. The routing probabilities among other agent classes (i.e. MA^h,MA^d) are computed in a similar way. Simple corrections on the definition of $u_m(\cdot)$ allow to include also the threshold variant introduced by Equations 4.12-4.13.

The evolution of the entire model can be studied by solving $\forall\mathbf{v},c$ the following differential equations:

$$\boldsymbol{\rho}^c(0,\mathbf{v}) = \xi^c(\mathbf{v})\boldsymbol{\pi}^c_0 \tag{4.15}$$

$$\frac{d\boldsymbol{\rho}^c(t,\mathbf{v})}{dt} = \boldsymbol{\rho}^c(t,\mathbf{v})\mathbf{K}^c(t,\mathbf{v}). \tag{4.16}$$

where $\boldsymbol{\pi}^c_0$ is the initial probability distribution vector of a class c agent and $\mathbf{K}^c(t,\mathbf{v})$ are the time-dependent infinitesimal generator matrices ruling the whole behavior of agent of class c in position $\mathbf{v}$. Equation 4.15 and Equation 4.16 are discretized in time and solved by resorting to standard numerical techniques for differential equations. Details on both the $\mathbf{K}^c(t,\mathbf{v})$ matrices' computation and the solution technique can be found in [120].

4.5.2 Results

The *MA* model of the MoCSACO traffic engineering example has been evaluated to provide useful insights on its effectiveness in reducing the average traffic within the urban area. Two scenarios are investigated: the first, called *NoPh*, assumes that vehicles choose roads along the

minimum-length path to their destination, ignoring any traffic information encoded by the pheromone. In the second, identified as *Ph*, vehicles still follow their minimum-length path, but also try to avoid high traffic roads where a strong trail of pheromone is present. The comparison between the results obtained in the two scenarios allow to evaluate the effectiveness of MoCSACO to reduce the average traffic.

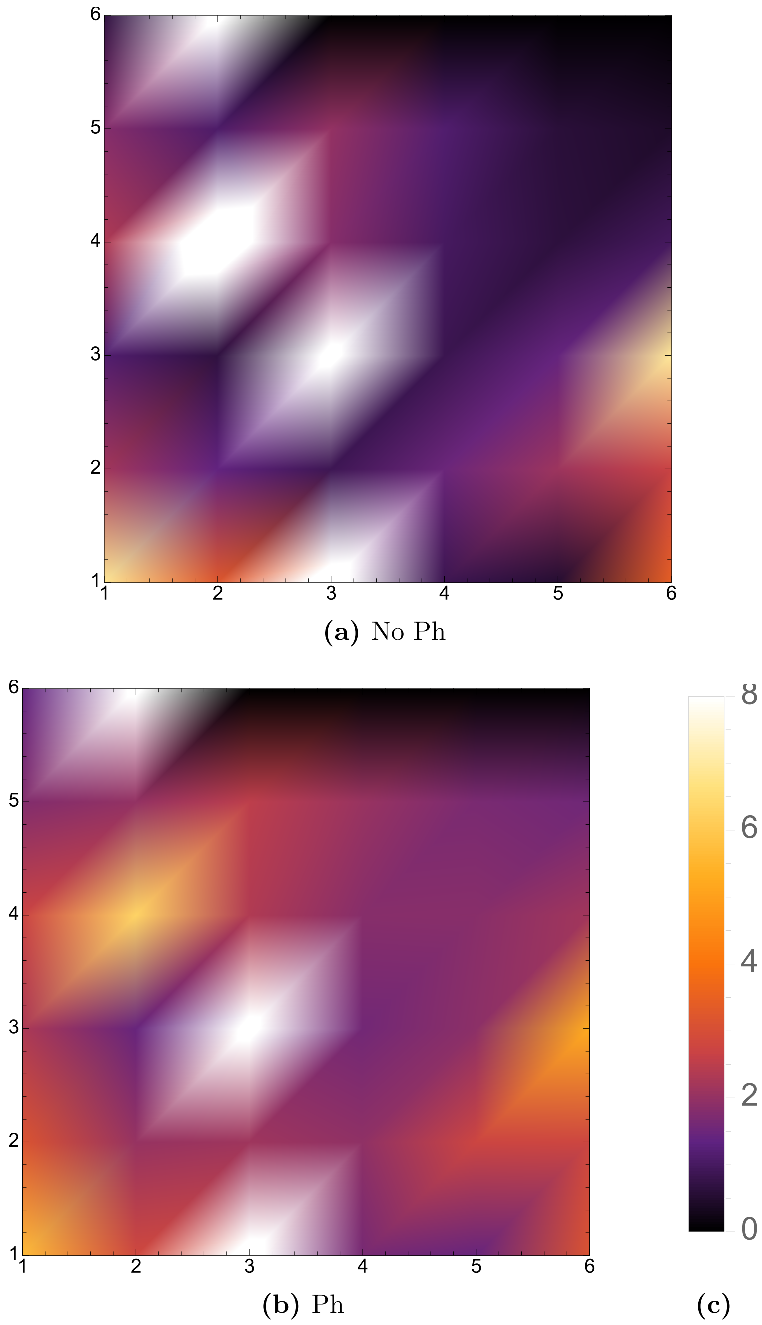
In such a complex and dynamic behavior, we first need to identify an instant in time that well represents the average (or, i.e., steady-state) condition for the system. The equations described in Section 4.5.2 provide the transient analysis of the model. The $MA^p(\mathbf{v})$ is said to be in a stable state when its average pheromone intensity does not vary anymore. Let $t_s(\mathbf{v})$ be the first time instant at which the $MA^p(\mathbf{v})$ is stable. Since the overall system reaches the stability when all the nodes are in a stable state, the time for stability $\widetilde{t}$ can be defined as:

$$\widetilde{t} = \max_{\mathbf{v} \in V(\mathcal{G})} t_s(\mathbf{v}). \tag{4.17}$$

where $V(\mathcal{G})$ is the set of vertices of the graph $\mathcal{G}$. Further details on the procedure to compute the time stability of $MA^p(\mathbf{v})$ can be found in [120]. To evaluate the average behavior of the system, from now on all the results will be computed in the stable state, i.e., at time $t = \widetilde{t}$.

Since a possible traffic intensity measure is the number of vehicles traversing a road in a unit of time and, in the model, messages exchanges represent vehicle movements, the global rate of messages traversing an arc is a proper metric for the traffic intensity of a road. The rate $\gamma(\widetilde{t}, \mathbf{v}', \mathbf{v})$ of the whole traffic in the direct arc $(\mathbf{v}, \mathbf{v}')$ of the graph can be obtained as the sum of the rate for all messages emitted by any agent classes from $\mathbf{v}$ to $\mathbf{v}'$:

$$\gamma(\widetilde{t}, \mathbf{v}', \mathbf{v}) = \sum_{m} \sum_{c=1}^{C} u_m(\mathbf{v}', \mathbf{v}, \widetilde{t}) \phi^c(m) \rho^c(\widetilde{t}, \mathbf{v}). \tag{4.18}$$



(a) No Ph

(b) Ph

(c)

Figure 4.5: *Results: pheromone distributions*

where, for a given message m and a class-c agent, the rate can be computed as the product of the density of class-c agents ρ^c that generate message of type m and the corresponding generation rate $\phi^c(m)$ modulated by the perception function. Further details can be found in [120].

The results of the model are analyzed for the graph shown in Figure 4.3 and with destination vertices (MA^d) in position $\{3, 15, 32\}$, collector vertices (MA^h) in position $\{1, 20, 18\}$. In the evaluation we set the following parameters: $P = 16$, $\lambda = 10$, $\mu = 2$, $\eta = 5$. Moreover, the vehicle behavior in the *NoPh* scenario can be obtained by setting $\alpha = 0$ in Equation (4.14), so that pheromone values do not contribute to $u_m(\cdot)$. In *Ph* scenario the choice of an high-pheromone node is discouraged by setting $\alpha = -1$ (as can be inferred by Equation 4.14). In both scenarios $\beta = 0.5$, so that the values of $E[\boldsymbol{\rho}^p(t, \mathbf{v})]$ and $\eta^{\beta}_{\mathbf{v}' \xrightarrow{\mathbf{v}} m}$ have the same magnitude thus setting a fair trade-off between choosing the minimum path and avoiding high traffic roads.

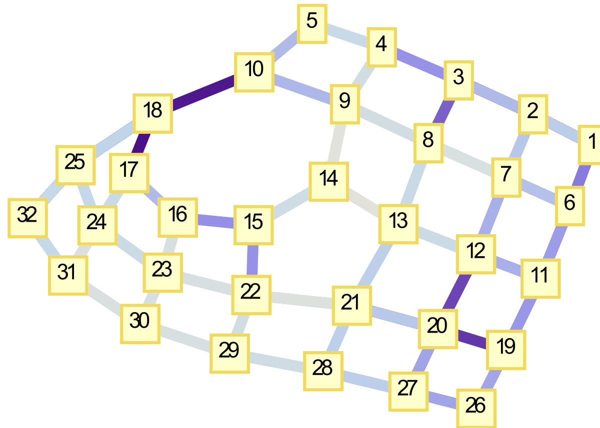
Figure 4.5 shows the pheromone distribution over the nodes of the example graph as a density plot. Ignoring the graph topology, nodes are arbitrarily arranged in a square grid and colored according to their pheromone intensity in the stable state, i.e. the value of $E[\boldsymbol{\rho}^p(\widetilde{t}, \mathbf{v})]$. Dark areas correspond to a low pheromone intensity, lighter ones to high intensity. In the *NoPh* scenario a very congested node can be detected in Figure 4.5(a) at position (2,4), instead in the *Ph* scenario the traffic intensity of the same node strongly decreases. A slightly reduction of traffic can be observed also for the nodes in position (1,1) and (6,3). To be noted also that in the same time the traffic of several low-used nodes in the *NoPh* scenario increase in *Ph* one, meaning that, to avoid congested nodes, vehicles are redistributed along other directions.

A quantitative analysis of the pheromone distribution also concurs to confirm the previous observations. In fact, evaluating the mean

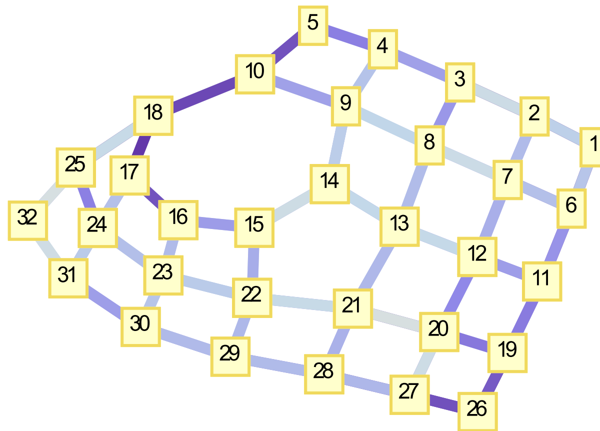
value μ of pheromone intensity over the set of vertices of graph $\mathcal{G}$ and its coefficient of variation c_v in both scenarios, we obtain $\mu^{NoPh} = 2.463$ and $c_v^{NoPh} = 1.195$, and $\mu^{Ph} = 2.734$ and $c_v^{Ph} = 0.837$, for *NoPh* and *Ph* scenario, respectively. A comparably greater value for the coefficient of variation in the *NoPh* scenario supports the existence of very congested nodes, whereas a greater mean value in the *Ph* scenario suggests that traffic is more evenly distributed.

The traffic within the urban area in a given road is evaluated by summing up the flow rate $\gamma(\widetilde{t}, \mathbf{v}', \mathbf{v})$ between node $\mathbf{v}$ and $\mathbf{v}'$ in both directions. In Figure 4.6 two isomorphic graphs of Figure 4.3 are shown, depicting the *NoPh* scenario on the left, and the *Ph* one on the right. The arcs are colored according to their flow rate: high traffic roads are dark blue, whereas less used ones feature lighter hues. In the *NoPh* scenario there are some very congested roads, in particular at branches 17-18-10 and 12-20-19. The introduction of MoCSACO allows to decrease the traffic of such roads, as shown by the *Ph* scenario depicted in Figure 4.6(b), at the cost of slightly increasing the traffic at 27-26-19.

As above, we can analyze the mean value and the coefficient of variation of the flow rate over the set of arcs of graph $\mathcal{G}$ obtaining $\mu^{NoPh} = 0.045$ and $c_v^{NoPh} = 4$, and $\mu^{Ph} = 0.055$ and $c_v^{Ph} = 3.636$, for the *NoPh* and the *Ph* scenario, respectively. Analogously to the pheromone intensity, also the traffic flow is more fairly distributed in the *Ph* scenario than in the *NoPh* one. From these results we can argue that the introduction of MoCSACO allows the overall traffic to be more evenly spread out over the urban area.



(a) No Ph



(b) Ph

Figure 4.6: *Results: traffic flow intensities*

CHAPTER

FIVE

NETWORK FUNCTION VIRTUALIZATION FOR CYBER-PHYSICAL SYSTEMS

5.1 Introduction

In a typical Infrastructure-as-a-Service (IaaS) Cloud, users are able to create and bring up virtual machines (VM), access the instances through ssh, VNC, or Web-based virtual console as well as to instantiate even topologically complex virtual networks among a set of VMs.

In a heavily distributed ecosystem, such as IoT-related scenarios, many requirements diverge significantly in comparison to typical IaaS Cloud environments, such as the presence of nodes installed behind firewalls and/or NATs (especially when IPv6 deployments are not an option) or, more in general, the necessity to deal with any restricted environment with denied-by-default (institutional or corporate) security policies.

Such constraints call for more powerful mechanisms to enable core functionalities for virtual infrastructure management, i.e., remote access to board-hosted resources and instantiation of virtual networks.

Any network virtualization [122],[123] mechanism for IoT infras-

tructure thus requires at least some form of reconfiguration capabilities for board-side networking facilities as well. Yet, in contrast to typically datacenter-oriented IaaS, the physical environment (cabling and media access setup, logical topologies and hierarchies, role allocation for equipment, etc.) in IoT scenarios is not always under control of the designer of the infrastructure, which may as well be opportunistically assembled, e.g., volunteer-contributed. Nor is the configuration of most deployments (or their extent, ownership, etc.) completely known in advance usually, in contrast to typical setups for certain specialized categories, e.g., Wireless Sensor Networks (WSN), where some advances [124] in terms of network virtualization have been accomplished.

As such, IoT poses unique challenges, including always-on reachability of boards, or at least suitable signaling, diagnostic and recovery mechanisms to cope with connectivity disruptions. Even more critical, the overall approach to virtualization may be considered reversed with respect to availability of remote access, as the latter alone may enable the former, thus networking primitives have to be piggybacked into the remoting framework.

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 Cloud-managed IoT resources in a technology agnostic fashion, still taking into account the limitations of smart devices, while at the same time suitable to be mapped onto an IaaS-focused solution, as investigated in [1] in terms of a device-centric approach for sensor-hosting nodes.

The contribution here is thus three-fold: a Cloud-based framework for the setup of virtual networks among IoT nodes, whichever the deployment scenario; a customizable and layered tunneling protocol; a

flexible and lightweight network virtualization solution, based on universally available and minimal tools, according to the Unix philosophy of composability and modular design.

5.2 Network virtualization for IoT

The proposed approach to network virtualization is based on enabling mechanisms in terms of custom layering and board-side tunneling facilities, to be coupled with the corresponding Cloud-side adaptations. To this end such preliminary investigation mostly focuses on virtualization architecture and patterns.

5.2.1 Tunneling

As remote infrastructure, boards are possibly going to be available over very restrictive, IPv4-only deployments. The only assumption that can (for all purposes, always) be considered true is outgoing Web traffic being permitted, i.e., board-initiated communication over standard HTTP/HTTPS ports. The aforementioned constraints thus suggest resorting to an HTTP-borne mechanism for bidirectional connectivity and reachability of internal services, namely WS.

WebSockets [38] as channels between a browser and a server are considered standard facilities for bidirectional communication and in particular server-pushed messaging. The main rationale behind its design lies in the need to replace the long-polling and Asynchronous JavaScript and XML (AJAX) approaches. In accordance to typical use cases targeted by these older mechanisms, WS enables the server to push unsolicited content to the browser without waiting for a request. Messages can thus be passed back and forth while keeping the connection open creating a two-way (bi-directional) ongoing conversation between a browser and the server. One of the main advantages

of WS is that it is network agnostic, by just piggybacking communication onto standard HTTP interactions. This is of benefit for those environments which block Web-unrelated traffic using firewalls. Less explored is the creation of generic TCP tunnels over WS, a way to get client-initiated connectivity to any server-side local (or remote) service.

In this chapter a design and implementation of a novel *reverse* tunneling technique has been devised, as a way to provide server-initiated, e.g., Cloud-triggered, connectivity to any board-hosted service, or any other node on a contributed resource network, e.g., a WSN. In particular the latter may enable typical IoT scenarios, e.g., Machine-to-Machine (M2M) interactions, by supporting these patterns in a device-centric [1] fashion: having a gateway act not only as a proxy for access to data gathered from mostly passive resources, but also as a relay to activate remoting toward nodes in a masqueraded network.

Figure 5.1 depicts systems, flows and interactions of such a WS reverse tunnel (abbreviated as *rtunnel*), in the case of board-provided access to a service hosted on the board itself. By leveraging the diagram, in the following the sequence of operations is outlined for the setup of a rtunnel. The rtunnel client (e.g., a board) first sends a WS connection request to the rtunnel server, specifying a TCP port. When the rtunnel server receives the WS connection request, a new TCP server is brought up listening on the specified port, the WS request is then accepted, and a WS connection (depicted in the figure as “control WS”) is started. When an external TCP client tries to connect to the TCP server on the rtunnel server side, the new TCP connection is paused and, through the control WS, a WS message is sent in order to signal the request for a new TCP connection, and specify a unique ID for that connection. When the rtunnel client receives the message signaling the request for a new TCP connection, it sends a new WS connection request to the rtunnel server, specifying

the ID of the connection. When the rtunnel server receives the WS connection request, it checks if the received ID does not match any of the existing TCP connections and, if so, it accepts the request and opens a new WS connection (depicted in figure as "WS tunnel"). The new TCP connection thus gets piped to the new WS connection (that acts as a WS-encapsulated tunnel for TCP segments) and then resumed. On the WS rtunnel client side, as soon as the new WS tunnel is established, a new TCP client is brought up connecting to the local service of interest, and such a new TCP connection gets piped to the new WS tunnel. TCP segments coming from the external TCP client are now able to reach the local service, and traffic thus gets to flow back and forth until the rtunnel is torn down.

5.2.2 Layering

As long as WS-based tunnels may be instantiated by the Cloud, a robust mechanism is already in place for accessing board-hosted services. What is missing to bridge the gap between remoting only and level-agnostic network virtualization are mechanisms to overlay network- and datalink-level addressing and traffic forwarding on top of such a facility. There are already solutions [125] for setting up VPNs on top of WS, but without decoupled control machinery nor the inherent flexibility of an on-demand mechanism.

A detailed description of the proposed layering for WS tunnel-based layer-2 virtual networks follows. In Figure 5.2 a diagram is modeled after the low-level reverse tunnel one, but focused on the instantiation of, e.g., a virtual bridge between two boards. Still sticking to the setup of a *control* WS, as a preliminary step in this workflow, in this case a rtunnel gets activated for each board to be virtually bridged. As a simplified scenario, the diagram depicts just two such boards, but no limitation is in place on the number of remote boards

to be virtualized in terms of networking. As any board here, from now on referred as client, needs to go through the same set of operations, just a single instance will be described in full, for the sake of brevity.

Taking into consideration the uppermost board in the diagram, a preliminary step lies in setting up a TCP connection based on a WS-based rtunnel, which consists in 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.

A level-3 tunnel is then to be established over this TCP-based tunnel, by employing an instance of an executable called Socat, which operates in listening mode on both sides of the chain and, on connection, starts exposing a virtual (TUN) device on either side, both set up with IP addresses of choice, as long as those belong to the same subnet.

Speaking about virtual devices setup and binding, *Socat* is a networking "swiss army knife" available as command-line tool for Unix systems. Similarly to its close counterpart, Netcat (nc), the more full-featured Socat brings a host of functionalities and quick shortcuts to network experimenters, such as socket piping and tuning, setup of virtual (TUN/TAP) devices, process control, and more. The minimal build-time dependencies (just the C library) translate into a significantly flexible tooling also when it comes to IoT-class, constrained devices, as long as a stripped-down version of a POSIX-compatible system and the relevant networking stack are available.

Even if the above reported steps are to be considered logically operations to be performed early on, all steps are to be considered

timing insensitive, by employing retries and listening sockets where needed, possibly recurring to the TCP *gender-changer* technique when both ends of the pipe are required to be in listening mode.

In order to then set up a level-2 encapsulation over the aforementioned IP-based communication, the system has to bring up a GRE tunnel, where the endpoints are the previously configured TUN IPs and the type of tunnel-hosting virtual device is set to TAP, thus exposing an Ethernet-compatible interface. Adding such interface to a dedicated virtual bridge on the server concludes the workflow.

As one of the technologies needed is IP-based tunneling, the choice has fallen on *Generic Routing Encapsulation (GRE)* [126], an IETF standard for a no-frills IP-in-IP tunneling protocol. Indeed, GRE support is not limited to level-3 encapsulation, but also available for tunneling of level-2 (Ethernet) frames over to the corresponding virtual (TAP) device.

By the kind of *reversed* layering here devised, it is thus possible to provide an initial, basic but effective mechanism for instantiation of a virtual network that exposes boards as either placed in the same broadcast domain, just routable or alternatively reachable higher up in the networking stack. Ultimately this means being able to set up, according to user needs, either a virtual bridge, i.e., same level-2 broadcast domain, by means of GRE TAP-based tunnels, virtual NICs, socat piping, and reverse tunneling over WS, or a virtual private network, i.e., level-3 reachability, by leveraging just a subset of the aforementioned mechanisms, plus static routes configured on the server for board-to-board forwarding.

5.3 A real-world example

As anticipated, use cases which may reasonably be envisioned, for IaaS-mediated functionalities exposed by IoT boards, are mostly about

board-hosted resources, such as sensors and actuators, being available over the Web as services, as well as board-related remote access facilities, modeled after the ones users typically expect for standard VMs, such as ssh or VNC. Such connectivity capabilities are enabled by runtime-instantiated ad-hoc tunnels exposing services running on the boards, advertised as available, and requested by user through the Cloud portal.

Even before getting remote access to the boards, for instance for deploying an application, the user may arrange a certain topology among boards by network virtualization, in order to accomodate the requirements of the application itself. In particular, an interesting case is that of the AllJoyn framework, which comprises a DBus-derived application protocol useful for messaging, advertisement and discovery of services, working via selected mechanisms on available transports. A very simple actuator-driving application has been designed, e.g., switching a bell on and off, triggered upon reaching a certain threshold, for measurements by sensors sampling certain phenomenona on another board, in this case light intensity through a photodiode. The distributed system works by letting these two boards interact through AllJoyn over an IP-based network and the corresponding transport implementation, where mDNS and a combination of multicast and broadcast UDP packets are used. A limitation indeed is that the protocol is currently designed to work only as long as the communicating boards are on the same broadcast domain. Therefore, such a case may be tackled by leveraging the Cloud to instantiate a bridged network among the two boards, coupled with the availability of remote access for deployment and execution of the required binaries.

To streamline the description of the use case, the nodes are assumed to be already registered to the Cloud.

A high-level description of the workflow, from the point of view of the user, comprises the following steps:

- Request for a bridge between two managed boards.
- Request for exposing SSH service on both boards.
- Connect via SSH to both boards for deploying and launching the AllJoyn applications.

The following list of sequences is then expected to take place, with (low-level) interactions as depicted and numbered in Figure 5.3.

1) The user requests the setup of a bridge between two specific boards, either through the dashboard or, in alternative, through the command line client.

2) The dashboard performs one of the available IoTronic APIs calls via REST, which pushes a new message into a specific AMQP queue.

3) The conductor pulls the message from the AMQP queue and correspondingly performs a query on the IoTronic database. In particular, it checks if the board is already registered to the Cloud and looks up the WAMP agent to which the board is registered. At last, it decides the WS tunnel agent to which the user can be redirected and randomly generates a free TCP port.

4) The conductor pushes a new message into a specific AMQP IoTronic queue.

5) 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.

6) Through the WAMP lib the lightning-rod engine receives the message by the WAMP router.

7) The lightning-rod engine sets up a rtunnel with the WS tunnel agent specified by the conductor, also providing the TCP port through the wstunnel lib. It also brings up a number of sockets to be piped and overlaid over the rtunnel, plus the corresponding virtual interfaces, as described in Section 5.2.2.

8) The WS tunnel agent follows up with its own set of server-side network virtualization duties, still according to Section 5.2.2. Then, it publishes a new message into a specific AMQP queue confirming that the operation has been correctly executed.

9) The IoTronic APIs call pulls the message from the AMQP queue and replies to the s4t dashboard.

10) The user gets notified of the success of the operation.

This first sequence has to be replicated for both nodes, as well as the following two.

In order not to stretch the description, here only phases which are different from the previous use case are outlined. In particular, the second sequence (remote access) steps 2-6,9 remain unchanged, step 1,7-8,10 are changed as follows:

1) The user asks for a connection to the SSH service local to a specific board, either through the s4t dashboard or, in alternative, through the s4t command line client.

7) The lightning-rod engine sets up a rtunnel with the WS tunnel agent specified by the conductor, also providing the TCP port through the wstunnel lib. It also opens a TCP connection to the internal SSH daemon and pipes the socket to the tunnel.

8) The WS tunnel agent brings up a TCP server on the specified port, and then publishes a new message into a specific AMQP queue confirming that the operation has been correctly executed.

10) The dashboard provides the user with the IP address and TCP port that she can use to connect to the SSH daemon running on the board.

And an additional step is present:

11) As the user employs an SSH client to connect to the specified IP address and TCP port, the session is tunneled right to the board.

In the following some theoretical considerations will be laid out, such as packet size and overhead estimates, that can be considered a preliminary analysis, to be conducted more extensively in a future work, taking into account specific key performance indices. More in detail, WS introduces 6 bytes of overhead (2 for the header and 4 for the mask value) and TCP tunneling 20 bytes in the best case. GRE-based encapsulation takes up additional 8 bytes, whereas Ethernet framing amounts to 18 bytes, adding up to 52 bytes. Switching to TLS for security, as for Secure WS, 41 bytes have to be added, thus totaling 93 bytes per packet overhead. It can be seen then that, with regard to the various encapsulations, the effects on the size of packets are still in line with those imposed by a typical VPN, such as OpenVPN, roughly weighting 69 bytes per packet (41 security and 28 tunneling overheads, respectively), i.e., only slightly smaller compared to this solution.

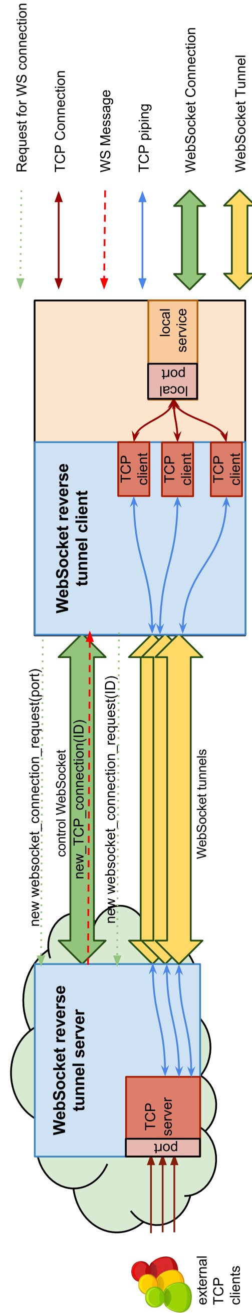


Figure 5.1: *Functional diagram of WS-based reverse tunneling*

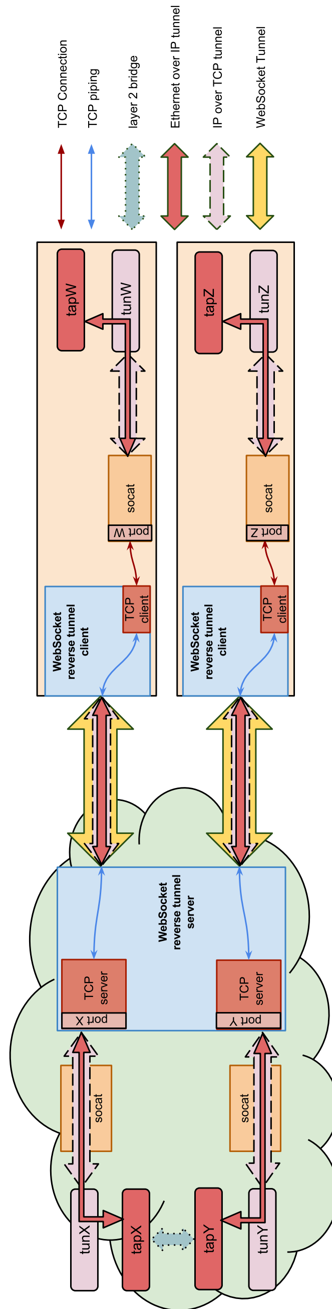


Figure 5.2: *Functional diagram of tunnel-based bridging over WS*

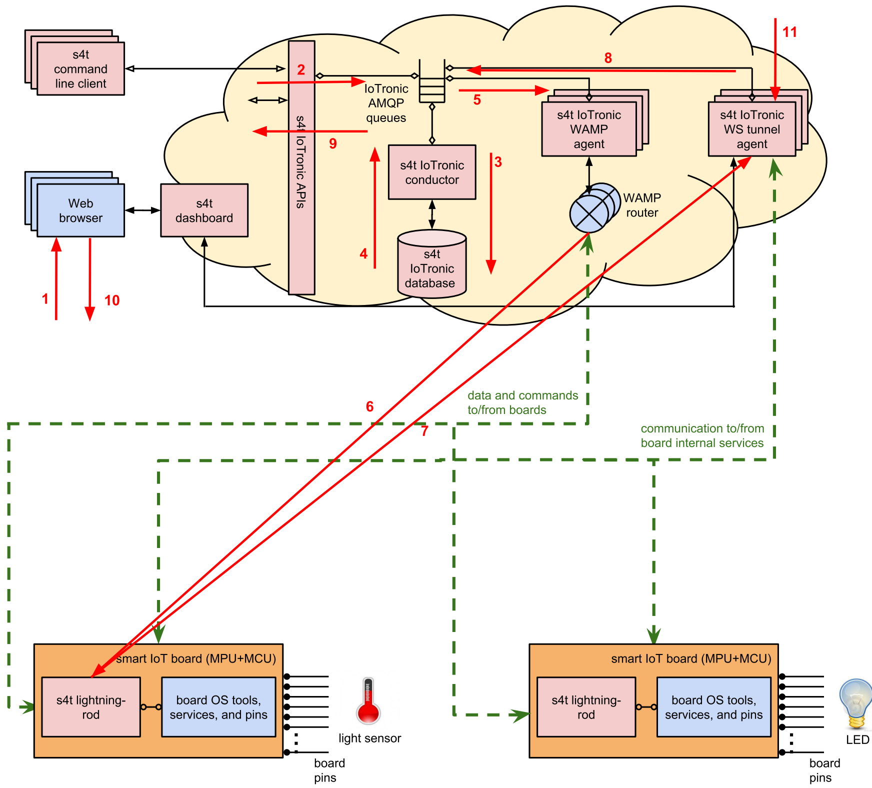


Figure 5.3: Virtual networking use case: workflow

CHAPTER

SIX

SOFTWARE-DEFINED CITY: AN ELASTIC MODEL FOR THE SMART CITY

6.1 Introduction

The Smart City scenario is a fertile application domain for different sciences and technologies, in particular for the information and communication ones, as also confirmed by ongoing projects [127] highlighting the need to equip cities with Cyber-Physical subsystems in a Smart City fashion. From this perspective, cities may be regarded as complex "ecosystems" composed of heterogeneous interconnected "things" providing sensing and actuating facilities, such as traffic sensors, security cameras, traffic lights as well as citizens' smartphones.

However, from a higher level and more urban-focused perspective, specific facilities for management, organization, and coordination of devices, sensors, objects and things are also required to build up a dynamic Smart City infrastructure. To this purpose, on the one hand the capabilities provided by existing solutions in the management of distributed systems, ensuring flexibility and dealing with the complexity of large scale systems, should be exploited to implement basic

mechanisms and tools for the resource management, also taking into account IoT solutions. On the other hand, it is necessary to provide and implement advanced solutions and policies able to manage and control the Smart City infrastructure, implementing strategies aiming at satisfying higher (applications and end users) requirements, on top of basic facilities provided at a lower level. This two-layer model recalls the *Software Defined Ecosystem* model, where the data plane provides basic, customizable functionalities and the control plane implements advanced mechanisms and policies to control the ecosystem by enforcing strategies on nodes and objects through the lower level basic mechanisms. Thus, the main idea proposed in this chapter is to treat a Smart City as a Software Defined Ecosystem, adopting a two-layer Software Defined model to manage the underlying infrastructure towards *Software Defined Cities* (SDC).

To implement the SDC concept, Cloud computing facilities, applying a service-oriented approach in the provisioning and management of resources, may be exploited. The Cloud-based SDC approach could be a good solution to address Smart City-related issues, fitting with the requirements of relevant service users and application providers: on-demand, elastic and QoS-guaranteed, to name a few, all needed properties for a Smart City service platform, to be addressed mainly at the SDC control plane.

The contribution of this chapter can be summarized as fourfold: a conceptual framework for function virtualization of Cyber-Physical Systems and the modeling of a Smart City as a Software Defined subsystem; a requirement analysis for an enhanced IaaS framework able to include and provide urban facilities as reconfigurable and complex CPS; an architecture of node-side modules and the corresponding mechanisms needed to empower City-scale virtualization of sensing and networking functions; an emergency management scenario coupled with two related use cases, respectively about automatic reaction

to situations of risk and seamless exploitation of field deployments.

6.2 Related work

Several works deal with infrastructure issues and solutions related to Smart Cities and their relationship with IoT and Cloud. For example, a platform for managing urban services that include convenience, health, safety, and comfort is proposed in [128, 129, 130], in the latter two cases based on Cloud computing infrastructure. Taking into account other experiences, such as the earliest experiments around Smart City planning, e.g., projects like SmartSantander [34], typically most efforts revolve around managing heterogeneous devices, usually by resorting to legacy protocols and vertical solutions out of necessity, and integrating the whole ecosystem by means of an ad-hoc solution. The intuition here is that the Software Defined approach coupled with service-oriented Cloud-enabled frameworks may play a role as a paradigm for IoT, at the same time leveraging to a great extent one or more ready-made solutions for the infrastructure management, to be adapted and extended to IoT. Indeed, even if a lot of applications in a Smart City scenario have been proposed so far, there is a lack of common initiatives and strategies to address most issues at an infrastructural level in a comprehensive way, nor is any of these efforts geared towards establishing a more general framework, i.e., one which is geared toward Software Defined control loops.

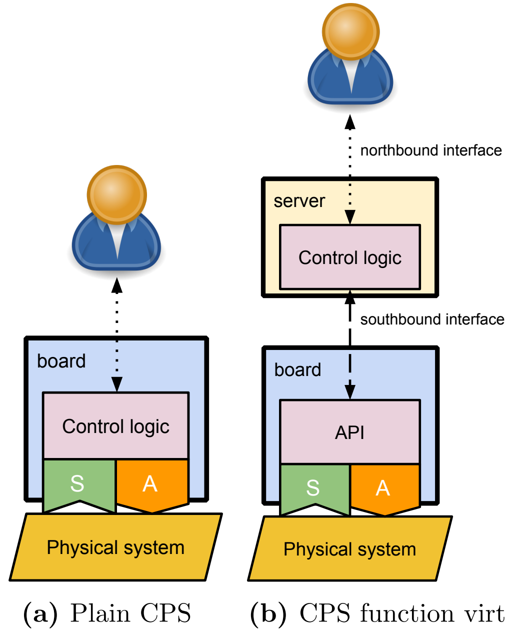
In order to fill the gap between Smart City applications and the underlying infrastructure in the SDC perspective, this chapter proposes to extend a well known framework for the management of Cloud computing resources, OpenStack, to sensing and actuation ones, implementing in the *Stack4Things* solution an infrastructure-oriented[16] two-layer approach, managing policies at control plane while coping with communication requirements and scalability concerns at data

plane, by leveraging Cloud-focused design choices and architectural patterns. In [37] details have been laid of a first step in the direction of standardized, cross-domain approaches, where the focus was in integrating an OpenSource framework for Cloud management, OpenStack with the IoT, addressing the data collection and visualization stages by leveraging existing functionalities and built-in scalability of the framework.

6.3 Overview of the approach

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, 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. In the following this idea is detailed focusing on the two levels of the proposed Software Defined Cities approach.



(a) Plain CPS **(b)** CPS function virt

Figure 6.1: *Cyber-Physical Systems*

6.3.1 Data Plane: Cyber-Physical Systems

A Cyber-Physical System (CPS) at its core may be defined as a system of computational elements interacting with physical entities. Often in the past such elements were referred to as *embedded systems*, where more emphasis is placed on the processing capabilities of such machines, compared to their pivotal arrangement at the boundary with the real world. Such a generic description lends itself to a diverse range of nuanced interpretations, yet in contrast to the typical CPS/IoT (*Internet of Things*) analogy, a couple of significant distinctions may be found squarely in the name itself, which sounds obviously rooted in the sensor/actuator-induced coupling between physical world phenomena and the digital domain, thus deemphasizing the role (inter)connectivity plays with regards to the IoT. More in general IoT

evokes an internetwork of (possibly) autonomous systems, while CPS is seen first and foremost as a *system*, underlining the organic, possibly centralized coordination, or even planned growth. Even if, when dealing with actual instances of complex real-world setups, most differentiations tend to blur, such conceptual framing helps in casting the discussion that follows on more convenient grounds.

Putting aside the definition of a CPS, it is now time to describe what a CPS may look like. In Figure 6.1 there are a couple of options to sketch a simple instance of a CPS, i.e., one that includes a single smart interface interacting by means of its transducers (sensors and actuators) with a physical entity. The standard configuration, as depicted in Figure 6.1a, features a "plain" CPS, thus the interface subsystem (e.g., a board) acts on its own, and any end-user interacts with the physical world through the node itself. A first useful abstraction needed for investigation further along this chapter is represented in Figure 6.1b, where the role played by the interface gets (partially) shifted from the actual hardware instance of the sensor-/actuator-hosting platform to another (physically detached, possibly remote) machine, whose only requirements are some available processing (and storage) quotas. This means exposing a *northbound* interface mostly equivalent to the one provided by a plain configuration, by leaving to a *southbound* interface to expose just low-level I/O primitives. In order to capture this notion, such configuration may be aptly labelled as *CPS function virtualization*.

6.3.2 Control Plane: Smart Cities

In line with the overarching CPS virtualization approach, such foreseeable outcome ultimately calls for function virtualization at a metropolitan scale, a *Cyber-City System* (CCS), as depicted in Figure 6.2, where Cloud-hosted *virtual boards* are introduced and two

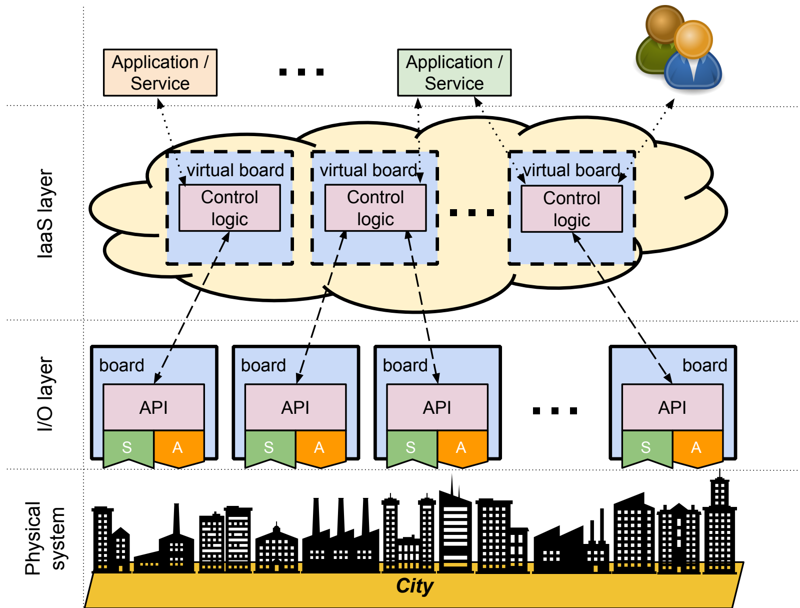
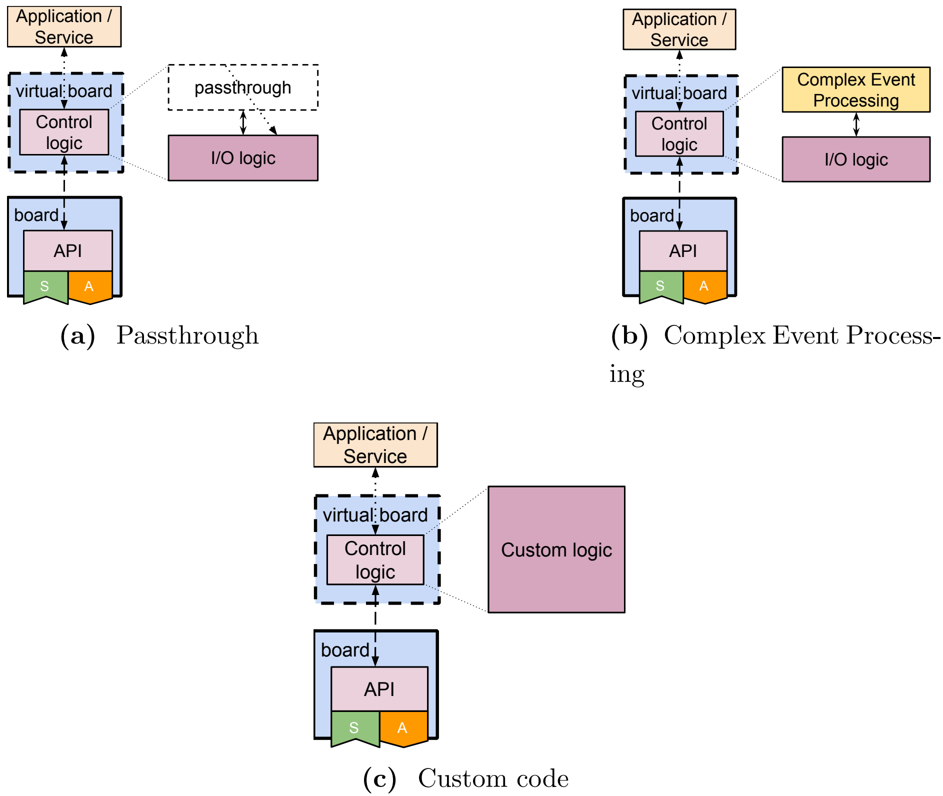


Figure 6.2: *Cyber-City System function virtualization*

"cyber" levels may be identified above the large-scale physical system that is the City: a distributed *I/O layer* and a centralized one modeled as an *Infrastructure-as-a-Service layer*.

Zooming in on the virtualization blocks, a series of approaches may be envisioned, represented in Figure 6.3, to the design of control logic as empowered by the function virtualization capabilities: at a minimum there should be the option to just get passthrough access, as depicted in Figure 6.3a, from the application directly to I/O logic, to just let the virtual board drive the transducers and leave all other duties to the application itself. Shifting more duties from the application level to the (virtual) infrastructure may be obtained by injecting (a part of, or the whole) control logic as rules for a *Complex Event Processing* engine to consume and act upon, as described in Figure 6.3b. In the end, the developer may skip this abstraction altogether and just inject some custom code, featuring both the rules and the I/O driving



(a) Passthrough

(b) Complex Event Processing

(c) Custom code

Figure 6.3: *Control logic: approaches*

logic, as highlighted in Figure 6.3c.

Getting back at the level of the whole Cyber-City System, the proposal is to extend the approach even to include dynamic reconfiguration of the networking subsystem of the underlying nodes, according to Software Defined Networking techniques and the more general SD* paradigm, leading to the definitive abstraction of a so-called Software Defined City, as sketched in Figure 6.4, where one or more centralized controllers remotely deploy (and therefore implement) the needed topologies among nodes by means of generalized rules according to predefined policies. This means that an unlimited number of control

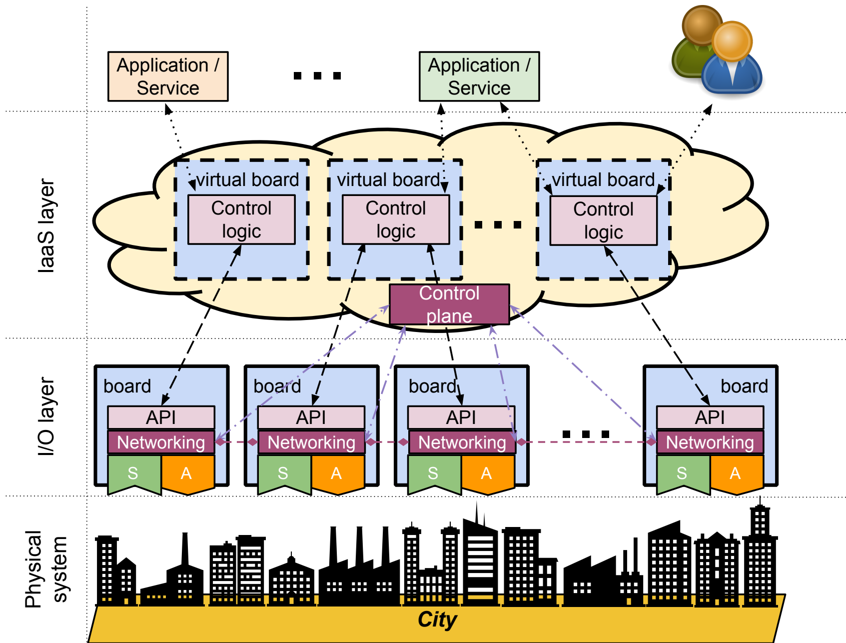


Figure 6.4: *Software Defined City*

loops of any scope and scale may be overlaid onto the SDC.

Plenty of applications may thus be envisioned, as long as all the aforementioned capabilities fall into place, from traffic monitoring to energy management, from e-health to e-government, from crowd to emergency management.

A middleware devoted to management of both sensor- and actuator-hosting resources may help in the establishment of these higher-level services, including policies for "closing the loop", such as, e.g., configuring triggers for a range of (dispersed) actuators based on sensing activities from (geographically non-overlapping) sensing resources. Figure 6.5 represents a high-level depiction of the overall scenario, where there need to be mechanisms to *rewire* such a "nervous system" into a number of elastic control loops.

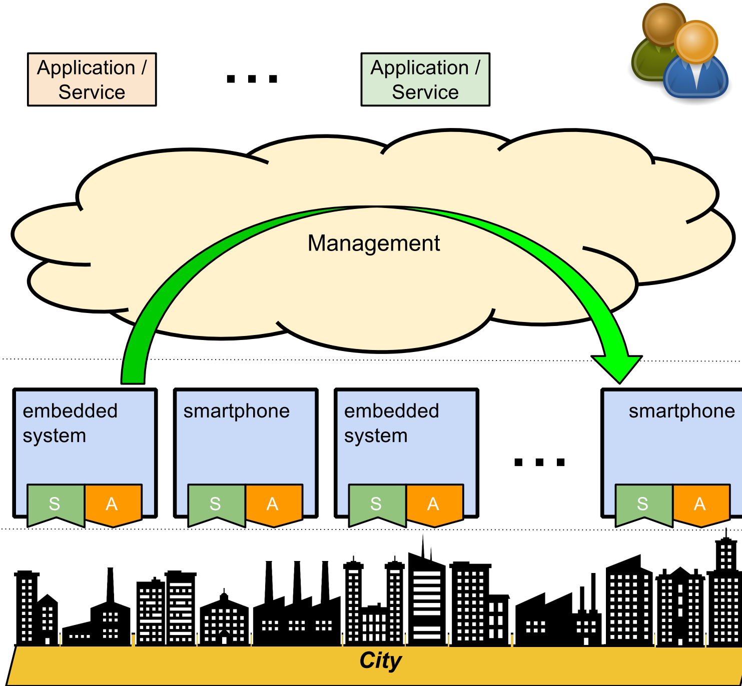


Figure 6.5: SD City as closed-loop system

6.4 Reference architecture

6.4.1 Requirements

The main actors in any SDC scenario are *contributors* and *end users*. Contributors provide sensing and actuation resources building up the Smart City infrastructure pool. End users control and manage the resources provided by contributors. In particular, end-users may behave as SDC infrastructure administrators and/or service providers, managing the raw resources and implementing applications and services on top of it. It is assumed that sensing and actuation resources are provided to the infrastructure pool via a number of hardware-constrained units, from now on referred to as *nodes*. Nodes host sensing and actuation resources and act as mediators in relation to the

Cloud infrastructure.

In order to actually accomplish the prospect of an SDC, a systematic requirement analysis is needed. A subset of requirements, shared by both virtualization and centralized control/orchestration objectives, are the ones relative to the contributor:

- **Out-of-the-box experience** - letting nodes and the corresponding sensors and actuators be enrolled automatically in the Cloud at, e.g., unpacking time.
- **Uniform interaction model** - resources should be hooked up (or unenrolled, when preferred) with the minimum amount of involvement for the contributor to feed the enrollment process with details about their hardware characteristics.
- **Contribution profile** - each contributor should be able to specify her profile for contribution in terms of resource utilization (CPU utilization, memory or disk space) and contribution period (frame time when the contributor is available for contribution).

and others coming from the end user such as:

- **Status tracking** - monitoring the status (presence, connectivity, usage, etc.) of nodes and corresponding resources, in order to, e.g., track significant outages or load profiles.
- **Lifecycle management** - exposing a set of available management primitives for sensing and actuation resources to, e.g., change sampling parameters when needed or, e.g., reap a pending actuation task to free the resource for another higher-priority duty.
- **Ubiquitous access** - enabled through instant-on bidirectional communication with resources as exposed from sensor-hosting nodes, whichever the constraints imposed by node-side network topology (e.g., NAT) and configuration (e.g., firewall).

- **Ensemble management** - letting nodes and the corresponding sensors and actuators be made available as pools of resources, e.g., to be partitioned in, and allocated as, groups according to requirements.
- **Instance provisioning** - resources should be made available (provisioned) subject to certain user-mandated constraints (geographical context, etc.), decoupling specific instances from the function they embody.
- **Orchestration** - exposing interfaces for the orchestration (e.g., dependency-based startup, endpoint wiring, etc.) of ensembles of resources.

A certain subset of end user requirements instead needs to be addressed by just providing the facilities for function virtualization, and the same considerations apply, for another subset, mainly in order to support a centralized orchestration of virtualized networking instances. With regard to the former, essential ones exclusively for this case are:

- **Delegation capabilities** - providing client-less (Cloud-enabled) interactions, by switching to alternative Cloud-hosted controlling surfaces (e.g., Web-based graphical or textual terminal) as needed, e.g., clients may need to disconnect at any time.
- **Uniform information model** - resources (e.g., down to single I/O pins) should be indexed according to a suitable model and searchable through standardized query syntax and predefined rules.

In relation to the latter, the case-specific list includes:

- **Service-oriented interfaces** - exposing primitives as asynchronous service endpoint, in order to ease development and third-party software integration.

- **Environment customization** - enabling runtime modifications to the software environment hosted by the node.
- **Topology rewiring** - providing mechanisms for the networking configuration underneath nodes to be modified at any time.

6.4.2 Sensing and Actuation as a Service for SDC

In the pursuit for integration of IoT infrastructure with paradigms and frameworks for heterogeneous resource management, a bottom-up approach is being followed, consisting of a mixture of relevant, working frameworks and protocols, on the one hand, and interesting use cases to be explored according to such integration effort, on the other.

To this purpose, Cloud computing facilities, here also implementing a service-oriented [16] approach in the provisioning and management of sensing and actuation resources, are exploited to enable a *Sensing and Actuation as a Service* (SAaaS) paradigm for SDC. In fact, in the SAaaS perspective, sensing and actuation devices should be handled along the same lines as computing and storage abstractions in traditional Clouds, i.e., on the one hand virtualized and multiplexed over (scarce) hardware resources, and on the other grouped and orchestrated under control of an entity implementing high level policies. This way, sensing and actuation devices have to be part of the Cloud infrastructure and have to be managed by following the consolidated Cloud approach, i.e., through a set of APIs ensuring remote control of software and hardware resources despite their geographical position.

A Cloud-oriented solution indeed may fit the SDC scenarios, meeting most requirements by default to cater to the originally intended user base, while at the same time also addressing other more subtle functionalities, such as a tenant-based authorization framework, where several actors (owners, administrator, users) and their interac-

tions with infrastructure may be fully decoupled from the workflows involved (e.g., transfer, rental, delegation). Bonus points include recycling existing (compute/storage-oriented) deployments, getting most visualization and monitoring technologies for free, as those are typically already available in such systems, possibly even enabling federation of different administrative Cloud-enabled domains.

Putting aside the core IaaS framework, as anticipated some additional facilities are needed for the envisioned Software Defined City paradigm and the specifics of the domain at hand (IoT), among which here two classes of mechanisms may be described that are core to the overall approach: those needed to access locally and transparently remote (I/O) resources, and those to set up arbitrary topologies among nodes.

CCS Functions Virtualization

With regard to the former, in Figure 2.3 (Section 2.4) a logical architecture may be found of the node-side stack needed for pub/sub or even RPC-style I/O primitives to be exposed to remote hosts through the Cloud, one of the core mechanisms for Cyber-City System function virtualization.

6.5 Use case

Once the SDC scenario has been laid out, it is easier to frame the discussion in terms of a focused scenario, such as management of large-scale emergency situations for civilians. A common trait in terms of requirements revolves around the assumption that, to be resilient while facing natural disasters, the population has to be kept in the loop as much and as early as possible, thus leading mobiles and other personal devices to be excellent candidates for prompt notifications,

or even lean feedback loops where users are actively involved, when needed.

A peculiar feature of such scenario lies in the lack of predefined boundaries in terms of the sensing infrastructure, which may span multiple geographical areas and administrative domains. Whichever the footprint of alerting and support activities for civilians, the foremost quality here is the dynamic involvement of infrastructure as well as the mapping this entails with regards to contributors, as mostly overlapping with end-users. Ultimately, these unique challenges may translate into requirements for:

- **On-demand solicitation and collection of measurements** by authorities (e.g., when an emergency gets forecasted), based on location of contributing devices.
- **Access and priority overrides** - to bypass standard checks with regard to QoS and SLA.
- **Federating domains** - where a central authority may not obtain enough permissions over, and thus control of, resources on certain areas of interest.

A scenario-specific use case with detailed description of the corresponding interactions follows, with a focus on the low-level management and use of SDC infrastructure facilities.

Reactive urban facilities to emergency events

In such a scenario a use case may be identified in the on-demand setup of facilities in the City that are ready to react to certain events which could anticipate an impending emergency, and may avoid or at least contain damages and/or casualties. For instance, a bridge may be considered at risk and put under control by placing the required

sensing infrastructure to monitor critical parameters, such as oscillations, load, and torque or compressive stress of certain sections and elements. In terms of actuators, the most fitting example may be gates at either side of the bridge, only involving entry lanes in order not to impact vehicular outflow, to be closed at the occurrence of such kind of event, as a precautionary step to be taken before deeper investigations. An operator of the emergency management services with access to the infrastructure just needs to put in place reactive mechanisms (switching the logic and replacing nodes to be involved, when needed) by invoking the SDC framework and resorting to:

- *Complex event processing with I/O function virtualization for CPS*

The aforementioned use case is implemented by deploying at least two transducers, a sensor and an actuator respectively, where a board driving an actuator hosts an application that operates it when triggered upon detection of an event of interest. The latter gets generated by a CEP engine every time predefined patterns (e.g., steady-state and/or structural anomalies) get recognized out of measurements by one or more sensors sampling the corresponding phenomena on (possibly other) boards. Interesting patterns are set by loading rules written in an engine-specific language.

The interactions are here described, when requesting for a number of boards currently enrolled to the Cloud to be booked, mapped to an enumerable set of resources, ultimately exposed for seamless interaction to a CEP engine, and the corresponding rules, deployed in Cloud-hosted VM:

1. Book two (or more) managed boards.

2. Request the instantiation of a VM based on an image, where a resource discovery service and CEP engine (plus the rules) is already deployed.

3. Push a list of reserved boards and their corresponding endpoints to the VM, in order to let the discovery service enumerate and expose as local I/O a set of remote resources, according to predefined policies.

4. Connect to the VM to start up the CEP, which is already configured to leverage the aforementioned resources.

According to the description of the core mechanisms for the Software Defined City, built on top of the IaaS framework, the first request is a routine one for the framework once extended to include enrollment of IoT nodes, as well as the second and the fourth one even when IoT extensions are not considered. The third request instead requires the framework to deploy (IaaS-context) data into a VM, but then the enumeration may take place only if the WAMP subsystem is available. Exposing remote resources as local I/O needs a wrapper around the same subsystem too.

CONCLUSIONS

This dissertation proposed a new device-centric perspective to approach the IoT problem domain, typically concerned with BigData management in sensing environments, with specific reference to data collection, with data-oriented interfaces as first-class mechanisms, and as such data-centric by definition. To this purpose an Infrastructure-oriented Cloud paradigm, Sensing and Actuation as a Service (SAaaS), is proposed for adoption in this context, since it provides mechanisms for customising resources through abstraction and virtualization technologies, and to provide any resource on demand, as a service. The Stack4Things framework has been therefore proposed as an implementation of the SAaaS vision within an OpenStack-based environment. A novel Mobile CrowdSensing (MCS) as-a-Service paradigm has been then proposed, under the guise of a platform for MCS mass deployment that essentially splits the MCS service application and infrastructure into two distinct levels of concern and supported functionalities, and is layered on top of SAaaS. Delving deeper into MCS use cases, a specific scenario has been defined, focusing on opportunistic contribution patterns and self-organizing, distributed approaches, to unlock the MCS potential for the design of innovative ITS applications. The solution proposed to exploit this distributed MCS pattern adapts and extends an ant colony optimization metaheuristic to a problem of pathfinding

and graph traversal according to a given distance metric, and has been then characterized into the ITS application domain, by dealing specifically with a traffic engineering problem, exploiting the opportunistic pattern for route planning. In order to enable elastic instantiation of overlay networks useful for similar crowd-powered scenarios, a novel take on network virtualization mechanisms for infrastructure management in IoT Clouds have been investigated. As an example of such a scenario, the Software Defined paradigm has been presented, to be considered as an approach to provide a simplified and programmable exploitation of the underlying ecosystem of devices so that innovative and powerful services can be realized.

Future work on Stack4Things will be devoted to extending and integrating other OpenStack services (e.g., Neutron) with SAaaS functionalities thus enabling more interesting use cases. Amid ongoing development efforts for MCSaaS, other interesting use cases and application scenarios are envisioned to be investigated going forward, including experimentation on actual devices and heterogeneous platforms. A refinement of the MCS-enabled cooperative ITS strategy for hierarchical meshes and correspondingly wider-scope optimization interplay is ongoing. Among forthcoming developments with regard to the proposed approach to network virtualization for IoT, a deeper integration of the design into the OpenStack fabric is expected to be investigated next, by leveraging Neutron directly, in order to enable more complex setups and higher-level abstractions. Future work on the Software-Defined City will include the validation of the whole architecture in a real-world Smart City scenario involving at first the Municipality of Messina, among other stakeholders, under the #SmartME umbrella project.

BIBLIOGRAPHY

[1] S. Distefano, G. Merlino, and A. Puliafito, "Device-centric sensing: an alternative to data-centric approaches," *Systems Journal, IEEE*, vol. 9, pp. –, Oct 2015.

[2] G. Merlino, S. Arkoulis, S. Distefano, C. Papagianni, A. Puliafito, and S. Papavassiliou, "Mobile crowdsensing as a service: a platform for applications on top of sensing clouds," *Future Generation Computer Systems*, pp. –, 2015.

[3] M. Fazio, G. Merlino, D. Bruneo, and A. Puliafito, "An architecture for runtime customization of smart devices," in *International Symposium on Network Computing and Applications*, (Los Alamitos, CA), pp. 157–164, IEEE COMPUTER SOC, 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264 USA, 2013.

[4] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4things: an openstack-based framework for iot," in *Future Internet of Things and Cloud (FiCloud), 2015 International Conference on*, pp. –, Aug 2015.

[5] G. Merlino, D. Bruneo, F. Longo, S. Distefano, and A. Puliafito, "Cloud-based network virtualization: An iot use case," in *Ad Hoc Networks* (N. Mitton, M. E. Kantarci, A. Gallais, and S. Papavassiliou, eds.), vol. 155 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 199–210, Springer International Publishing, 2015.

[6] Cisco Visual Networking Index (VNI), "The Zettabyte Era: Trends and Analysis - White Paper," June 2014.

[7] Gartner Inc., "Top 10 strategic technology trends for 2013," 2013.

[8] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch, "Everything as a service: Powering the new information economy," *Computer*, vol. 44, pp. 36–43, March 2011.

[9] 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.

[10] A. Shokrollahi, "Raptor codes," *Information Theory, IEEE Transactions on*, vol. 52, pp. 2551–2567, June 2006.

[11] M. Wang and B. Li, "R2: Random push with random network coding in live peer-to-peer streaming," *IEEE J.Sel. A. Commun.*, vol. 25, pp. 1655–1666, Dec. 2007.

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

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

[14] Open Networking Foundation, “Software-Defined Networking: The New Norm for Networks,” White Paper. April 13, 2012.

[15] IOT-A Project Consortium, “Final architectural reference model for the iot,” tech. rep., http://www.iot-a.eu/public/public-documents/d1.5/view, 2013.

[16] S. Distefano, G. Merlino, and A. Puliafito, “Sensing and Actuation as a Service: A new development for Clouds,” in *Proceedings of the 2012 IEEE 11th International Symposium on Network Computing and Applications*, NCA ’12, (Washington, DC, USA), pp. 272–275, IEEE Computer Society, Aug 2012.

[17] Open Geospatial Consortium, *OGC(R) Sensor Planning Service Implementation Standard.* OGC, 2.0 ed., 2011.

[18] Apache Foundation, *Apache Storm*, 2015 (accessed January 6, 2015). `https://storm.apache.org/documentation/Tutorial.html`.

[19] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, “Taming information-stealing smartphone applications (on android),” in *Trust and Trustworthy Computing*, pp. 93–107, Springer, 2011.

[20] E. Fernandes, A. Crowell, A. Aluri, and A. Prakash, “Anception: Application virtualization for android,” *CoRR*, vol. abs/1401.6726, 2014.

[21] M. Gordon, L. Zhang, and B. Tiwana, “PowerTutor A Power Monitor for Android-Based Mobile Platforms,” 2011.

[22] Y. Wang, S. Jain, M. Martonosi, and K. Fall, “Erasure-coding based routing for opportunistic networks,” in *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-tolerant Networking*, WDTN ’05, (New York, NY, USA), pp. 229–236, ACM, 2005.

[23] T. Ho, M. Medard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong, “A random linear network coding approach to multicast,” *Information Theory, IEEE Transactions on*, vol. 52, pp. 4413–4430, Oct 2006.

[24] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, “Network information flow,” *IEEE Transactions on Information Theory*, pp. 1204–1216, 2000.

[25] T. Clohessy, T. Acton, and L. Morgan, “Smart city as a service (scaas) - a future roadmap for e-government smart city cloud computing initiatives,” in *The 1st International Workshop on Smart City Clouds: Technologies, Systems and Applications*, no. DOI 978-1-4799-7881-6/14, (London), pp. 836–842, December 2014.

[26] S. Alam, M. Chowdhury, and J. Noll, “Senaas: An event-driven sensor virtualization approach for internet of things cloud,” in *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pp. 1 –6, nov. 2010.

[27] B. Consortium, “Betaas building the environment for the things as a service,” 2012.

[28] M. Yuriyama and T. Kushida, “Sensor-cloud infrastructure - physical sensor management with virtualized sensors on cloud computing,” in *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pp. 1 –8, sept. 2010.

[29] M. Avvenuti, P. Corsini, P. Masci, and A. Vecchio, “An application adaptation layer for wireless sensor networks,” *Pervasive Mob. Comput.*, vol. 3, pp. 413–438, Aug. 2007.

[30] M. Iqbal, D. Yang, T. Obaid, T. J. Ng, and H. B. Lim, “Demo abstract: A service-oriented application programming interface for sensor network virtualization,” in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pp. 143 –144, april 2011.

[31] J. Jamsa, M. Luimula, J. Schulte, C. Stasch, S. Jirka, and J. Schoning, "A mobile data collection framework for the sensor web," in *Ubiquitous Positioning Indoor Navigation and Location Based Service (UPINLBS), 2010*, pp. 1–8, 2010.

[32] G. Gil, A. Berlanga de Jesus, and J. Molina Lopez, "incontexto: A fusion architecture to obtain mobile context," in *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on*, pp. 1–8, 2011.

[33] 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.

[34] L. Sanchez, J. Galache, V. Gutierrez, J. Hernandez, J. Bernat, A. Gluhak, and T. Garcia, "Smartsantander: The meeting point between future internet research and experimentation and the smart cities," in *Future Network Mobile Summit (FutureNetw), 2011*, pp. 1–8, June 2011.

[35] "OpenStack documentation [URL]." http://docs.openstack.org.

[36] "Stack4Things source code [URL]." https://github.com/MDSLab.

[37] G. Merlino, D. Bruneo, S. Distefano, F. Longo, and A. Puliafito, "Stack4things: Integrating IoT with OpenStack in a Smart City context," in *Proceedings of the IEEE First International Workshop on Sensors and Smart Cities*, SMARTCOMP 2014, (Washington, DC, USA), IEEE, 2015.

[38] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, RFC Editor, December 2011.

[39] S. Papavassiliou, C. Papagianni, S. Distefano, G. Merlino, and A. Puliafito, "M2m interactions paradigm via volunteer computing and mobile crowdsensing.," in *Machine-To-Machine Communications - Architectures, Technology, Standards, and Applications* (J. M. Vojislav Misic, ed.), Oxford: Taylor & Francis, 2014.

[40] 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*, MCC '12, (New York, NY, USA), pp. 13–16, ACM, 2012.

[41] Fox, G.C. et al., "Open Source IoT Cloud," 2015. https://sites.google.com/site/opensourceiotcloud.

[42] G. Fox, S. Kamburugamuve, and R. Hartman, "Architecture and measured characteristics of a cloud based internet of things," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pp. 6–12, May 2012.

[43] F. Li, M. Voegler, M. Claessens, and S. Dustdar, "Efficient and scalable iot service delivery on cloud," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pp. 740–747, June 2013.

[44] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.

[45] I. LogMeIn, "xively," 2015. https://xively.com/.

[46] PTC, "ThingWorx - Internet of Things and M2M Application Platform," 2015. http://www.thingworx.com/.

[47] SmartThings, Inc., "SmartThings Open Cloud," 2015. http://www.smartthings.com/opencloud/.

[48] A. Alamri, W. S. Ansari, and M. M. Hassan, "A Survey on Sensor-Cloud: Architecture, Applications, and Approaches," *International Journal of Distributed Sensor Networks*, vol. 2013, 2013.

[49] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure - physical sensor management with virtualized sensors on cloud computing," in *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pp. 1–8, Sept 2010.

[50] M. M. Hassan, B. Song, and E.-N. Huh, "A framework of sensor-cloud integration opportunities and challenges," in *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, ICUIMC '09, (New York, NY, USA), pp. 618–626, ACM, 2009.

[51] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, pp. 14–23, Oct 2009.

[52] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar, "Provisioning software-defined iot cloud systems," in *Proceedings of the 2014 International Conference on Future Internet of Things and Cloud*, FICLOUD '14, (Washington, DC, USA), pp. 288–295, IEEE Computer Society, 2014.

[53] S. Alam, M. Chowdhury, and J. Noll, "Senaas: An event-driven sensor virtualization approach for internet of things cloud," in *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pp. 1–6, Nov 2010.

[54] J. Soldatos, M. Serrano, and M. Hauswirth, "Convergence of utility computing with the internet-of-things," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 874–879, July 2012.

[55] "Arduino [URL]." http://www.arduino.cc.

[56] "BaTHOS [URL]." https://github.com/ciminaghi/bathos-mcuio.

[57] FI-WARE team, "Fi-ware project website," 2011.

[58] R. Ganti, F. Ye, and H. Lei, "Mobile crowdsensing: current state and future challenges," *Communications Magazine, IEEE*, vol. 49, no. 11, pp. 32–39, 2011.

[59] Y. Xiao, P. Simoens, P. Pillai, K. Ha, and M. Satyanarayanan, "Lowering the barriers to large-scale mobile crowdsensing," in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, (New York, NY, USA), pp. 9:1–9:6, ACM, 2013.

[60] European Commission, "Definition of a research and innovation policy leveraging cloud computing and iot combination. tender specifications, smart 2013/0037," tech. rep., European Commission, 2013.

[61] J. Zhou, T. Leppanen, E. Harjula, M. Ylianttila, T. Ojala, C. Yu, H. Jin, and L. Yang, "Cloudthings: A common architecture for integrating the internet of things with cloud computing," in *Computer Supported Cooperative Work in Design (CSCWD), 2013 IEEE 17th International Conference on*, pp. 651–657, June 2013.

[62] A. Botta, W. de Donato, V. Persico, and A. Pescape, "On the integration of cloud computing and internet of things," in *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pp. 23–30, Aug 2014.

[63] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte, *Modelling with generalized stochastic Petri nets.* John Wiley & Sons, Inc., 1994.

[64] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B. Srivastava, "Participatory sensing," in *In: Workshop on World-Sensor-Web (WSW'06): Mobile Device Centric Sensor Networks and Applications*, pp. 117–134, 2006.

[65] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell, "A survey of mobile phone sensing," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 140–150, 2010.

[66] B. Guo, Z. Yu, X. Zhou, and D. Zhang, "From participatory sensing to mobile crowd sensing," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pp. 593–598, March 2014.

[67] X. Yu, W. Zhang, L. Zhang, V. O. Li, J. Yuan, and I. You, "Understanding urban dynamics based on pervasive sensing: An experimental study on traffic density and air pollution," *Mathematical and Computer Modelling*, vol. 58, no. 56, pp. 1328 – 1339, 2013. The Measurement of Undesirable Outputs: Models Development and Empirical Analyses and Advances in mobile, ubiquitous and cognitive computing.

[68] E. Aubry, T. Silverston, A. Lahmadi, and O. Festor, "Crowdout: A mobile crowdsourcing service for road safety in digital cities," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pp. 86–91, March 2014.

[69] P. Mohan, V. N. Padmanabhan, and R. Ramjee, "Nericell: rich monitoring of road and traffic conditions using mobile smartphones," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, (New York, NY, USA), pp. 323–336, ACM, 2008.

[70] S. Hu, L. Su, H. Liu, H. Wang, and T. Abdelzaher, "Smartroad: A crowd-sourced traffic regulator detection and identification system," in *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, IPSN '13, (New York, NY, USA), pp. 331–332, ACM, 2013.

[71] G. Merlino, D. Bruneo, S. Distefano, F. Longo, A. Puliafito, and A. Al-Anbuky, "A smart city lighting case study on an openstack-powered infrastructure," *Sensors*, vol. 15, no. 7, p. 16314, 2015.

[72] G. Cardone, L. Foschini, P. Bellavista, A. Corradi, C. Borcea, M. Talasila, and R. Curtmola, "Fostering participaction in smart cities: a geo-social crowdsensing platform," *Communications Magazine, IEEE*, vol. 51, pp. 112–119, June 2013.

[73] D. Zhao, X.-Y. Li, and H. Ma, "Budget-feasible online incentive mechanisms for crowdsourcing tasks truthfully," *Networking, IEEE/ACM Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[74] F.-J. Wu and T. Luo, "Wifiscout: A crowdsensing wifi advisory system with gamification-based incentive," in *Mobile Ad Hoc and Sensor Systems (MASS), 2014 IEEE 11th International Conference on*, pp. 533–534, Oct 2014.

[75] Y. Chon, N. D. Lane, F. Li, H. Cha, and F. Zhao, "Automatically characterizing places with opportunistic crowdsensing using smartphones," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, (New York, NY, USA), pp. 481–490, ACM, 2012.

[76] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell, "Bikenet: A mobile sensing system for cyclist experience mapping," *ACM Trans. Sen. Netw.*, vol. 6, pp. 6:1–6:39, Jan. 2010.

[77] X. Chen, E. Santos-Neto, and M. Ripeanu, "Crowdsourcing for on-street smart parking," in *Proceedings of the second ACM international symposium on Design and analysis of intelligent vehicular networks and applications*, DIVANet '12, (New York, NY, USA), pp. 1–8, ACM, 2012.

[78] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan, "Medusa: a programming framework for crowdsensing applications," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, (New York, NY, USA), pp. 337–350, ACM, 2012.

[79] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos, "Anonysense: Privacy-aware people-centric sensing," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, (New York, NY, USA), pp. 211–224, ACM, 2008.

[80] N. Brouwers and K. Langendoen, "Pogo, a middleware for mobile phone sensing," in *Proceedings of the 13th International Middleware Conference*, pp. 21–40, Springer-Verlag New York, Inc., 2012.

[81] X. Hu, T. Chu, H. Chan, and V. Leung, "Vita: A crowdsensing-oriented mobile cyber-physical system," *Emerging Topics in Computing, IEEE Transactions on*, vol. 1, pp. 148–165, June 2013.

[82] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma, "Prism: platform for remote sensing using smartphones," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 63–76, ACM, 2010.

[83] T. Metsch, A. Edmonds, R. Nyrén, and A. Papaspyrou, "Open cloud computing interface–core," in *Open Grid Forum, OCCI-WG, Specification Document. Available at: http://forge. gridforum. org/sf/go/doc16161*, 2010.

[84] O. T. T. Committee, "Topology and Orchestration Specification for Cloud Applications," tech. rep., OASIS.

[85] R. Mordani and S. W. Chan, "Java servlet specification," *Sun Microsystems Inc., version*, vol. 3, 2009.

[86] "Apache Tomcat."

[87] A. Developers, "Google cloud messaging for android."

[88] "PoliMi POLICLOUD."

[89] "Waze: Free gps navigation with turn by turn," 2012.

[90] R. Long Cheu, C. Xie, and D.-H. Lee, "Probe vehicle population and sample size for arterial speed estimation," *Computer-Aided Civil and Infrastructure Engineering*, vol. 17, no. 1, pp. 53–60, 2002.

[91] W. F. Adams, "Road traffic considered as a random series.(includes plates).," *Journal of the ICE*, vol. 4, no. 1, pp. 121–130, 1936.

[92] "Android open source project."

[93] "Genymotion."

[94] "Android Debug Bridge."

[95] S. Distefano, G. Merlino, and A. Puliafito, "A utility paradigm for iot: The sensing cloud," *Pervasive and Mobile Computing*, vol. 20, no. 0, pp. 127 – 144, 2015.

[96] European Parliament, "Directive 2010/40/EU," 2010.

[97] M. Da Lio, F. Biral, E. Bertolazzi, M. Galvani, P. Bosetti, D. Windridge, A. Saroldi, and F. Tango, "Artificial Co-Drivers as a Universal Enabling Technology for Future Intelligent Vehicles and Transportation Systems," *Int. Transp. Sys., IEEE Trans. on*, vol. 16, pp. 244–263, Feb 2015.

[98] N. Groot, B. De Schutter, and H. Hellendoorn, "Toward sys.-optimal routing in traffic networks: A reverse stackelberg game approach," *Int. Transp. Sys., IEEE Trans. on*, vol. 16, pp. 29–40, Feb 2015.

[99] L. Du and H. Dao, "Information dissemination delay in vehicle-to-vehicle communication networks in a traffic stream," *Int. Transp. Sys., IEEE Trans. on*, vol. 16, pp. 66–80, Feb 2015.

[100] E. Lee, E.-K. Lee, M. Gerla, and S. Oh, "Vehicular cloud networking: architecture and design principles," *Communications Magazine, IEEE*, vol. 52, pp. 148–155, February 2014.

[101] V. Hodge, S. O'Keefe, M. Weeks, and A. Moulds, "Wireless sensor networks for condition monitoring in the railway industry: A survey," *Int. Transp. Sys., IEEE Trans. on*, vol. PP, no. 99, pp. 1–19, 2014.

[102] B. Ai, X. Cheng, T. Kurner, Z. dui Zhong, K. Guan, R.-S. He, L. Xiong, D. Matolak, D. Michelson, and C. Briso-Rodriguez, "Challenges toward wireless communications for high-speed railway," *Int. Transp. Sys., IEEE Trans. on*, vol. 15, pp. 2143–2158, Oct 2014.

[103] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic flow prediction with big data: A deep learning approach," *Int. Transp. Sys., IEEE Trans. on*, vol. PP, no. 99, pp. 1–9, 2014.

[104] S. Hamdar, A. Talebpour, and J. Dong, "Travel time reliability versus safety: A stochastic hazard-based modeling approach," *Int. Transp. Sys., IEEE Trans. on*, vol. 16, pp. 264–273, Feb 2015.

[105] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *Pervasive Computing, IEEE*, vol. 7, pp. 12–18, Oct 2008.

[106] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden, "CarTel: A Distributed Mobile Sensor Computing Systems," in *Proc. of the 4th Intern. Conf. on Embedded Networked Sensor Sys.*, SenSys '06, (New York, NY, USA), pp. 125–138, ACM, 2006.

[107] A. Petkovics and K. Farkas, "Efficient event detection in public transport tracking," in *Telecommunications and Multimedia (TEMU), 2014 Intern. Conf. on*, pp. 74–79, July 2014.

[108] X. Hu, X. Li, E.-H. Ngai, V. Leung, and P. Kruchten, "Multidimensional context-aware social network architecture for mobile crowdsensing," *Communications Magazine, IEEE*, vol. 52, pp. 78–87, June 2014.

[109] X. Hu and V. C. Leung, "Towards context-aware mobile crowdsensing in vehicular social networks," in *Cluster, Cloud and Grid Computing (CCGrid), 15th IEEE/ACM Intern. Symp. on*, pp. 749–752, May 2015.

[110] X. Hu, J. Zhao, B.-C. Seet, V. Leung, T. Chu, and H. Chan, "S-aframe: Agent-based multilayer framework with context-aware semantic service for vehicular social networks," *Emerging Topics in Computing, IEEE Trans. on*, vol. 3, pp. 44–63, March 2015.

[111] D. Barth, "The bright side of sitting in traffic: Crowdsourcing road congestion data," 2009. http://googleblog.blogspot.ca/2009/08/bright-side-of-sitting-in-traffic.html.

[112] M. Gerla, E.-K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pp. 241–246, March 2014.

[113] B. Guo, D. Zhang, Z. Wang, Z. Yu, and X. Zhou, "Opportunistic iot: Exploring the harmonious interaction between human and the internet of things," *Journal of Network and Computer Applications*, vol. 36, no. 6, pp. 1531 – 1539, 2013.

[114] D. Zhao, H. Ma, S. Tang, and X.-Y. Li, "COUPON: A Cooperative Framework for Building Sensing Maps in Mobile Opportunistic Networks," *Parallel and Distributed Sys., IEEE Trans. on*, vol. 26, pp. 392–402, Feb 2015.

[115] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[116] Y.-T. Yu, Y. Li, X. Ma, W. Shang, M. Sanadidi, and M. Gerla, "Scalable opportunistic vanet content routing with encounter information," in *Network Protocols (ICNP), 2013 21st IEEE Intern. Conf. on*, pp. 1–6, Oct 2013.

[117] Y.-T. Yu, X. Li, M. Gerla, and M. Sanadidi, "Scalable vanet content routing using hierarchical bloom filters," in *Wireless Communications and Mobile Computing Conf. (IWCMC), 2013 9th Intern.*, pp. 1629–1634, July 2013.

[118] F. Bruno, M. Cesana, M. Gerla, G. Mauri, and G. Verticale, "Optimal content placement in icn vehicular networks," in *Network of the Future (NOF), 2014 Intern. Conf. and Workshop on the*, vol. Workshop, pp. 143–147, Dec 2014.

[119] P. Kromer, J. Martinovic, M. Radecky, R. Tomis, and V. Snasel, "Ant colony inspired algorithm for adaptive traffic routing," in *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*, pp. 329–334, Oct 2011.

[120] D. Bruneo, M. Scarpa, A. Bobbio, D. Cerotti, and M. Gribaudo, "Markovian agent modeling swarm intelligence algorithms in wireless sensor networks," *Performance Evaluation*, vol. 69, pp. 135–149, 2012.

[121] D. Cerotti, M. Gribaudo, A. Bobbio, D. Bruneo, and M. Scarpa, "An intelligent swarm of markovian agents," tech. rep., Università del Piemonte Orientale, Alessandria, Italy, TR-INF-2014-06-01-UNIPMN 2014.

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

[123] A. Fischer, J. Botero, M. Till Beck, H. de Meer, and X. Hesselbach, "Virtual network embedding: A survey," *Communications Surveys Tutorials, IEEE*, vol. 15, pp. 1888–1906, Fourth 2013.

[124] I. Ishaq, J. Hoebeke, I. Moerman, and P. Demeester, "Internet of things virtual networks: Bringing network virtualization to resource-constrained devices," in *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pp. 293–300, Nov 2012.

[125] "VPN-WS [URL]." https://github.com/unbit/vpn-ws.

[126] S. Hanks, T. Li, D. Farinacci, and P. Traina, "Generic Routing Encapsulation over IPv4 networks," RFC 1702, RFC Editor, October 1994.

[127] M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris, "Smarter cities and their innovation challenges," *Computer*, vol. 44, pp. 32–39, June 2011.

[128] J. Lee, S. Baik, and C. Choonhwa Lee, "Building an integrated service management platform for ubiquitous cities," *Computer*, vol. 44, pp. 56–63, June 2011.

[129] Z. Li, C. Chen, and K. Wang, "Cloud computing for agent-based urban transportation systems," *IEEE Intelligent Systems*, vol. 26, pp. 73–79, Jan. 2011.

[130] N. Mitton, S. Papavassiliou, A. Puliafito, and K. S. Trivedi, "Combining cloud and sensors in a smart city environment," *EURASIP J. Wireless Comm. and Networking*, vol. 2012, p. 247, 2012.