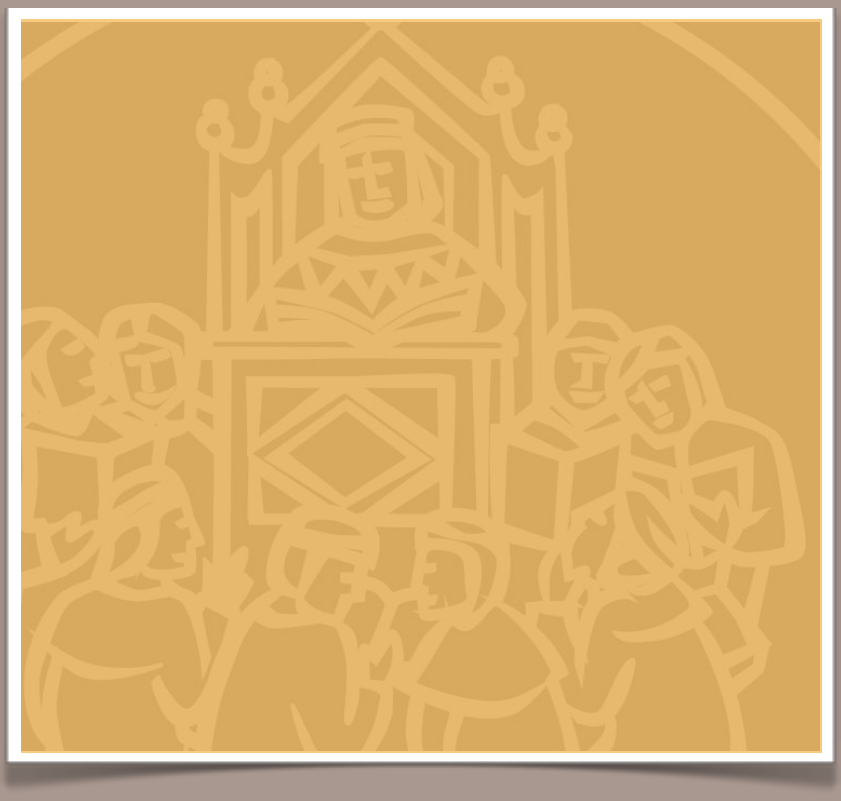
Andrea Azzarà

Programming Abstractions for the Internet of Things: from Macroprogramming to Virtual Resources



Scuola Superiore
Sant'Anna
di Studi Universitari e di Perfezionamento

Anno Accademico
2014-2015
Corso di perfezionamento
in Tecnologie Innovative

Programming Abstractions for the Internet of Things: from Macroprogramming to Virtual Resources

Autore
Andrea Azzarà

Tutor
Prof. Giorgio Buttazzo

Supervisor
Dott. Paolo Pagano

Abstract

The evolution of the Internet constantly changes the way we interact with our world. The ongoing transition from an Internet of People to an Internet of Things offers countless opportunities of societal development and improvement of our everyday life. Inspired by the research on Wireless Sensor Networks, the Internet of Things is based on the interconnection of the physical and the digital worlds, with billions of new connected devices expected in the next ten years. Smart objects, characterized by sensing, actuation, computation and communication capabilities, are expected to become an essential part of our world. The application scenarios and the actors of this revolution are numerous and heterogeneous. However, the development of a connected world poses novel challenges to the research community. On the one hand, networks of smart objects are expected to be easy to deploy, program and integrate with high-end systems. On the other hand the demand for long-lasting and large-scale applications require a constant research effort with the goal of reducing energy consumption and optimizing network protocols. Unfortunately, an extensive adoption of networks of smart objects is still hindered by the interaction with low-level communication and computation details. The problem is made more challenging by the increased complexity of the applications in which smart objects are used.

The goal of this thesis is to present new abstractions and tools to facilitate the development of applications for the Internet of Things. The thesis focuses on the introduction of novel concepts for the abstraction and the virtualization of the resources offered by smart objects. We first introduce the recent evolution of Wireless Sensor Networks in multimedia systems. The Wireless Multimedia Sensor Networks offer new possibilities to develop intelligent and versatile applications. Then we describe a modular middleware allowing to exploit the potential of new sensing technologies. However, the ubiquitous adoption of the proposed solution poses new requirements that can only be fulfilled by Internet-based systems. In the rest of the thesis we focus on abstraction mechanisms aimed at simplifying the development of applications for Internet of Things systems. We first describe an abstraction of the network resources allowing to easily develop applications for groups of smart nodes through macroprogramming scripts. Then we demonstrate the flexibility of the proposed architecture presenting two different application scenarios: the Smart Factory and the Intelligent Transport System. Finally, we introduce a software architecture defining a virtualization model for the Internet of Things resources as a tool to promote the separation of concerns in the software development cycle, still supporting advanced features such as the distributed information processing.

To my family.

Contents

List of Publications

Publications included in the thesis

1. D. Alessandrelli, A. Azzarà, M. Petracca, C. Nastasi, and P. Pagano. ScanTraffic: Smart Camera Network for Traffic Information Collection. In *Proceedings of European Conference on Wireless Sensor Networks*, pages 196–211, Trento, Italy, February 2012
2. A. Azzarà, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano. PyoT, a macro-programming framework for the Internet of Things. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*. IEEE, 2014
3. M. Petracca, S. Bocchino, A. Azzarà, R. Pelliccia, M. Ghibaudi, and P. Pagano. WSN and RFID Integration in the IoT scenario: an Advanced Safety System for Industrial Plants. *Journal of Communications Software & Systems*, 9(1), 2013
4. A. Azzarà, M. Petracca, and P. Pagano. The ICSI M2M Middleware for IoT-based Intelligent Transportation Systems. In *International Workshop on COoperative Sensing for Smart MObility (COSSMO)*, 2015
5. A. Azzarà and L. Mottola. Virtual Resources for the Internet of Things. In *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT), Milano (Italy)*, December 2015

Other publications

In addition to the papers included in the thesis I have also co-authored the following papers:

1. P. Pagano, C. Salvadori, S. Madeo, M. Petracca, S. Bocchino, D. Alessandrelli, A. Azzarà, M. Ghibaudi, G. Pellerano, and R. Pelliccia. A middleware of things for supporting distributed vision applications. In *Proceedings of the 1st Workshop on Smart Cameras for Robotic Applications (SCaBot)*, 2012
2. A. Azzarà, D. Alessandrelli, S. Bocchino, P. Pagano, and M. Petracca. Architecture, Functional Requirements, and Early Implementation of an Instrumentation Grid for the IoT. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, June 2012
3. A. Azzarà, S. Bocchino, P. Pagano, G. Pellerano, and M. Petracca. Middleware solutions in WSN: The IoT oriented approach in the ICSI project. In *Software, Telecommunications and Computer Networks (SoftCOM), 2013 21st International Conference on*, pages 1–6, Sept 2013

4. A. Azzarà, D. Alessandrelli, M. Petracca, and P. Pagano. Demonstration abstract: PyoT, a macroprogramming framework for the IoT. In *Proceedings of the 13th international symposium on Information Processing in Sensor Networks*, pages 315–316. IEEE Press, 2014
5. F. Pacini, A. Azzarà, S. Bocchino, F. Aderohunmu, M. Petracca, and P. Pagano. Poster abstract: Performance analysis of data serialization formats in m2m wireless sensor networks. In *Proceedings of the 12th European Conference on Wireless Sensor Networks*, 2015

CHAPTER 1

Introduction

WITH more than three billions users, the Internet is one of the largest infrastructures ever built. Over the last twenty years, the Internet radically changed the way people interact with the virtual world. The Internet of Things represents the next revolution of the Internet, in which common objects will be able to communicate and cooperate with each other to realize innovative applications. The Internet of Things relies on two fundamental technological cornerstones, the Radio Frequency Identification (RFID) and Wireless Sensor Networks [**MPV11**]. The characteristics of pervasiveness, physical world sensing, and actuation of the Internet of Things, permit to forecast a huge impact on the world economy and on the life of people [**MCB+15**]. The application domains of the Internet of Things are numerous and not yet entirely defined. Among the most significant ones it is worth to mention the transportation and logistic, health-care, smart environments, personal and social domains [**AIM10**]. The concept of Smart City is an effective collector for those scenarios [**VD10**]. In the Smart Cities context the Internet of Things allows to face the challenges posed by the city of the future, with a particular emphasis on sustainability and on the quality of life of the citizens. The cooperation of sensors, actuators and mobile devices will allow to collect and process detailed environmental information, enabling new possibilities of analysis and forecast of urban phenomena. Recent urban-scale deployments demonstrated the potential impact of ICT in the creation of value for the citizens [**HMnVMn+11**, **OBB+13**].

The Smart Objects are one of the pillars of the Internet of Things revolution. Technological advancements in the field of electronics enabled the creation of small and low-cost devices that can become an integral part of the environment embedding tiny processors and network interfaces allowing the communication with the global Internet. The true innovation potential of the Smart Objects resides in their capacity of exchanging information on a global scale speaking an universal language. A massive adoption of Smart Objects faces unique challenges. The first problem is represented by the number of connected devices and by the potential scale of the deployments. The unprecedented pervasiveness of the Internet of Things demands significant efforts in the fields of network architectures and protocols in order to guarantee a scalable and seamless interoperability among heterogeneous systems. The second issue is represented by the energy consumption of smart devices, which are often battery powered. A specific field of research is focused on the creation of novel power saving and energy harvesting mechanisms. Another challenge is represented by the scarce resources typically offered by Smart Objects. The small size and the low-power features of Smart Objects impose strict limitations on available memory and computing power. Designing applications for the Internet of Things requires a specific attention to the optimization of resource utilization. The limitation of resources and the need for power

efficiency affect also the networking capacity of Smart Objects. Internet of Things networks are characterized by low bitrates, limited transmission range and small messages.

1.1. Enabling Technologies

In order to support Internet-based communication in the context of Smart Objects, a specific network stack has been defined by standardization entities. IEEE 802.15.4 is the reference standard for Low-Rate Wireless Personal Area Networks (LR-WPANs), focusing on short-range operation, low-data-rate, energy-efficiency, and low-cost. The standard is the base for several wireless industrial technologies, such as ISA100.11a and WirelessHART, in which proprietary network and application layers are considered. At the networking layer the IETF 6LoWPAN standard aims at integrating existing IP-based infrastructures and sensor networks by specifying compression policies for sending IPv6 datagrams in IEEE 802.15.4-based networks. According to the 6LoWPAN concept, "the Internet Protocol could and should be applied even to the smallest devices" [**Mul07**]. 6LoWPAN defines the frame format for the transmission of IPv6 packets, mechanisms for header compression, and formalizes how to create IPv6 link-local addresses on top of IEEE 802.15.4 networks. Along with 6LoWPAN, the Routing Protocol for Low-power and Lossy networks (RPL) [**WTB+12**] has been proposed by the IETF ROLL group as a standard. Routing issues are very challenging for 6LoWPAN due to the low-power nature of such networks, multi-hop mesh topologies, and frequent topology changes. At the application layer the CoAP [**BCS12**] protocol has been recently standardized by the IETF CoRE working group, providing a RESTful framework [**RR08**] for resource-oriented applications. CoAP supports RESTful transactions, resource abstraction, URIs addressing, while avoiding most of the complexities of HTTP. Moreover CoAP provides mechanisms for resource discovery and an asynchronous notification model allowing servers to stream the state of an observed resource to the clients over a period of time. CoAP is specifically designed to cope with limited packet size, low-energy devices and unreliable channels [**KDD11**]. Through CoAP it is possible to manage RESTful web services in constrained devices. An important concept in REST is the existence of resources (sources of specific information), each of them referenced with a global identifier. A resource can be essentially any coherent and meaningful concept that may be addressed and manipulated through standard methods.

Despite the technological advancements and the standardization efforts, the fulfillment of the Internet of Things vision is delayed by many factors. In the following section we describe some of the research problems posed by the Internet of Things.

1.2. Research Challenges

The rise of the Smart Objects era will enable the creation of pervasive monitoring and actuation applications on large scale, having a revolutionary impact on society. Through the adoption of open standards, the research community successfully transformed the Wireless Sensors Networks from a collection of isolated environments to inter-operable systems connected to the Internet. However, the Internet of Things is still in its infancy and far from showing its real

potential. A reason for this delay can be identified in the complexity of developing applications for Internet of Things systems. Smart Objects are inherently difficult to program and to debug. Many applications require a degree of coordination and cooperation among the nodes, which further complicates the development process. Another issue is related to the large number of different technological solutions emerged in recent years. Hence, programming IoT systems requires a deep knowledge of several different engineering domains, including electronics, networking and low-level system architectures. Many solutions have been proposed in order to simplify the development and maintenance processes of Wireless Senor Networks. However, those solutions usually lack of generality for two reasons. First, they are often targeted to isolated systems, often adopting proprietary protocols, which are difficult to interoperate with others. Second, they are designed for a specific application and are difficult to generalize. The Internet of Things paradigm requires instead open solutions, easily adaptable to new application domains. Also, in Internet of Things systems scalability issues acquire a fundamental importance.

1.3. Contribution and outline

This thesis is dedicated to the presentation of novel abstraction and tools designed to simplify the development of applications for Internet of Things systems. Chapter 2 introduces the evolution of Wireless Sensor Networks in multimedia-enabled systems. Wireless Multimedia Sensor Networks (WMSN) present unique opportunities for advanced and cost-effective monitoring applications to be rapidly deployed in the environment. Yet they require specific tools designed for supporting a complex configuration and management phase. ScanTraffic is a middleware with real time features capable of supporting advanced monitoring applications, showing how a proper network and software design is necessary to fully exploit smart cameras features.

The application domains of WMSN provide an ideal case study for the Internet of Things paradigm due to the typical large scale of the deployments and the need for standard interfaces simplifying the configuration procedures. Indeed, the Internet of Things *resource* abstraction model permits a significant simplification of the configuration and discovery processes of Smart Objects. Chapter 3 focuses on the abstraction of resources in Internet of Things (IoT) systems. PyoT is a macroprogramming system facilitating the development of applications for the IoT. PyoT abstracts IoT resources as programming language objects allowing the programmers to interact with large sets of resources through simple macroprogramming scripts. Thanks to the resource abstraction it is also possible to dynamically configure the processing of IoT nodes. The architecture proposed by PyoT can be easily adapted to different application scenarios, as described in Chapter 4, in which two innovative applications for the Smart Factory and the Intelligent Transport System use cases are presented.

The WSN research community invested significant efforts in making sensor networks smarter, delegating processing tasks to the nodes and creating complex cooperation patterns [**KHH05**]. The distributed, *in-network processing* distinguishes the WSN as autonomous systems in which nodes are capable of taking decisions (e.g., on how to process or where to forward the sensed data). In typical WSN applications sensors transfer aggregated information to a special node (called the sink), after locally filtering and processing the raw data. The recent adaptation

of the Internet model in the Smart Objects world resulted in a different trend, in which the objects only expose elementary and application-agnostic resources, following the so called *thin server* model. At the same time, the growing number of actuators, capable of modifying the state of the environment, opened the way to complex control applications, processing input data coming from sensors in real time and generating feedback commands distributed to the actuators. Applications, residing on external Cloud-based systems, access the raw information provided by the nodes through RESTful interfaces. The benefits of this model are clear: Cloud-based systems offer virtually unlimited processing and storage resources. With an ever growing number of connected devices, the IoT is expected to produce huge amounts of data. Indeed, Cloud-based applications are considered the key to exploit the full potential of smart objects. Another advantage resides in the simplified structure of sensor networks. The nodes are only in charge of collecting raw data and exposing them as resources through a standard network interface, while all the logic of the application resides in Cloud-based systems. However, the Cloud-based model presents also some drawbacks. The first one affects application developers, who are rarely interested in raw data gathered by individual nodes. Rather they are usually interested in higher level, aggregated information. As a result application developers are burdened by unnecessary details about the underlying network architecture, protocols and sensing and actuation logic. IoT networks are expected to exhibit a dynamic nature, with nodes frequently connecting and disconnecting from the network. Hence, the applications logic is further complicated by the task of indexing the available resources. Since the Cloud-based model requires individual calls for each and every sensor/actuator involved in the computation the performance of the system is affected as well. The application is forced to interact with a potentially large number of nodes arbitrarily distributed. Also, multiple applications interacting with a common smart sensor infrastructure may generate significant traffic load, reducing network's lifetime and creating congestion in low bandwidth networks. Moreover, the Cloud-based model sacrifices the advantages of *in-network processing*, such as lower energy consumption and greater flexibility of the application. This issue is particularly relevant due to the growing demand for multimedia applications for which the Cloud-model is hardly applicable and a form of pre-processing of the information is required to reduce the amount of transmitted data.

Chapter 5 presents the Virtual Resources architecture, contrasting the Cloud-centric designs by giving developers the ability to push slices of the application logic down to the IoT network. The architecture is presented for the use case of intelligent buildings. Virtual Resources provide several benefits to developers, including better utilization of network resources that results in higher energy efficiency and lower latencies, a simplification of the application logic at the Cloud, and better separation of concerns throughout the development process. Most importantly, Virtual Resources allow the Cloud-model to coexist with a model based on distributed processing involving Smart Objects in the computation.

CHAPTER 2

Wireless Multimedia Sensor Networks

THE application domains where Wireless Sensor Networks (WSNs) are being deployed are becoming more and more complex including monitoring and actuation capabilities in industrial, commercial, and ordinary social environments. Intelligent Transport Systems (ITSs) are gaining growing interest from governments and research communities because of the economic, social and environmental benefits they can provide. An open issue in this domain is the need for pervasive technologies to collect traffic-related data. In this chapter we discuss the use of visual Wireless Sensor Networks (WSNs), i.e., networks of tiny smart cameras, to address this problem. We believe that smart cameras have many advantages over classic sensor motes. Nevertheless, we argue that a specific software infrastructure is needed to fully exploit them. We identify the three main services such software must provide, i.e., monitoring, remote configuration, and remote code-update, and we propose a modular architecture for them. We discuss our implementation of such architecture, called ScanTraffic, and we test its effectiveness within an ITS prototype we deployed at the Pisa International Airport. We show how ScanTraffic greatly simplifies the deployment and management of smart cameras collecting information about traffic flow and parking lot occupancy.

Intelligent Transport Systems (ITSs) are nowadays at the focus of public authorities and research communities aiming at providing effective solutions for improving citizens lifestyle and safety. The effectiveness of such kind of systems relies on the prompt processing of the acquired transport-related information for reacting to congestion, dangerous situations, and, more generally, for optimizing the circulation of people and goods. To obtain a dynamic and pervasive environment where vehicles are fully integrated in the ITS, low cost technologies (capable of strongly penetrating the market) must be let available by the effort of academic and industrial research: for example low cost wireless devices are suited to establish a large-area roadside network.

In this chapter we describe ScanTraffic, the software managing the data collection layer of an ITS prototype developed within the Ipermob[1] project. Ipermob proposes a pervasive and heterogeneous infrastructure to monitor and control urban mobility. Its multi-tier architecture aims at the integration and optimization of the chain formed by data collection systems; aggregation, management, and on-line control systems; off-line systems aiming at infrastructure planning; information systems targeted to citizens and municipalities to handle and rule the vehicle mobility. Moreover Ipermob proposes to use visual Wireless Sensor Networks (WSNs) to

[1] "Infrastruttura Pervasiva Eterogenea Real-time per il controllo della Mobilità" (*"A Pervasive and Heterogeneous Infrastructure to control Urban Mobility in Real-Time"*).

collect traffic-related data. Nodes in a visual WSN are different from the traditional WSN motes. Indeed, they are (tiny) smart cameras, i.e., devices equipped with a microcontroller, an IEEE 802.15.4 transceiver and a low-resolution CMOS camera. Smart cameras use image-processing techniques to extract information from the acquired image.

The main advantage of smart camera over classic sensors is versatility. An image (and a sequence of images) contains much more information than a scalar value, therefore camera-based sensors can perform a wide range of tasks, *functionally replacing* different types of sensors. For example, a smart camera can be used as a light sensor, a motion detector, an occupancy sensor, etc. Smart cameras can also *quantitatively replace* classic sensors. For example we use a single smart camera to monitor the occupancy status of up to 10 parking spaces, instead of using one inductive sensor for each space. The major drawback of smart cameras is higher cost and power consumption. Yet the higher cost per unit is offset by the reduction in the number of sensors, that also leads to cheaper deployment and maintenance. We address the power consumption problem adopting on-board image processing: only relevant information (e.g., number of counted vehicles, parking space occupancy status, etc.) is sent over the network, thus reducing the number of exchanged messages which are the main cause of energy consumption in WSNs.

Due to their peculiarities, smart cameras and visual WSNs require a software infrastructure capable of fully exploiting their features. Within the Ipermob project we addressed this issue and we developed ScanTraffic to manage the visual WSNs used to collect traffic-related data. The contribution of our work is fourfold:

- We apply visual WSNs to the ITS domain, providing a new solution having many advantages over current systems based on scalar sensors;
- We identify the minimum set of services to be provided by a software managing a visual WSN;
- We propose a software architecture for such services;
- We discuss its implementation, deployment and validation within an ITS prototype.

The organization of this chapter is as follows. Section 2.1 discusses related work. Section 2.2 provides a brief description of the Ipermob project and describes ScanTraffic software design. Section 2.3 describes the implementation and deployment of ScanTraffic within the ITS prototype we built at the Pisa International Airport. Section 2.4 discusses the experimental validation of the system. Section 2.5 provides concluding remarks.

2.1. Related Work

WSN technology have already proven to be an effective tool for supporting next generation ITS, enabling pervasive traffic monitoring on surface streets, arterial roads, and freeways. In [**CCO+11**] a WSN is used to provide automatic adaptive lighting in road tunnels with the aim of improving road safety and reducing power consumption. In [**BBM+11**] acoustic sensors are employed to estimate traffic flows and early detect traffic jams. Both projects have been extensively validated on real testbed deployments. VTrack [**TRL+09**] and ParkNet [**MJK+10**] are ITS solutions both based on GPS localization, using mobile nodes to collect information

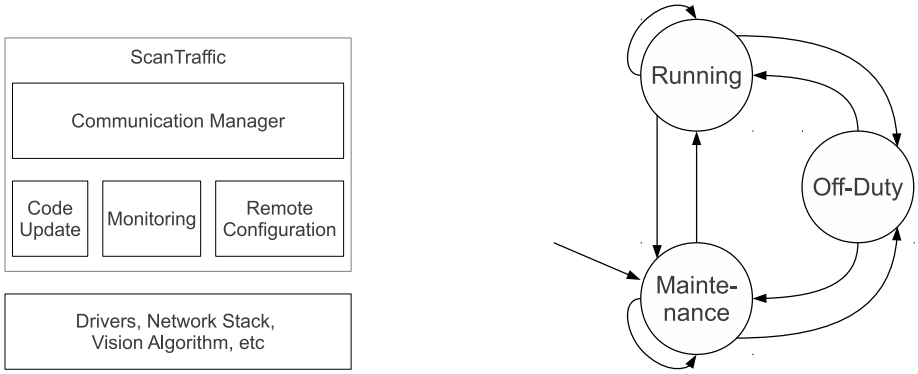
about traffic flows and parking occupancy. The main advantage of those systems is that a fixed infrastructure is not necessary. In VTrack smart-phones or on-board-units equipped with GPS are used to track vehicle positions which are sent to a central processing server. The effectiveness of the system relies on the localization accuracy and on the number of vehicles equipped with mobile nodes. In ParkNet a drive-by parking monitoring system is presented. Vehicles equipped with GPS and ultrasonic sensors move around the city detecting the presence of free parking spaces on the road side. Anyway such systems are not suitable for high mobility environments and for applications requiring high precision of the measurements. A Wireless sensor network for Intelligent Transportation System (WITS) [**CCCT06**] gives a systematic approach to design and deploy a WSN system to assist transportation. The WITS system makes use of three different types of wireless sensor nodes called vehicle units, roadside units and intersection units to gather and measure relevant information on vehicle and traffic movement. The system makes use of data aggregation policies depending on the request made by the traffic control execution system. TRACKSS [**AvA08**] is another project employing WSNs for cooperative sensing and for predicting flow, infrastructure and environmental conditions surrounding vehicle traffic. The goal of the project is to use advanced data fusion techniques for extracting additional information from the collected data, and to develop decision support systems for intelligent transportation.

Traditional cameras are currently used on a regular basis in the transportation domain to monitor traffic and detect incidents, e.g., with observations points in critical motorways. The advances in image sensor technology have enabled the development of low cost sensor networks performing collaborative processing of rich visual information extracted from the environment. In [**SH09**] a survey on visual WSN is presented, taking into account specific problems such as on-board image processing, object tracking and detection, communication protocols etc. In [**CGN$^+$05**] the concept of sensor network is extended to Internet enabled PC-class devices connected to off-the-shelf webcams. The system lets users query globally distributed nodes acquiring and transmitting data at a high bit rate. An example parking space monitoring application is included. Anyway, compared to ScanTraffic, resource-rich devices and high bandwidth links are used for image processing and data transmission.

Other research work has been focused on WSN in ITS. However, there exists very few real world implementation of systems employing WSNs and even less using video sensor for vehicles detecting and classification. In this regard, Ipermob presents a unique solution with video sensors giving better inference and estimation of traffic-related data.

2.2. System Design

The Ipermob project proposes a pervasive and heterogeneous infrastructure to monitor and control urban mobility. Within the project, a prototype of such infrastructure has been implemented and deployed at the Pisa International Airport. The Ipermob architecture has three tiers: *i)* data collection, *ii)* data sharing, and *iii)* data consumption. The data collection tier employs different technologies to acquire data related to urban mobility. Specifically, the prototype employs Vehicular Ad-hoc Networks (VANETs) and visual WSNs to collect traffic data. The data sharing tier provides, to the upper layer, a standard interface for accessing the



(a) General architecture. (b) Finite State Machine.

FIGURE 2.1. SCANTRAFFIC software design.

data produced by the lower layer and stores it for future use. The data consumption tier is the application layer. Applications can be on-line control systems providing real-time information to the users (drivers, police, etc.), or off-line systems aiming at infrastructure planning. The implemented prototype provides example applications of both types.

To manage and control the visual WSNs employed in the data collection tier, we designed and implemented SCANTRAFFIC, a distributed software infrastructure running within the WSN (i.e., in each node of the network) and providing three main services: monitoring, remote sensor configuration, and remote code-update. We argue that this is the minimal set of services to be provided by any visual WSN in order to fully exploit the potentiality of smart cameras. *Monitoring* is the primary service offered by any WSN. It allows to control the sensing activity performed by the nodes and to retrieve the collected data. *Remote sensor configuration* is often not necessary in traditional WSN employing simple sensors which can be configured (e.g., calibrated) before deployment. On the contrary, it is an important service for smart cameras whose visual algorithm needs to be configured for the current camera view, i.e., after deployment, and usually adjusted over time. *Remote code-update* is essential to benefit from smart camera versatility. By replacing sensor firmware, code-update allows to improve or completely change the functionality of a smart camera sensor. 2.1a shows the software architecture of SCANTRAFFIC. There is a total of 4 software modules: 3 service modules (one for each of the main services previously discussed), plus the "Communication Manager" controlling and coordinating them.

2.2.1. Monitoring Module

The purpose of WSNs is to acquire measurements from the environment and transmit them to a main collection point. The way this activity is performed should take into account timeliness, energy consumption, and reliability. Such issues contrast with each other, and each application requires a specific trade-off between them. Within the IPERMOB project, we use the MIRTES middleware [**APN+10**] to implement the monitoring functionality.

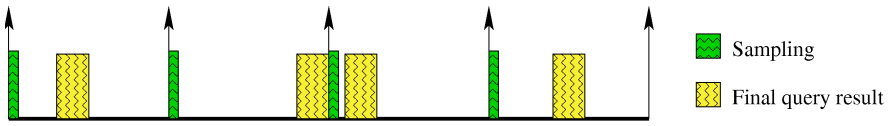


FIGURE 2.2. Example of jitter-free environment sampling by means of MIRTES periodic queries: the sensing is always performed at the beginning of the period, whereas the transmission of the acquired data can be delayed as long as it does not miss its deadline.

MIRTES is a real-time middleware providing a database-like abstraction of the WSN, thus allowing the user to retrieve sensor readings by spawning SQL-like queries. This kind of abstraction has two main advantages: it allows to easily interface the WSN with users and other systems, and it permits to support applications from different domains. The virtual database provided by MIRTES has a table for each physical quantity measured through the WSN: scalar quantities have (at least) three columns, two for the node ID and the time-reference associated to the measurements, and the other for its value; vector quantities have (at least) as many columns as their components, plus the node ID and time reference columns. The time reference associated to the measurements is taken from the MAC layer (see below). The information set required to fully describe the tables is called database schema.

MIRTES supports periodic queries having timeliness constraints. The system guarantees that, using periodic queries, the environment is periodically sampled with no jitter and that the sampling is synchronously undertaken in the WSN, i.e., all sensor nodes acquire their measurements exactly at the same time and specifically at the beginning of the periodic interval. As shown in 2.2, the transmission of the sampled data to the sink node can be delayed, but MIRTES ensures that, in any case, the query result is produced before the next activation. This behavior is especially suited for time-discrete control applications: using MIRTES periodic queries, the system output is measured exactly at time instant t_k and the measured value is available to the controller before instant t_{k+1}. If the user spawns a new periodic query, the system performs an acceptance test (admission control), which guarantees that the new instance does not disrupt the timeliness of the old set while fulfilling that of the new query.

MIRTES real-time features relay on a communication strategy based on the IEEE 802.15.4 standard. Specifically, MIRTES works with beacon-enabled WPANs supporting the Guaranteed Time Slot (GTS) feature. MIRTES sends the query in the beacon payload and allocates a GTS to each node that can potentially reply with its readings. This approach allows *i)* to know *a priori* the exact query execution time; *ii)* to schedule queries using the well-known EDF algorithm; and *iii)* to use the simple schedulability test based on utilization. The beacon is also used for time synchronization between nodes. The time is expressed in beacon intervals. In case of periodic queries, the jitter-free sampling of the environment is achieved synchronizing the sensing period with the query period by adding additional information (i.e., period and activation offset) to the query request. Sensor nodes always acquire measurements at the beginning of the period and keep them in memory until the transmission request is received.

MIRTES dependency on beacon-mode and GTS features currently constraints the WSN to a star topology, in which the sink node is the central node, called coordinator. We are well aware of the limitations posed by such topology, and in Section 2.5 we propose alternative solutions for the monitoring module. Nevertheless, a star topology is sufficient for our prototype and the beacon-enabled network also provides some benefits to the power-saving policy and reliability mechanism. Specifically, the superframe structure used in a beacon-enabled IEEE 802.15.4 network can contain an inactive part during which nodes can enter a power-save mode. If the node runs a vision algorithm requiring a continuous monitoring of the scene (e.g., the traffic flow sensor described afterward), it can just switch off its radio. Otherwise, if the vision algorithm can run sporadically (as it happens for the parking space occupancy sensor, also described later), the node can even enter the sleep mode. That is possible because MIRTES synchronizes the sensing activity with the network communication and runs the algorithm in the active part of the superframe. The sensor duty cycle is application specific and can easily be changed, even dynamically, by modifying the corresponding network parameters, i.e., the Beacon Order (BO) and the Superframe Order (SO). Specifically SO defines the duration of the active part, and must be chosen large enough to accommodate the sensor responses to the query. BO defines the beacon interval length and it must be chosen greater than or equal to SO. Once SO is fixed, the inactive part can be introduced/extended only increasing the beacon interval. Therefore the BO value is a trade-off between the power consumption and the minimum monitoring period achievable.

To cope with the communication issues posed by real-world scenarios, we extended MIRTES with mechanisms aimed at increasing the communication reliability. We implemented a very simple Forward Error Correction (FEC) mechanism to reduce the Packet Error Rate (PER). We replicate the message payload (adding a 2-byte CRC-code for each replica) to increase the probability that at least one copy will be correctly received. For most of the messages, i.e., all the messages from sensors to the coordinator, we do not need to worry about the corruption of the packet header: since they are sent using GTSs, the coordinator (which allocates GTSs) already knows to be the recipient and which node is the sender.

2.2.2. Code-Update Module

In complex IT systems code management is an essential service: as system requirements and environmental conditions change over time users may need to introduce new functions. For example, the embedded image processing algorithm may be subject to periodic revisions and improvements. In a WSN scenario the need for remote software updates is driven both by the scale of deployment, and the potential inaccessibility of sensor nodes. Moreover, since sensor networks must often operate unattended for long periods of time, the system must be reliably upgradeable without a physical intervention. The main requirements of this service are: *i)* dynamic updates of the entire code on a set of nodes; *ii)* robustness and integrity. The possibility to change every part of the code allows the users to upgrade not only the application layer, but also the lower levels, such as operating system or network stack. Robustness is critical as the ability to download new software into a node introduces a whole set of failure modes. This

FIGURE 2.3. Snapshot of the graphical user interface for setting ROIs and other parameters for a parking sensor. In this case, each ROI defines the part of the image associated with a specific parking space ID.

requirement implies the need to handle failures (or corruptions) during the update process. In our system a separate copy of the upgraded firmware is downloaded to the node in such a way that the original firmware is preserved. If a failure is detected during the upgrade process, the original firmware can be restored. Reliability is guaranteed through the use of integrity check techniques on every fragment of the firmware. Administrators can perform firmware upgrade through a remote interface, which permits to select the target node and the new firmware to upload. The network coordinator is in charge of distributing firmware fragments to the device. Nodes host a wireless boot-loader to receive the fragments, verify their correctness and store them in non-volatile memory.

2.2.3. Configuration Module

The ability to remotely view images captured from camera sensors is essential in the visual WSN context. The access to the image is needed in system setup, configuration, and maintenance phases to: *i)* check the proper functioning of the device; *ii)* properly orient the lens and capture the desired scene; *iii)* select the regions of interest in the scene. Image transfer over constrained wireless channel is a heavy task from the communication point of view. Since image sizes are relatively large compared to the MAC frame size, a fragmentation is needed to perform data transfer. In ScanTraffic we implemented a simple Stop-and-Wait protocol between the network coordinator and the devices.

Every smart camera frames a different scene. A static configuration in setup phase leads to problems of scalability and do not respond to possible environmental changes over time. Remote configuration allows administrators to set key parameters for the image processing algorithm running on each device. In ScanTraffic the following parameters can be set: the number

of the regions of interest (ROI) to be monitored, the coordinates of the points defining such regions, and the identifiers used to tag the collected data (e.g., the ID of a parking space). The system supports the setup of other parameters, e.g., to tune the algorithm sensitivity in particular light conditions. ROI configuration is performed by administrators through a remote graphical interface allowing to draw ROIs directly on the image received by the sensor (see 2.3). Parameters are saved in non-volatile memory, therefore no reconfiguration is needed when batteries are changed.

2.2.4. Communication Manager

The three service modules have different communication requirements. Specifically, both the remote configuration and the code-update services require high bandwidth (for transferring images and firmware, respectively), whereas the monitoring service requires timeliness and power management. The Communication Manager is in charge of handling this conflicting situation.

In ScanTraffic we defined two possible communication settings, corresponding to two possible states for the WSN: the running state and the maintenance state. During the running state, only the monitoring module is active and the WSN is configured to use GTSs and a superframe with inactive part. That permits the timely and power-aware communication described in Section 2.2.1. During the maintenance state the monitoring service is suspended and the WSN uses a superframe without inactive part, i.e., nodes are always active and they can send and receive packets continuously. Therefore, the remote configuration and code-update modules can quickly transfer large amounts of data (i.e., images and firmware).

We also designed a third state, called the off-duty state. It is used to temporarily "switch-off" the WSN when the ITS does not need its services (for example during the night). In this state all nodes are in sleep mode. The switch-off signal must specify the time length of the off-duty interval. Once this time elapses, the nodes automatically wake up. When specifying the time length, administrators should take into account device clock deviation: a lower value should be chosen if late wake-ups must be avoided. The resulting final state machine (FSM) is depicted in 2.1b. State transitions are caused by control messages sent from upper layers to the coordinator.

2.3. Implementation and Deployment

We implemented a prototype of the system infrastructure proposed by Ipermob and deployed it in the land-side of the Pisa International Airport (depicted in Figure 2.4). The prototype serves as a proof-of-concept for the entire system, including visual WSNs managed by ScanTraffic. The goal of such visual WSNs is to collect information about parking lot occupancy and traffic flow. For this purpose, we used the two embedded vision algorithms described in [**MMN$^+$11**]: one algorithm counts cars passing in a road section; the other algorithm detects the occupancy status of a set of parking spaces. For the sake of brevity, we use the name "flow sensor" to denote a smart camera running the former algorithm and the name "parking sensor" to denote a smart camera running the latter. As described in [**MMN$^+$11**], those algorithms achieve an overall detection rate of 95% with a false alarm rate of 0.1%.

FIGURE 2.4. A view of the Pisa International Airport land-side. We deployed flow sensors in the main intersections, and parking sensors in both the outdoor parking lot (on the right) and the indoor parking lot (on the left).

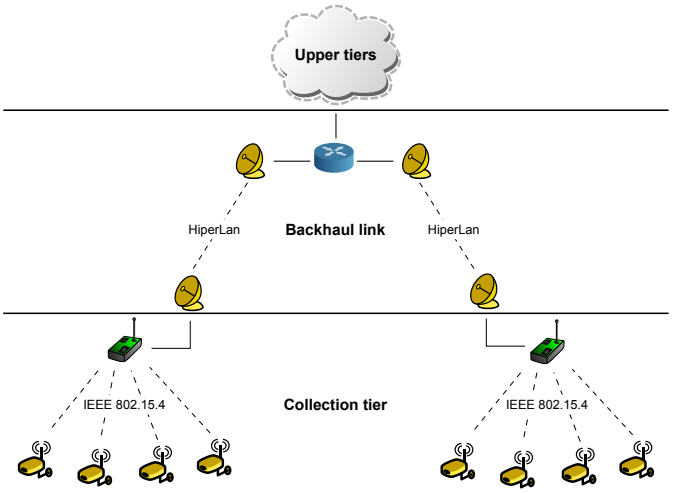
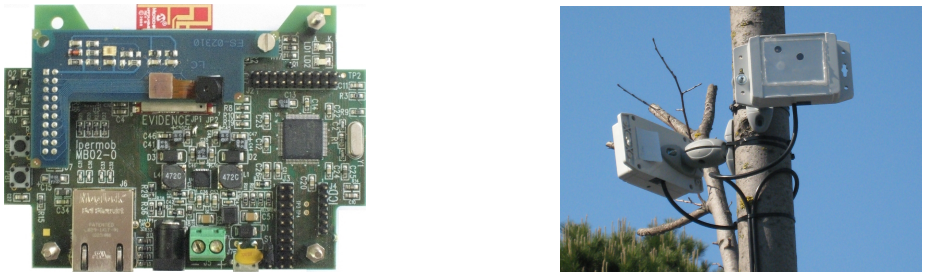


FIGURE 2.5. Network architecture of the IPERMOB prototype: backhaul links connect WSN coordinators to upper tiers.

Each visual WSN is connected to the rest of the system via a backhaul link. A special node in the WSN, the coordinator, is a gateway between the WSN and the upper layers. In the IPERMOB prototype the backhaul link is a HiperLAN link (see Figure 2.5) and the coordinator uses the UDP/IP protocol to communicate with upper tiers.

2.3.1. Hardware Description

The smart cameras used in the prototype were designed within the IPERMOB project. Specifically, two different boards were developed. The first board (Figure 2.6) is designed to be particularly low-cost and low-power. It is equipped with an 80 MHz PIC32 microcontroller with built-in 128 KB of RAM and 512 KB of ROM. It has no extra hardware for driving the camera (e.g., a frame buffer), therefore it cannot achieve a high frame rate. Nevertheless, it is still suitable to act as a parking sensor, because, in such application, the inter-arrival time of the target events



(a) (b)

FIGURE 2.6. The microcontroller-based smart camera used in the IPERMOB project. The board (a) is equipped with a PIC32 microcontroller, an IEEE 802.15.4 transceiver, a CMOS camera, and an Ethernet port used for debugging purposes. Since the device is battery-powered and communicates wirelessly, it can be easily installed almost everywhere (b).

(i.e., cars occupying or leaving a parking space) is expected to be high. In the development of the second board, power constraints have been slightly relaxed in order to explore a solution with more flexibility and processing capabilities. The second board is an FPGA-based device equipped with 32 MB of RAM. The FPGA hosts a soft-core microprocessor, thus allowing to reuse the code developed for the other board. The FPGA also drives the camera to its maximum frame (i.e., 30 fps), although the soft-core runs at only 40 MHz. This feature, together with a relatively high amount of RAM, makes the board suitable for monitoring traffic flow.

2.3.2. Software Platform

Each node runs ERIKA [**eri**], a real-time multi-tasking operating system especially designed for time-constrained embedded applications. ERIKA's priority-based scheduling allows to run the computationally intensive vision algorithms concurrently with SCANTRAFFIC without compromising its functionality. Moreover, ERIKA's implementation of the IEEE 802.15.4 standard supports the beacon-mode and the GTS feature required by the monitoring module of SCANTRAFFIC, i.e., MIRTES. Support for flow and parking sensors was easily added to the monitoring module by customizing the MIRTES virtual tables (cf. 2.2.1) with the two sensor types.

SCANTRAFFIC memory requirements for each possible combination of hardware platform and node type are shown in Table 2.1. Sensor node RAM requirements have a variable component depending on the chosen image resolution, e.g., in case of 160×120 frames, 19 KB of additional RAM is accounted.

2.3.3. Testbed Planning and Implementation

The deployment of the system in a complex urban scenario required an accurate planning, with the following objectives: *i)* minimize the number of sensing devices, keeping the number of monitored parking spaces as large as possible, *ii)* minimize the number of WSN coordinators, i.e., nodes with broadband connection to the control center; *iii)* reducing installation time; *iv)* enhancing the reuse of existing poles and supports; *v)* maximizing system performance,

TABLE 2.1. Memory requirements for each possible combination of hardware platform and node type.

Device	Node Type	RAM [KB]	ROM [KB]
PIC32	Coordinator	26.7	150.1
PIC32	Parking Sensor	24.5 + Image Size	111.6
FPGA	Flow Sensor	16.0 + Image Size	80.0

carefully choosing the positions of the sensors, avoiding non-line-of-sight communication. We deployed a total of 21 smart cameras, specifically: 14 parking sensors monitoring 83 parking spaces, and 7 flow sensors monitoring 8 traffic lanes.

The deployment of smart cameras was greatly simplified and sped up by the availability of remote configuration and code-update. The actual deployment (i.e., excluding the planning phase) was carried out by two people and it took less than three days. Most of the time was spent to install the sensors on pre-existing poles or trees. The configuration phase (including camera orientation) took just half a day. We programmed every smart camera with a special deployment firmware which constantly acquires the current camera view and sends it to a laptop connected to the Ethernet interface, thus allowing to properly orient the camera to capture the desired scene. Once the installation was completed, we remotely uploaded the operational firmware (flow or parking sensor) to each smart camera and we configured the algorithm using the remote configuration service. We could have used the configuration service also to remotely acquire the current camera view, thus avoiding the need for the procedure just described. However, since the limited bandwidth does not permit a fast image retrieval, this approach would have been inefficient, considerably slowing down the camera orientation setup.

2.4. Validation

The IPERMOB prototype served as a test-bed to validate SCANTRAFFIC design and implementation. SCANTRAFFIC real-time features (inherited from MIRTES) proved to be essential for the correctness of the information produced by the WSNs. Jitter-free periodic queries guarantee that the number of counted vehicles is related to a known time interval equal to the query period (tunable at run-time). It is therefore possible to precisely compute the traffic flow with a specific time resolution. The synchronous sensing enforced by SCANTRAFFIC permits to build a time-coherent status of the whole parking lot from the single parking spaces.

The prototype deployment showed that visual WSNs running SCANTRAFFIC can be quickly installed in a urban scenario reusing existing poles and supports, without the need for costly and time-consuming engineering works. Indeed, one unique characteristic of SCANTRAFFIC visual WSNs is that they can be easily deployed on-request where needed, and subsequently removed. Such temporary installation can be used to provide ITS services during big events (e.g., concerts, festivals, etc.) or to collect data for planning infrastructure improvements.

TABLE 2.2. Power consumption of both devices for different configurations.

Device	Node Type	Image Size	Frame Rate [fps]	$\mathbf{P}_{\max}$ [mW]	$\mathbf{P}_{\text{idle}}$ [mW]
PIC32	Parking Sensor	160×120	1	450	6
		320×240	1	462	6
FPGA	Flow Sensor	160×120	30	1230	25
		320×240	20	1230	25

2.4.1. Power Consumption

Currently ERIKA does not support the idle state for our boards and we did not implement it, since IPERMOB main goal is to prove the feasibility of using visual WSNs in ITSs, leaving energy-related issues for a possible follow-up. However, we measured the power consumption of each board in idle state (P_{idle}) and at full load (P_{max}), i.e., with the radio transmitting and both the camera and the algorithm running. Results are shown in Table 2.2. For the parking lot monitoring application, it is reasonable to assume that, with a monitoring period of one minute, the PIC32-based board powered by 4 AA batteries, i.e., with a total voltage (V_{bat}) of 6 V, will last for about 3 weeks. The camera and the algorithm can run just once a minute, therefore the board will be on for less than 2 seconds every minute (considering a frame rate of 1 fps, the start-up time, and the computation time). If we use a high beacon order, e.g., 10, and a low superframe order, e.g., 4, the radio is on for just 1 second every minute. The resulting duty cycle (α) is equal to 5% and, supposing a battery capacity (C_{bat}) of 2500 mAh the node life is:

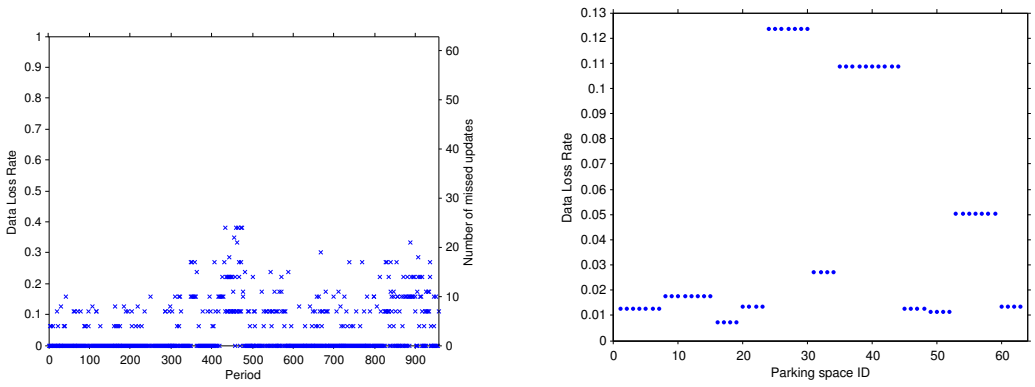
$$\frac{C_{bat} \cdot V_{bat}}{\alpha \cdot P_{max} + (1-\alpha) \cdot P_{idle}} = \frac{2500 \cdot 6}{.05 \cdot 462 + .95 \cdot 6} \left[\frac{\text{mAh} \cdot \text{V}}{\text{mW}}\right] \simeq 520.8\,[\text{h}] \tag{1}$$

On the contrary, the FPGA board, used for the vehicular flow monitoring, will last for just 11 hours. The reason is not only the higher power consumption, but primarily the duty cycle of 100% required by the flow monitoring to capture all the vehicles.

2.4.2. Data Transmission Reliability

From a communication point of view, SCANTRAFFIC must operate in a harsh environment. In public places as the Pisa International Airport the ISM 2.4 GHz band is much crowded. Moreover cars, as well as other metal objects, highly reflect electromagnetic waves and produce multipath interference whose modeling is difficult because of the rapid changes in the scenario. The adopted data protection can reduce the probability of data loss, but cannot completely avoid it. However, as long as data loss is moderate, the IPERMOB target applications can tolerate missing data.

To evaluate the data transmission reliability of the deployed prototype, we set a monitoring period of 20 seconds and, for every period, we counted the number of missing parking space and flow updates. The experiment lasted for more than 5 hours, i.e., for almost 1000 periods. 2.7



(a) (b)

FIGURE 2.7. SCANTRAFFIC prototype data transmission reliability: (a) evolution of the data loss rate during the 5-hour-long validation experiment, and (b) average data loss rate per parking space.

shows the results of such experiment performed on the biggest visual WSN deployed within the prototype, having 11 sensors monitoring a total of 63 parking spaces. Therefore, in each period, 63 updates were expected. 2.7a shows the percentage of missed updates for every period. On average, just 4.5% of the data is lost (i.e., the occupancy status of 2-3 parking spaces is not updated). 2.7b shows the loss rate per parking space. Parking spaces monitored by the same sensor have the same loss rate, because the sensor groups all the updates together in one message. For most of the sensors only 2-3% of the messages are lost, with the exception of 3 sensors whose loss rate is up to 12%. Those sensors are responsible for the data loss peaks shown in 2.7a. Being the farthest from the coordinator, their link quality can easily drop under the minimum acceptable level. The easiest solution to fix this problem is to add another coordinator nearer to them, thus creating a new WSN. A deployment in the considered area, spanning for more than $3\,000\,\mathrm{m}^2$, would benefit from the addition of a second coordinator.

2.5. Conclusions

In this chapter we presented SCANTRAFFIC, a system to collect traffic information using visual WSNs. The implemented prototype acquires information about parking lots occupancy and vehicle flows. We claim that, at least for parking monitoring, the proposed system is a good alternative to existing solutions based on scalar sensors, specifically inductive sensors. First of all, as discussed in Section 2.4, our system can be used for temporary installations thus addressing new application scenarios such as surveys for infrastructure planning and ITS support for special events. However, our system is interesting for permanent installations as well. The battery depletion problem (discussed in Section 2.4.1) can be solved using small solar panels (in our calculations an area of $0.2\,\mathrm{m}^2$ is sufficient) and still the total cost of the system will be competitive with that of solutions based on inductive sensors (which are cheaper per se, but require expensive engineering works to be embedded in the asphalt). Finally, our system has

greater potentiality with respect to existing solutions. Thanks to the smart camera versatility, our system can support new functions (e.g., vehicle classification) as well as new applications (e.g., surveillance), without any physical intervention.

ScanTraffic shows how a proper network/software design is necessary to fully exploit smart cameras features. We built ScanTraffic starting from two previous works of ours, i.e., Mirtes and the embedded vision algorithms. We successfully integrated the computation-intensive vision algorithm with Mirtes without compromising its real-time features and we added error correction techniques to Mirtes in order to address the communication issues posed by real-world scenarios. However we quickly realized that to achieve the aforementioned advantages over traditional scalar WSNs, additional services were needed, namely remote configuration and code-update. The resulting system has been described in this chapter, proving that it is suitable for real-world deployment.

However, the presented solution presents some limitations, potentially hindering the industrial impact of the results obtained so far. First, the presented systems is hardly interoperable: the configuration and the management of the sensors require custom tools, implementing application-specific protocols. Second, the star topology limits the number of nodes supported by the network. In order to cover larger areas, additional gateways must be deployed on the field, raising the operational costs and increasing the complexity of the architecture. Third, the system does not exploit the WSN potential of hosting collaborative applications and permitting in-network processing of partial records.

In the next chapter we present a possible solution for the aforementioned issues. The definition of a network architecture based on standard, Internet-based protocols solves the interoperability problem: the adoption of 6LoWPAN allows to create scalable mesh networks that are robust to points of failure, while CoAP enables the use of standard tools to perform configuration and maintenance of the system. Moreover, the automatic resource discovery mechanism and the macroprogramming paradigm simplify the management of large-scale, dynamic deployments. Finally, the support for "in-network" processing tools allows the motes to cooperate, performing aggregation and decentralized control loops.

CHAPTER 3

Programming Abstractions for the IoT

THE extensive diffusion of new generation wireless sensor networks requires the adoption of flexible mechanisms capable of abstracting the functionality of the nodes, facilitating their integration in larger computing infrastructures. This chapter presents PyoT, a framework designed to simplify the development of complex applications for the Internet of Things, coordinating the activities of groups of nodes. Adopting the macroprogramming paradigm, PyoT lets programmers focus on the application goal, hiding low-level communication details. The framework is capable of efficiently distributing the processing effort inside the network, providing support for "in-network" processing applications.

Thanks to the Internet of Things vision, a wide range of novel applications can be designed, exploiting the pervasiveness of sensor and actuator networks, and the unprecedented amount of data gathered from the physical world. However, in order to facilitate a widespread adoption of the IoT technology, mechanisms for simplifying the development of IoT applications are necessary. The macroprogramming paradigm is an effective answer to this need [**MP11**, **HM06**], allowing developers to build complex applications involving large groups of nodes without having to deal with low-level implementation details.

In order to simplify the development of such applications, a complete system architecture for easily managing networks of IoT nodes is needed. This issue is addressed by PyoT, a system for macro-programming and managing IoT-based WSNs. PyoT abstracts the WSN as a set of software objects that can be manipulated and combined in order to perform complex tasks. Specifically, PyoT allows the user to:

1. automatically discover available resources;
2. monitor sensor data;
3. handle its storage;
4. control actuators;
5. define events and the actions to be performed when they are detected;
6. interact with resources using a scripting language (macro-programming).

The high-level abstraction provided by PyoT completely hides the nodes and the network, letting users and application developers focus on the sensing and actuation capability of the system. Moreover, this approach allows for a seamless federation of multiple WSNs as long as such WSNs are based on the IoT protocols (i.e., 6LoWPAN and CoAP). The development of applications involving large groups of nodes is further supported by the distributed and scalable architecture of PyoT, which, for example, allows application developers to parallelize operations on sets of nodes. Moreover, PyoT integrates T-Res [**APP13**], a framework enabling "in-network processing" in

IoT-based WSNs, i.e., the possibility to run part of the application logic directly on sensor/actuator nodes. PyoT also provides a storage mechanism that is used for both caching purposes and long term storage. Sensor data caching is performed automatically by PyoT in order to reduce the number of network interactions with IoT nodes thus extending their lifetime (nodes are usually battery-powered and network communication is their main source of energy consumption) and improving application performance (radio duty-cycling policies commonly used by IoT nodes lead to slow network communication).

Finally, PyoT does not define a custom language for its macro-programming mechanism; on the contrary, it uses a popular high-level programming language (i.e., Python) that features a large set of libraries, including scientific ones (e.g., SciPy, NumPy, matplotlib, pandas, etc.). The use of a high-level language makes it possible to quickly prototype complex applications in a few lines of code. Moreover, PyoT's macro-programming mechanism is not solely targeted to application development but it can also be used to control IoT nodes "on the fly" by means of interactive shells.

The rest of the chapter is organized as follows: Section 3.1 provides a general overview of the system. Section 3.2 describes the system architecture and discusses the most relevant design choices; Section 3.3 presents the technical implementation details of the system and the support for T-Res "in-network" processing applications; Section 3.4 discusses the experimental validation of the system; Section 3.5 presents related work and, finally, Section 3.6 provides concluding remarks.

3.1. System overview

PyoT is designed to work with any IoT-based WSN, i.e., any WSN that uses IoT protocols and abstracts actuators and sensors as CoAP resources. PyoT hides the complexity of the network by presenting it as a set of software objects. Such objects are provided with a collection of attributes and support a set of operations. Attributes provide information about the resource, such as its type (e.g., actuator or sensor), its purpose (e.g., temperature monitoring, fan control, etc.), its location, etc. The following is a list of typical operations that can be performed on resource objects: querying a resource for its current or past values, modifying a resource, monitoring a resource for new values, and defining an event handler associated to notifications from a sensor. Some of these operations translate into actual interactions with IoT nodes (i.e., CoAP requests) whereas other exploit internal services provided by PyoT (e.g., historical data recording). Some complex operations, like event detection, may also require both network interactions and use of PyoT's internal services.

PyoT provides two ways for interacting with resources: a convenient web-based user interface (WUI) and a powerful shell interface, allowing experienced users to use the macro-programming mechanism interactively. The WUI is designed as a virtual control room that allows for easy execution of basic operations such as resource listing, sensor monitoring, actuator control, event detection and reaction, and access to historical data. Figure 3.1 shows a screenshot of PyoT's WUI. The macro-programming mechanism allows the use of a high-level language, specifically Python, for defining more complex operations (e.g., operations involving group of sensors or

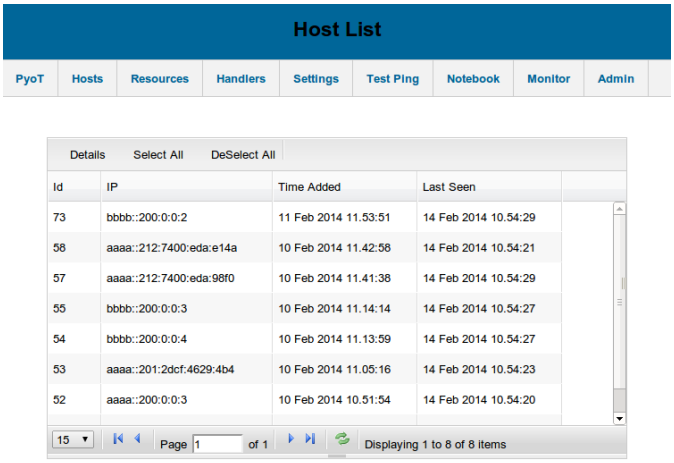


FIGURE 3.1. PyoT's web-based user interface allowing for easy execution of basic operations such as sensor monitoring, actuator control, event detection and reaction, and access to historical data.

having an elaborate logic). PyoT provides a set of Python APIs for interacting with resources, which are abstracted as Python objects. Using the macro-programming mechanism, the user can programmatically access all the basic operations provided by PyoT's WUI and combine them to build IoT applications.

Figure 3.2 shows an example of WSN macro-programming with PyoT. The presented code activates a fan depending on the average value of all the temperature sensors available in the WSN. In line 1, the list of all the temperature sensors is obtained applying a filter to the available resources. In line 2, the current value for each sensor is retrieved calling the `GET()` method, which internally translates into a CoAP GET request for the resource. Finally, in lines 3–5, the average temperature is computed and, in case it exceeds 24°C, the fan is activated. The `GET` and `PUT` methods used in the previous example are synchronous, i.e., every time they are performed, the execution blocks until the corresponding network operation on the target resource is completed. This means that the `GET` requests in the for loop in line 2 are not executed in parallel, possibly slowing down the execution. Therefore, PyoT also provides an asynchronous version of methods involving network operations. Asynchronous execution makes it possible to run different operations in parallel, allowing, for example, the concurrent execution of GET requests on a group of resources. Figure 3.3 shows the parallel version of the previous example. Line 2 calls the `asyncGET()` method on a list of temperature sensor objects. The method (immediately) returns an object containing the status of the requests. Then, in line 3, when the whole transaction is complete, the list of results is created. The difference with the previous example is that, in this case, the GET requests are simultaneously activated, while in the previous example a GET request is activated only when the previous one has completed its execution. The two examples show how resources can be handled as a group and how PyoT hides the communication details to the developer.

```
temps = Resource.objects.filter(title='temp')
results = [temp.GET() for temp in temps]
avg = sum (results) / len(results)
if avg > 24:
  Resource.objects.get(title='fan').PUT('on')
```

FIGURE 3.2. Example of WSN macro-programming with PyoT. The code activates a fan depending on the average value of all the temperature sensors available in the WSN.

```
temps = Resource.objects.filter(title='temp')
requests = temps.asyncGET()
results = requests.wait()
avg = sum (results) / len (results)
if avg > 24:
  Resource.objects.get(title='fan').PUT('on')
```

FIGURE 3.3. Example of usage of asynchronous methods to parallelize network operations.

PyoT exploits CoAP's RESTful interface to interact with nodes. Basic applications can consider only the sensing and actuating capabilities of motes, shared with the external world through URIs. Anyway, when programming a WSN, other aspects of the network are often to be considered, such as the physical location of the nodes, the relationships with their neighbors, and their connectivity. While some of these information can be made available as meta-resources, others can be inferred performing a global analysis of nodes' local data. For example, PyoT can reconstruct the RPL Directed Acyclic Graph (DAG) by querying the nodes for their local routing information. The knowledge of the graph can be used to optimize other applications, e.g, when performing "in-network" aggregation it may be useful to install the processing function as close as possible to the data sources, thus minimizing the average number of hops required to reach the aggregating node. Figure 3.4 shows a screenshot of a PyoT application building the representation of a RPL DAG.

3.2. Architecture

As shown in Figure 3.5, PyoT is a distributed system having five main components: a Virtual Control Room (VCR), a Shell interface, a Storage Element (SE), a message queue, and one or more PyoT Worker Nodes (PWN). The VCR provides the WUI previously described. PyoT adopts a Data-Centric design. The SE manages the system status: it indexes the resources available in the sensor networks, keeps track of the subscriptions currently active, and contains the definitions of the events currently monitored, as well as the associated actions to be performed when the events are detected. The SE is also used to provide data persistence, storing information such as event notifications or sensor readings.

Each PWN manages one IoT-based WSN, providing a set of workers, i.e., processes that can perform generic tasks and communication activities with IoT nodes. The PWN keeps track of

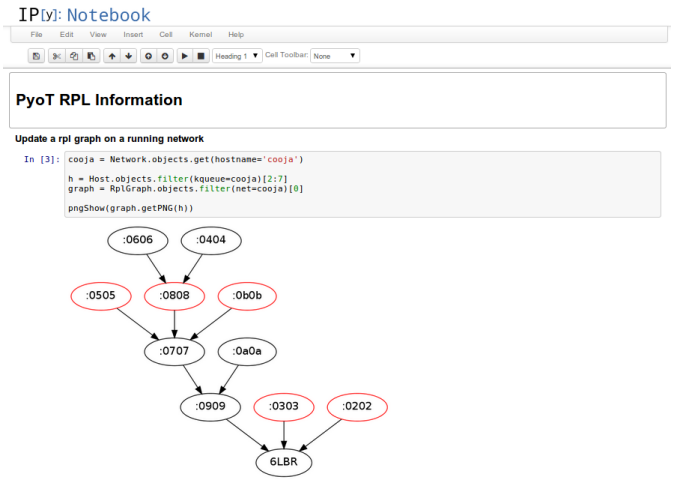


FIGURE 3.4. PyoT dynamically displaying the RPL graph of a network.

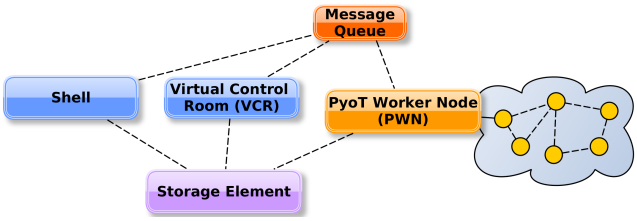


FIGURE 3.5. The architectural elements of PyoT and their connections.

the nodes and of the resources available in the WSN and updates the related information stored in the SE. Moreover, it directly manages the sensor data collection and the event detection. If the user subscribes to some sensor data, the PWN asynchronously monitors the proper resources for changes and adds new sensor values to the SE every time they become available. Events are handled in a similar way at the PWN level. When the user defines a new event, the PWN starts monitoring the proper resources. Every time the WSN generates a resource-changed notification, the PWN checks whether the event has occurred, and, in such a case, it adds an event notification to the SE and performs the actions defined by the user, if any. An action can be a simple actuator control signal or an elaborate set of operations defined by a macroprogramming script. A PyoT installation can have more than one PWN. The typical configuration is to have a PWN for every WSN, preferably hosted in a full-fledged device directly connected to the 6LoWPAN border router or on the border router itself if allowed by its computing power. However, it is also possible to have a single PWN managing multiple WSNs, as well as multiple PWNs managing a single WSN (useful when the WSN has multiple border routers).

In order to save energy in the WSN and avoid flooding it with too many messages, PyoT implements a basic form of caching: whenever a GET operation is requested by the user, the PWN queries the SE for recent values before creating the network request. If a fresh value is

available, it is transparently returned to the user. Moreover, the Pwn can limit the rate of network requests in order to avoid congestion on the WSN.

3.3. Implementation

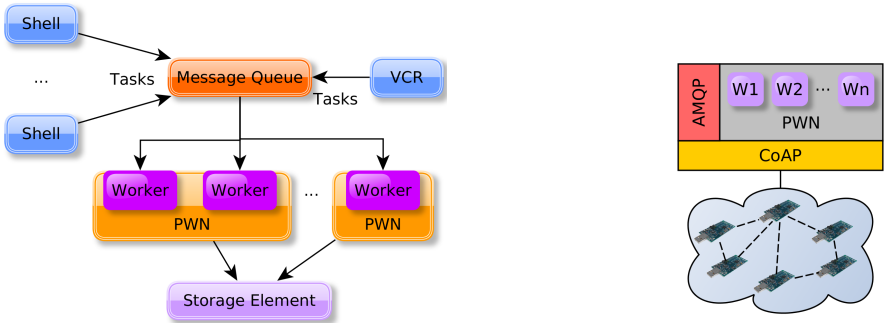
In this section we detail our implementation of the architectural modules. The core component of the system is the Se which is implemented using Django's Object-Relational Mapping (ORM)[1]. PyoT defines models representing networks, hosts, resources, messages. Those models are then mapped by the ORM into database objects. Moreover models provide a set of methods abstracting the interaction with network resources, which can be accessed through the Shell. This can be realized by any Python shell importing Pyot's modules. The VCR also provides a Web-based version of the shell, using IPython Notebooks [**PG07**]. The system supports multiple, distributed instances of the shell allowing multiple users to work concurrently on the same physical infrastructure.

Pwns are responsible for indexing the resources currently available in the network. More specifically, they run a CoAP Resource Directory (RD) server [**SBK13**]. Nodes periodically send a message to the RD. If the node is not known to the Pwn, the latter performs resource discovery by querying the node's "*/.well-known/core*" resource. A periodic task is in charge of removing inactive nodes from the Se.

PyoT is designed and implemented to be scalable. The basic operations that can be performed on network resources (e.g., resource retrieval, resource monitoring, etc.) are implemented as tasks that can be allocated to different WSN, each one typically managing a different WSN. PyoT task distribution system is depicted in Figure 3.6a. Tasks are instantiated by the shell or the WUI, as a result of user's actions or automatic event handling, and inserted in a message queue to which Pwns are subscribed. Each Pwn provides a set of workers, i.e., processes that can perform tasks. When a new task is inserted in the queue, the proper Pwn is selected and, if there are available workers, the task is immediately dispatched; otherwise, the task is enqueued and dispatched whenever a worker becomes available. When the task is completed, its results are stored in the SE. Typically, Pwn are selected for task allocation depending on the WSN they manage. Tasks are basic operations on a single resource, and, therefore, it is convenient to allocate them to the Pwn that manages the WSN containing the target resource. However, other allocation policies taking into account, for example, load balancing, priorities, or QoS are possible. PyoT's task distribution system is implemented using Celery[2], a distributed task queue, and RabbitMQ as the message broker. RabbitMQ uses AMQP [**Vin06**], an open standard application layer protocol for message-oriented middleware. Figure 3.6b shows the structure of a Pwn and its relations with the messaging and communication layers provided by the operating system. Tasks are received through the AMQP messaging layer, while worker processes communicate with sensor nodes using a CoAP layer. The AMQP protocol is designed to provide scalability and interoperability. Also the VCR is implemented to be scalable. Indeed, the use

[1]https://www.djangoproject.com/

[2]http://www.celeryproject.org/



(a) (b)

FIGURE 3.6. (a) PyoT's task distribution system. Tasks (i.e., basic operations on resources) are instantiated by the shell or the VCR, inserted in a message queue, and subsequently allocated to PWN's workers. (b) Detailed structure of a PWN.

of web technologies allow the VCR to be distributed in the Cloud. The same applies to the SE, which has been implemented as a mySQL database that can be managed by a distributed database management system.

3.3.1. T-Res support

T-Res [**APP13**] is a programming abstraction for IoT-based WSNs. T-Res allows developers to create simple "tasks", called T-Res tasks, that can be installed on IoT-based WSNs at run-time. Using CoAP procedures, T-Res allows configuring the data-processing performed by sensor nodes and the interaction among them, thus enabling "in-network" applications. A T-Res task is defined by its input sources, the data-processing it performs, and the destination of its output. Both the input sources and the output destination must be CoAP resources and are specified by means of their URIs.

PyoT provides a convenient, high-level interface for T-Res. Figure 3.7 shows the use of PyoT to define, deploy and execute a simple task inside a 6LoWPAN network. The first line simply creates a T-Res processing function object by taking as input the source file containing the script to be executed by a T-Res node. A processing function can also be created directly from a string containing the source code. In both cases when the object is created the source code syntax is validated. Moreover the function is saved in a library of functions and can be reused by other tasks. Lines 2–3 select the input and output resources involved in the task. Note that the list of resources is not explicitly defined by the user, but is the result of a filtering operation on the resources currently available.

Line 4 creates a T-Res Task object using the previously defined processing function, input, and output resources. PyoT supports the creation of a processing function by checking the source code for compilation errors. However it may be important to validate the functional correctness of the task before the deployment on real nodes. T-Res tasks can only be installed on special nodes, called T-Res nodes, providing the required runtime environment. PyoT supports an emulation

```
1 task = TResPF(open('avg.py'), 'avg')
2 inputRes = Resource.objects.filter(title='Temp')
3 outputRes = Resource.objects.get(title='Fan')
4 tresTask = TResTask(TresPf=task, inputS=inputRes,
5                     output=outputRes)
6 tresTask.emulate()
7 tresNode = Resource.objects.get(title='TRes')
8 tresTask.deployOn(tresNode)
9 tresTask.start()
```

FIGURE 3.7. PyoT script for defining, testing, and deploying a T-Res task. The processing function takes as input a set of temperature resources and turns a fan actuator on or off depending on the average temperature value.

mode, as shown in line 6, in which the functions associated with a T-Res node are realized by a PWN. When the emulation starts the PWN subscribes to the real input resources, executes the processing function, and sends the output to the real output resource. The output of the computation can be subsequently retrieved to check if the processing function works as expected. When the function is functionally verified it can be deployed "in network" on a real T-Res node. Lines 7–9 show the selection of a T-Res resource, the deployment, and the activation of the task.

3.4. Performance evaluation

This section is aimed at validating the effectiveness of the architecture of PyoT, described in Section 3.2 and implemented as discussed in Section 3.3. In all the experiments our nodes are programmed with the Contiki Operating System [**DGV04**], which supports 6LoWPAN specifications for the Network layer, and CoAP draft specifications for the application layer [**KDD11**]. We performed tests both in an emulated environment and on a real testbed in our laboratory with the goal of measuring:

1. the execution time of a macroprogramming script;
2. the scalability of the architecture;
3. the overhead of the task distribution mechanism.

Figure 3.8 shows the reference network topologies we used in our simulations. The three topologies (A, B, C) have been realized building a binary tree starting form the border router. Therefore the topologies include both a variable number of nodes and a variable number of hops to reach them. As an example, topology C includes 14 nodes, 8 of which can be reached within three hops from the border router. Table 3.1 shows the common configuration parameters used for Cooja simulations.

3.4.1. Execution time

In the first experiment a laptop runs both a Cooja simulation, implementing the topologies previously described, and a PWN instance. The PWN is configured to support up to 14 concurrent worker processes, thus allowing a complete parallelization of the macroprogramming operation. In order to reduce the effect of interfering routing packets we first tested the system with a static

TABLE 3.1. Simulation Configuration

Nodes	1 6LoWPAN Border Router and 2, 6, 14 CoAP servers.
Radio Medium	Unit Disk Graph Medium (UDGM)
PHY and MAC	IEEE 802.15.4 with CSMA
Duty Cycling	NullRDC
Routing	Static and RPL

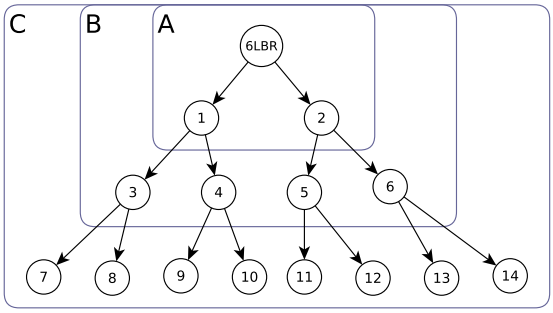


FIGURE 3.8. Simulated network topologies.

routing configuration. The SE (a mySQL server) and the Message Queue are hosted by the same laptop running the PWN.

In the first experiment we measured the execution time of a PyoT script, i.e., the time required to complete a macroprogramming transaction on a set of IoT nodes. In this experiment a script performing a `PUT` operation, toggling a led on every node included in the network topology, is executed from a PyoT Shell. The measured time is the one required to complete the whole transaction. CoAP requests are of type *confirmable*, that means that a single operation is considered successful when the confirmation message is received at application level. I the acknowledgment message is not received, the request is retransmitted after a default timeout and exponential back-off, in compliance with the IETF CoAP draft.

We measured the performance varying the network topology and adopting different task scheduling policies. The first policy is completely synchronous, as presented in Figure 3.2: a new `PUT` task is dispatched only when the result of the previous operation is available. The second policy is asynchronous. As already discussed in Sec. 3.1 tasks are dispatched to a set of concurrent worker processes on the PWN. The workers generate the CoAP request addressed to the correct node and, when the confirmation message is received, update the SE. As shown in Figure 3.9 the performance of the asynchronous policy degrades as the number of the nodes grows. Indeed, the concurrent workers initiate a large number of CoAP transactions at the same time, causing congestion at the level of the border router and collisions at MAC level: thus the number of re-transmissions, required by the CoAP client, offsets the benefit coming from parallelization.

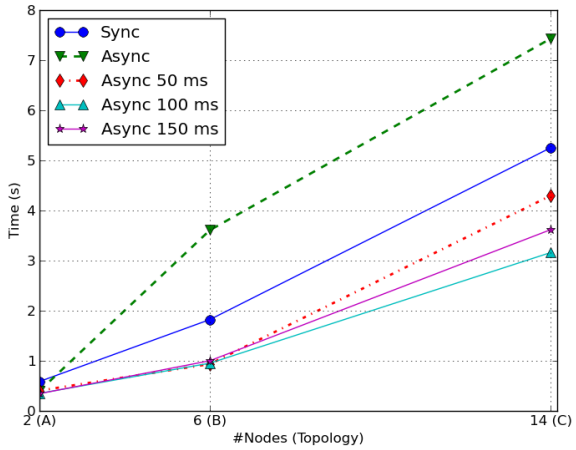


FIGURE 3.9. Macroprogramming time with PyoT, varying the topology (and the number of nodes) and the PWN transmission policy.

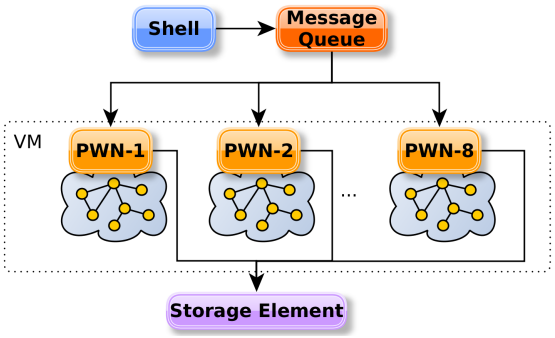


FIGURE 3.10. Experimental setup for the scalability test.

To overcome this problem a basic traffic shaping mechanism has been implemented, introducing a fixed delay between two consecutive CoAP requests, while keeping the asynchronous semantic.

The rate-limiting mechanism is implemented on the PWN and can be configured defining the *sleep time* α, i.e. the time interval between two consecutive requests. Considering a set of worker processes $\{W_i, i = 1 \dots n\}$ each process W_i waits $i \times \alpha$ ms before transmitting the CoAP message. Figure 3.9 demonstrates the benefit of adopting the rate-limiting mechanism, comparing the performance of the system with three different configurations of the *sleep time*. The best result is obtained using a *sleep time* of 100 ms. This traffic shaping technique is not intended to provide an optimal solution but to demonstrate the flexibility of PyoT's macroprogramming engine. Indeed, more complex mechanisms could be implemented in order to minimize the probability of collision at MAC level (e.g, scheduling the packet transmission taking into account the current network topology).

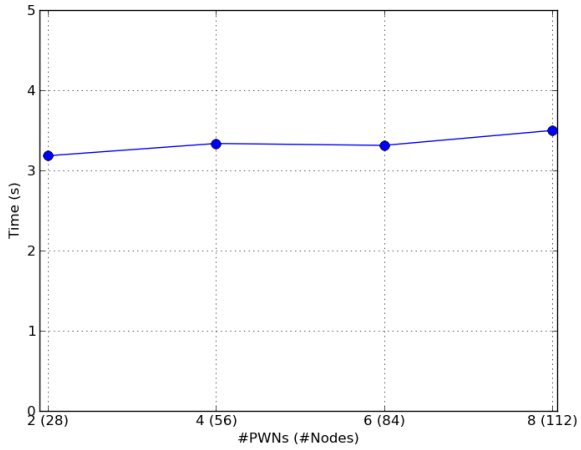


FIGURE 3.11. Scalability performance results, varying the number of networks and nodes subject to concurrent macroprogramming.

3.4.2. Scalability

The second set of experiment is intended to validate the scalability of PyoT's architecture. To this end a different setup has been built, as shown in Figure 3.10, with the goal of emulating a medium-scale WSN infrastructure, i.e., the federation of multiple independent networks. In this scenario each WSN is managed by a different PWN. The infrastructure has been implemented deploying up to eight virtual machines, each one running an instance of the Cooja emulator and a PWN. Each PWN implements the traffic shaping policy previously described, with the *sleep time* parameter set to 100 ms. Again, the SE and the Message Queue are hosted on another server machine. The Cooja instances emulate networks having a topology of type C. Therefore, the maximum number of emulated nodes sums to 122, with 8 6LoWPAN border routers and 112 CoAP server nodes.

We considered a macroprogramming operation involving all the nodes of the emulated WSNs and we analyzed its execution time varying the number of WSNs. Moreover, we repeated each experiment twice: once using static routing and the other time using RPL routing, thus providing a more realistic scenario in which the communication is affected by routing interference. Specifically, we considered the common case of networks in a steady state, that is networks in which the RPL DAG has already been created (and, therefore, the rate of RPL control message transmissions is minimum).

The results of the experiments are shown in Figure 3.11. In the case of networks with static routing it is evident that the cost of en-queuing the tasks and allocating them to multiple worker nodes is negligible when compared to the time spent for completing a macroprogramming operation on a single network. In other words, the time spent to toggle the actuators on two or eight different networks is almost equal. On the contrary, the performance for the case of RPL routing degrades increasing the number of PWNs. However, that is due to a higher probability of

TABLE 3.2. Execution time (in s) of different network operations, performed by the PyoT Shell and by a PWN

	CoAP Put	CoAP Discovery	T-Res Deployment
PyoT Shell	0.281	0.386	3.716
PWN Local	0.049	0.143	2.859

collisions at MAC level of CoAP requests with RPL control packets. The probability grows with the overall number of nodes involved. However this effect has a tolerable impact on the general performance and may be mitigated modifying the parameters of RPL routing, thus reducing the number of interfering packets.

3.4.3. PyoT overhead

The third test is targeted at evaluating the overhead of PyoT's Shell programming in terms of execution time. In this experiment we measured the time needed to perform three different network operations on a single node. The tasks are first executed from a PyoT Shell: in this case each task is instantiated, enqueued in the Message Queue, and sent to the right PWN. Then the task is executed and its result is sent back to the SE. As previously discussed the task distribution system allows the management of multiple IoT networks and the macroprogramming feature of PyoT. However, the additional communication introduces an overhead. Differently, in the second case, the request is generated locally on the PWN. We performed the experiment with three different network operations. The first is a simple CoAP Put request. The second is a CoAP resource discovery, i.e., a Get request on the "`/.well-known/core`" resource, having a larger payload and requiring multiple message transmissions. The third one corresponds to a T-Res task deployment (as discussed in Sec. 3.3.1). The operation performed by the PWN can be decomposed in the following steps: *i)* retrieve the source file from the SE; *ii)* compile it into Python bytecode; and *iii)* create the task on the T-Res node (multiple CoAP requests to upload the bytecode and to create the input and output resources). As shown in Table 3.2 the overhead is not negligible. However, in percentage terms, it decreases when the complexity of the single operation grows and it is well compensated by the added value of a completely distributed system. Specifically, using PyoT's Shell the system becomes more scalable and provides additional advanced services, such as the storage of network operation results and possibility to remotely monitor tasks execution.

3.4.4. Real world implementation

The last experiment investigates the performance of PyoT's macroprogramming with real IoT nodes. Our laboratory testbed is composed by an heterogeneous set of WSN nodes, including WiSMote, TelosB and Seed-Eye boards. The nodes execute several CoAP servers, sharing sensors and actuators, and are connected to the backbone network using 6LoWPAN and RPL. The border router is implemented on a TelosB mote, physically plugged into a Raspberry Pi board. The latter, running the Raspbian embedded Linux distribution and a PWN instance, directly connects to the Message Queue and the SE. The experiment has been realized, as before,

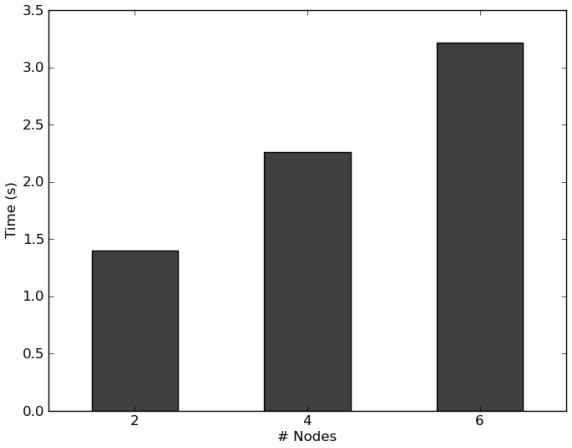


FIGURE 3.12. Macroprogramming time on a real testbed, with a variable number of IoT nodes.

computing the time necessary to perform a CoAP request on a variable set of nodes, executing a macroprogramming script from PyoT's Shell. The PWN implements the asynchronous transmission policy with the traffic shaping mechanism and a *sleep time* of 200 ms. The results, shown in Figure 3.12, exhibit a performance reduction with respect to the time experienced on the first experiment, when the PWN was implemented on a laptop. The gap is mainly due to the reduced computational power of the PWN, implemented on a Raspberry Pi. However it may be reduced implementing the PWN on a more powerful architecture, e.g., on a multi-core embedded system.

3.5. Related work

Programming a WSN is a complex task, requiring knowledge in the fields of embedded systems, distributed computing, and wireless networking. A large number of programming abstractions have been proposed in the literature with the goal of helping the development of WSN applications. Some of these abstractions focus on node-level behavior [**HAAIKR11**]. Differently, macroprogramming abstractions allow developers to write a high level program that captures the operation of the sensor network as a whole. Database abstractions [**MFHH05**, **YG02**] allow the user to describe sensor data of interest using SQL-like languages, providing a high level interface to simplify data collection. However they are not flexible enough to support general purpose applications with custom logic. Other abstractions focus on how to specify the behavior of group of nodes, defined either logically [**MP06**] or physically [**WSBC04**]. Regiment [**NMW07**] allows the programmer to view the network as a set of spatially-distributed data streams. Those streams can be manipulated in groups, defined either by topological or geographic relationships between nodes. The "deglobalization process" translates a network-wide representation of the program into node-level code.

TABLE 3.3. Comparison of PyoT and T-Res with other abstractions.

Abstraction	PyoT + T-Res	Actinium	Ecocast	Regiment	Nano-CF
Interactive Programming Support	Yes	Yes	Yes	No	No
General Purpose Programming Language	Yes	Yes	Yes	No	No
Standard Communication Protocols	Yes	Yes	No	No	No
Scalable Architecture	Yes	No	No	No	No
In-Network Processing	Yes	No	Yes	Yes	Yes
Concurrent Applications Sharing Nodes	Yes	Yes	No	No	Yes

Kovatsch et al. [**KLD12**] suggest to move the application logic from the firmware to the cloud. In the proposed architecture, the application logic runs only on a central back-end server, while IoT nodes simply provide access to their sensors and actuators through a CoAP server. While such an approach allows for easy application development, it sacrifices direct communications among IoT nodes, does not provide "in-network" aggregation capabilities and lacks of scalability. Compared to Actinium, PyoT provides a completely distributed architecture, where the logic is partially executed from the shells and the worker nodes. Our approach enhances the system scalability and in case of event detection applications it responds faster as the events are processed in a decentralized way. Moreover PyoT combines with T-Res to provide "in network" processing.

Ecocast [**TLCC11**] presents a shell-based macroprogramming framework where group of nodes can be abstracted as objects in a high-level scripting language. It allows dynamic reprogramming of sensor nodes with native code and supports parallel semantic and job control features. Nonetheless it makes use of non-standard network protocols and is not designed to inter-operate with other systems.

Nano-CF [**GKP+11**] defines a macroprogramming framework supporting in-network processing and multiple concurrent applications sharing the same sensing infrastructure. Moreover it optimizes the network utilization batching the computation and radio messages transmission. However, the user is required to explicitly define the list of nodes to fulfill a service and the framework does not support decentralized control applications.

Compared to other solutions, PyoT provides an interactive programming framework, allowing to manage generic, multi-purpose IoT networks interactively. Moreover it does not require any specific runtime support on the nodes, except when using the T-Res extension. Instead of using custom programming interfaces we choose to communicate with the nodes through a standard RESTful interface. This way, even when PyoT is executing, the nodes can still be used by concurrent applications. Moreover PyoT is based on a general purpose programming language. As such it has a soft learning curve, and allows a deep integration with existing external libraries. Table 3.3 summarizes some of the important differences between the integration of PyoT and T-Res and other abstractions.

3.6. Conclusions

In this chapter we presented PyoT, a macroprogramming framework for IoT applications. PyoT facilitates the development of distributed applications for IoT-based networks. The experimental validation demonstrated, through simulation and a real implementation, the scalability and flexibility of our architecture. The execution time overhead is compensated by the advanced features of group programming and task distribution. PyoT can be easily extended to support custom, highly optimized application logic, improving the efficiency of concurrent interaction with many WSN nodes. In the next chapter we will describe two novel IoT applications where the features of PyoT have been successfully adopted. The first application presents an integration of RFID technologies in the IoT world, and exploits PyoT's automatic event handling feature to perform authorization and access control in a smart factory environment. The second application introduces an advanced middleware for IoT-based Intelligent Transportation System where PyoT have been used to support in-network processing and advanced configuration of Smart Cameras.

CHAPTER 4

Applications for the Internet of Things

IN this chapter two original contributions are presented. The first part of the chapter proposes and discusses the integration of WSN and RFID technologies in the IoT scenario. The proposed approach is based on the REST paradigm, thanks to which the two technologies can be seamless integrated by representing sensors, actuators and RFID-related data as network resources. The integration approach is detailed for the Smart Factory use case: we present an advanced IoT-based WSN and RFID integrated solution aiming at improving safety in industrial plants. The developed system can guarantee a safe access to dangerous areas in which safety equipments are required.

The second part of the chapter introduces the ICSI M2M Middleware. Intelligent Transportation Systems are among the most promising application domains for the Internet of Things, being inherently large-scale and composed by heterogeneous systems. The ICSI M2M Middleware is a software capable of supporting standard Machine-to-Machine communication, while tackling with the dynamic nature of resource-constrained devices and networks. The RESTful interface of the ICSI M2M Middleware provides a way to dynamically reconfigure distributed sensing applications, while the support for in-network aggregation guarantees an efficient utilization of network resources. Preliminary experimental results indicate that the ICSI M2M Middleware is a suitable solution for the ITS use-case of a visual sensor network of resource-constrained devices.

The rest of the chapter is organized as follows. Section 4.1 introduces the RFID technologies and the application scenario, while Section 4.1.1 describes the state-of-the-art in WSN and RFID integration by considering several application domains. In Section 4.1.2 the IoT-based WSN and RFID integration is presented. Section 4.1.3 details the proposed integration solution for the Smart Factory use case, along with hardware and software implementation details. The performance of the developed application in terms of accuracy and response times is presented in Section 4.1.4. In Section 4.2 we introduce the ITS application domain. Section 4.2.1 presents the reference M2M architecture, while Section 4.2.2 details the ICSI use case by explaining the role of the ICSI M2M Middleware. In Section 4.2.3 the middleware architecture is presented, while in Section 4.2.4 a preliminary performance assessment is presented. Finally, Section offers concluding remarks.

4.1. Smart Factories

RFID [**KKFS09**] is a low-cost and low-power technology mainly characterized by passive devices, i.e., tags, which are able to send information when powered by electromagnetic fields generated by an RFID reader. It is characterized by a short-range radio technology and it is

mainly used for object identification [**BT06**] and tracking [**WPA07**]. Differently, WSNs are composed by low-power embedded devices characterized by reduced computational capabilities that actively communicate among them to perform distributed tasks. The transmission range of WSN devices is in the order of a hundred meters, and they are mainly used for real-time environmental monitoring [**LWS04**], tracking [**ZLZ+08**], and localization purposes [**WZWX11**].

The advantages provided by both technologies promote the design of integrated solutions in which the outstanding pervasiveness of RFIDs and the advanced sensing and communication features of WSNs are combined together to obtain pervasive and configurable resources in a worldwide network of objects.

Though the integration between RFID and WSN technologies has already been proposed in the literature, the problem has been mainly addressed from an hardware point of view, proposing custom application layers enabling the communication among devices. In this chapter we propose a WSN and RFID integration according to the IoT vision. The integration is proposed both at hardware and logic levels by discussing the use of IoT protocols at network and application layers. Moreover, the proposed approach is implemented, and assessed for the Smart Factory use case by developing an advanced safety system able to guarantee a safe access to dangerous areas in which safety equipments are required.

4.1.1. Related work

In recent years an increasing number of research activities proposed the integration of WSN and RFID systems with the aim of providing new services or improving already available applications [**LBS08**].

As already mentioned, research efforts mainly focused on the hardware aspects of WSN and RFID integration. From the hardware point of view the integration between WSN and RFID can be reached either by integrating RFID tags or RFID readers on WSN nodes. The former case consists in extending the sensing capabilities of a WSN node with those provided by RFID tags. In this scenario, the RFID tag can be used to provide location-aware services. In another notable application the RFID tag is used as an interface to wake up the WSN node from a power-saving state. In this context the RFID reader is used to trigger the activation of WSN nodes, enabling them to communicate through different channels, thus minimizing the time spent in passive listening mode.

A first example of RFID and WSN integration is presented by Chen [**Che**] where a wireless localization system for monitoring children location in theme parks is proposed. In the work, the integration is reached by installing an RFID reader on each WSN node. The resulting system is able to estimate the child position with a maximum error of 3 meters. The work presented by Xiong et al. in [**XSSG11**] can be classified in the same application scenario. In their paper a grid of RFID tags is used to enhance the positioning accuracy reached by state-of-the-art WSN localization algorithms based on the the received signal strength indicator. In the latter case the integration of both technologies is reached again by installing RFID readers on WSN nodes.

RFID readers usually combine transmission and reception (to and from the tags) functions in a single physical device. In [**DRC+10**] a different device, called RFID Listener, is introduced.

The RFID Listener is dedicated to the reception functions and is targeted to localization applications. The easier integration of RFID listeners on low-cost and low-power WSN nodes allows to realize an architecture with a single transmitter and multiple listeners, enabling a denser deployment and increasing the localization accuracy.

In the above mentioned works, RFIDs are mainly used for implementing a coarse grain localization while trying to optimize the power consumption of each WSN unit. Looking at the power consumption optimization scenario, Jurdak et al. proposed in [**JRO08**] a low-cost system making use of IEEE 802.15.4 transceiver as a fake RFID tag reader. In particular, their system transmits, through the installed IEEE 802.15.4 transceiver, the electromagnetic energy necessary for triggering a tag and indirectly for waking up the associated WSN node.

More recently, integrated WSN and RFID solutions have been proposed in other application domains. In [**XW08**] Xiaoguang and Wei proposed the combined use of WSN and RFID for the development of a smart warehouse management system. In the proposed application several possible network architectures are analyzed and discussed looking at the different trade-offs between system reliability and deployment costs. The use of WSN and RFID in a smart home scenario is proposed by Hussain et al. in [**HSM09**]. Leveraging on both WSN and RFID advantages the authors aim at assisting elderly people by tracking their movements while providing personalized services to increase their comfort. In [**NS09**] an Intelligent Transportation System is considered. In their work Nasir and Soong propose to acquire pollutant emission levels gathered by hybrid WSN and RFID sensors installed on the vehicles. The pollutant levels are acquired only when an embedded RFID tag receives enough energy to wake up the sensors. The developed solution permits on the one hand to acquire environmental data, and on the other hand, to correlate the pollutant emission level with a car identification number, i.e., the car license plate.

Although the above mentioned works greatly contribute to the integration of WSN and RFID technologies, none of them addresses the problem of a seamless integration in the IoT scenario and of the interoperability with devices compliant with Internet protocols.

4.1.2. IoT-based WSN and RFID Integration

Figure 4.1 depicts a 6LoWPAN network with its main components: *i)* Host node (H); *ii)* 6LoWPAN Router (6LR); *iii)* 6LoWPAN Border Router (6LBR). The H node is a simple node of the network and does not provide any forwarding and routing service. The 6LR is a node with forwarding and routing capabilities, while the 6LBR is in charge of connecting each subnet to Internet by translating 6LoWPAN into IPv6 packets and vice-versa.

The possibility of representing sensors, actuators and other sources of information as generic resources identified by a global URI allows to abstract the physical components of the system with a common operation logic. Considering a WSN and RFID integrated system, in which both RFID readers and RFID tags are integrated in hardware with WSN nodes, we introduce two new nodes to the architecture presented in Figure 4.1: *i)* Host Reader (HR), a WSN node integrated with a RFID reader; *ii)* Host Tag (HT), a WSN node integrated with a RFID tag. The two nodes and their web interfaces are depicted in Figure 4.2 as part of the 6LoWPAN network architecture. An H node can expose a simple CoAP sensor or actuator resource (e.g.,

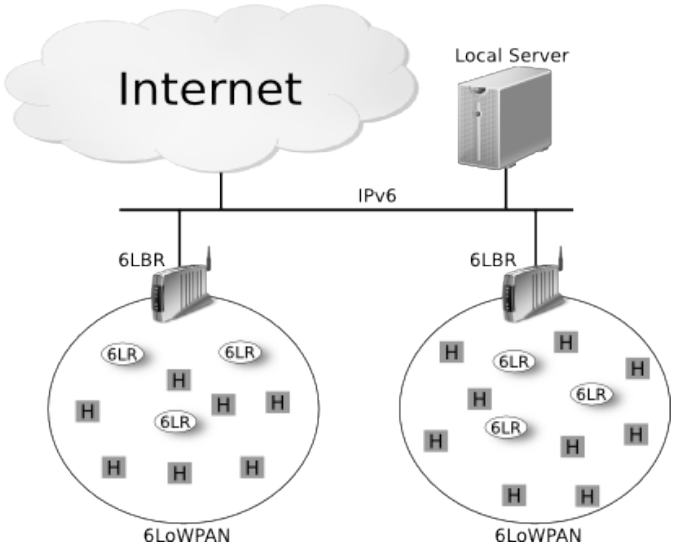


FIGURE 4.1. 6LoWPAN network.

`coap://[aaaa::1]/sensor_resource` and `coap://[aaaa::2]/actuator_resource`), while HR and HT nodes can expose RFID-related resources. More in detail, an HR node can expose RFID reader resources (e.g., `coap://[aaaa::3]/reader_resource`), while HT can expose RFID tag resources (e.g., `coap://[aaaa::4]/tag_resource`). For example, a reader-related resource can represent hardware configuration parameters or an aggregated information obtained by reading a set of tags in the communication range. Similarly, a tag-related resource can either represent the Electronic Product Code (EPC) memory content of a passive tag or a sensor value collected by a semi-passive tag.

4.1.3. Advanced safety system for industrial plants

In this section we describe a possible application of the proposed integration of RFID and WSN technologies. We consider the Smart Factory use case and we design an advanced safety system able to guarantee a safe access to dangerous areas in which safety equipments are required. Considering an industrial plant, the environment is usually divided into several restricted areas, each of them characterized by a security access level. The access control system provides safety for the workers and prevents unauthorized use of the facilities. The access to the Area Under Control (AUC) can be granted if a worker asking for the access is wearing the safety equipments required for that area. If the worker's request matches the minimum set of requirements for the AUC, then the door can be opened and the worker can enter the area.

A sketch of the AUC area is depicted in Figure 4.3. In the area, several 6LR and H nodes are deployed to collect data from the environment. Each of them is able to expose a sensing or actuating resource which can be manipulated through CoAP methods by the Local Server (LS) connected to the 6LBR. Two main actors of the system are installed next to the door: *i)* the HR node, in charge of detecting whether a worker identified by its own identity device,

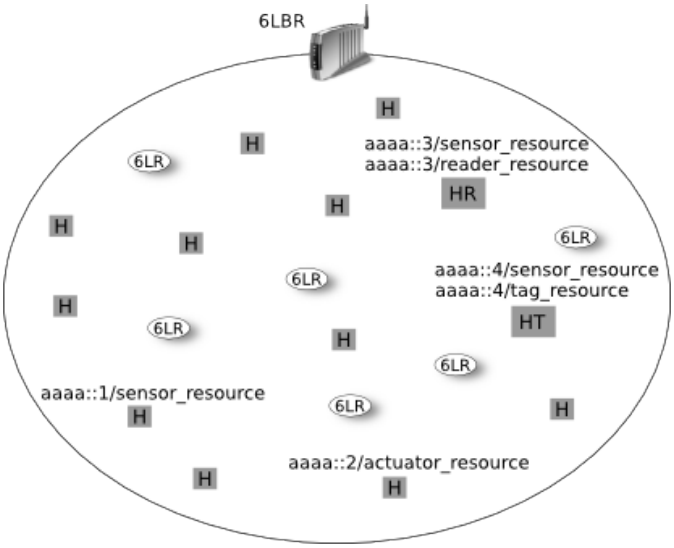


FIGURE 4.2. IoT-based WSN and RFID integrated network.

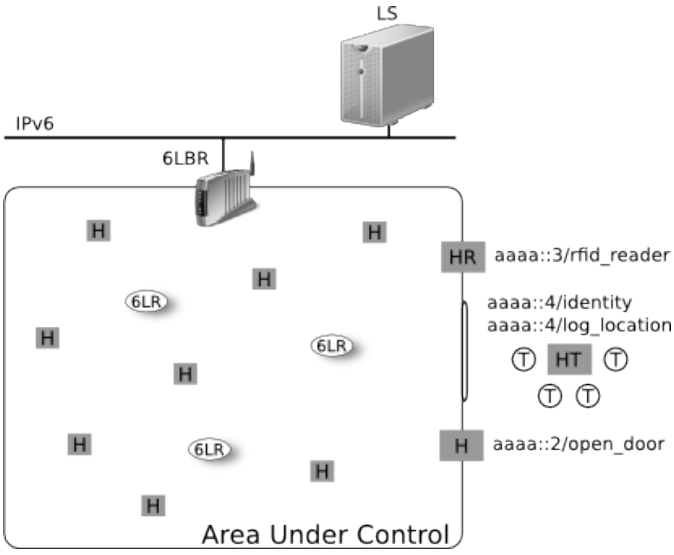


FIGURE 4.3. System components and exposed resources.

HT, is wearing all the necessary equipments, and *ii)* the H node able to open the door in case the access is authorized. The HR node is exposing a `coap://[aaaa::3]/rfid_reader` resource from which the LS can receive detection events. In the system design the HR and H nodes have been kept separate to support cases in which a remote actuation is required (e.g., two consecutive doors in a corridor must be opened). However, in the simple scenario reported in Figure 4.3 the functions of the H node could be embedded in HR, thus reducing deployment costs. To better explain the proposed safety access system design a sequence diagram has been

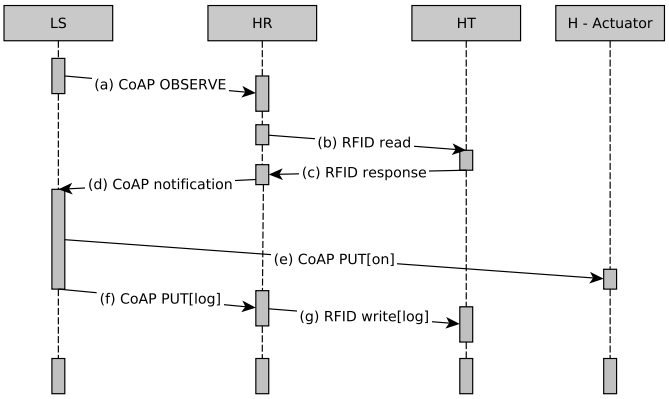
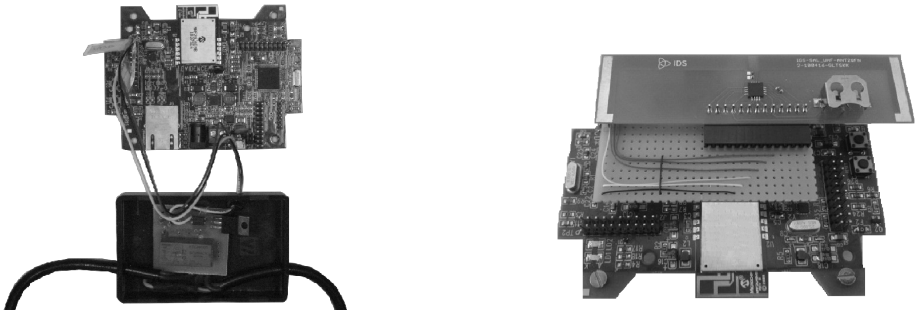


FIGURE 4.4. Messages exchanged among network devices.

reported in Figure 4.4. The diagram shows the messages exchanged in the setup and operational phases of the system. In the setup phase, the LS activates the subscriptions to the necessary resources (e.g., `coap://[aaaa::3]/rfid_reader`) by sending a CoAP message (a) to the HR node using the CoAP observe protocol. The HR node maintains a list of the active subscriptions, while in LS an event handler is installed at run-time and associated to received notification messages. During the operational phase, when the HT node is in the communication range of HR, the identity of the worker and the list of the equipments to wear are read by accessing the memory of the tag embedded on it (b and c messages). At the same time a new CoAP resource `coap://[aaaa::3]/tag_aaaa_4` is created in HR. The new resource logically represents the memory of the HT tag, and provides a virtual access to the CoAP resources exposed by HT (e.g., `coap://[aaaa::4]/identity` and `coap://[aaaa::4]/log_location`). If all the required equipments are detected by reading their own passive T tags, the HR node sends an authorized request event (d) to LS using CoAP. The event is then automatically handled by LS which sends a request (e) to the actuator node exposing the `coap://[aaaa::2]/open_door` resource. The list of all necessary equipments is securely stored in the HT node in order to keep the safety requirements close to the worker, thus enabling multiple checks in case several HR nodes are installed in the AuC area. After the authorization phase, the `coap://[aaaa::3]/tag_aaaa_4` resource is used as a proxy resource to store the identification number of the AuC the worker is accessing in the tag memory of the HT node (messages (f) and (g). This mechanism creates a per-AuC tracking application. The `coap://[aaaa::3]/tag_aaaa_4` resource becomes a virtual access point to the `coap://[aaaa::4]/log_location` resource.

If two workers are in the communication range of the HR node, it may be possible for some of the safety equipment to be associated with the wrong worker. The problem has been solved by storing a worker identification number on both HT and T tags. When the identity of a worker is read from HT, the HR node ignores all the equipment tags in the same operational range showing a different identification number. If several HT nodes have been detected, the check for authorizing the entrance in the AuC area is performed sequentially for each worker.



(a) (b)

FIGURE 4.5. H node with actuation resource (a) and HT node embedding a semi-passive tag (b).

Hardware components

The main hardware components of the system are, according to Figure 4.3, the LS, the 6LBR, the H node (embedded with an actuation resource), and the two new nodes HR and HT. We selected the Seed-eye WSN board to support the RFID integration and to implement the functions of 6LBR and H nodes. The H node, supporting actuation capabilities, has been built by embedding a relay with two exchange connectors on the board, and operating it via a serial connection. A picture of he H node is reported in Figure 4.5a.

The HT node has been implemented by connecting a semi-passive RFID tag with the Seed-eye board. A picture of the node is reported in Figure 4.5b. The selected embedded tag is the *IDS SL900A* chip, able to communicate in the UHF bands ranging from 860 MHz to 960 MHz,

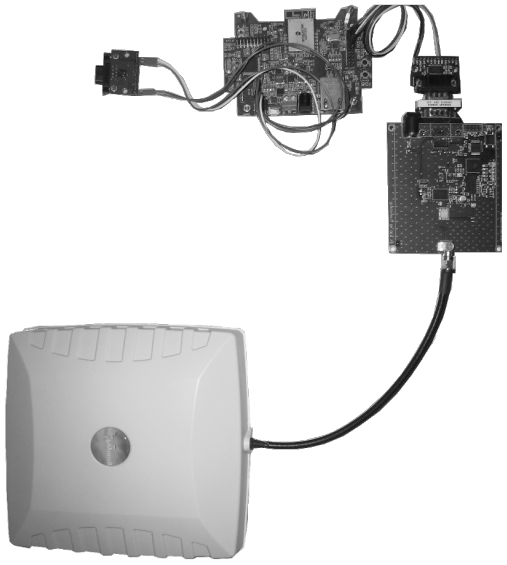


FIGURE 4.6. HR node embedding an RFID reader.

and usable as passive or semi-passive RFID tag. Indeed, thanks to an internal real-time calendar, the *SL900A* chip can be operated as data-logger by connecting external sensors to dedicated pins. The *SL900A* has been connected to the Seed-eye board trough a serial peripheral interface, from which it is possible to read the EPC memory of the device. The last key component of the system is the HR node, depicted in Figure 4.6. It has been created by interconnecting an RFID reader to the Seed-eye. The selected reader is the module *Sensor ID Discovery UHF OEM* which has been connected to the Seed-eye by means of a simple serial interface. The reader supports the EPC standard for reading data from tag memories. Thanks to the selected omni-directional antenna the reader is able to read tags at a distance of 5 meters. The choice of using RFID hardware equipments working in the frequency band ranging from 860 MHz to 960 MHz avoids any possible interference with the selected WSN devices. However, the interference between RFID and WSN is still an open issue to be solved in large integrated networks operating in the same frequency band [**MD09**]. Regarding the passive tags used on safety equipments, the *Alien ALN-9654 G* have been selected due to their low-cost and compliance with the EPC standard.

Software components

We selected Enterprise OS [**GBL+00**] to develop the firmware of all the nodes of the system. Erika OS is characterized by an extremely reduced memory footprint and provides real-time scheduling policies (e.g., fixed priority, earliest deadline first) for organizing tasks execution. We choose to use Erika OS because of its real time features that allow the integration of background monitoring and maintenance tasks on the nodes. These tasks may run in the node at a low priority level, while preserving the required QoS for high priority activities, such as those related to RFID authorization. Furthermore, Erika OS comes with a fully compliant, lightweight network stack, including IEEE 802.15.4 (μWireless), 6LoWPAN, and CoAP protocols. In the 6LoWPAN stack both flooding and geographical routing protocols have been implemented, namely 6LoWPAN Ad-Hoc On-Demand Distance Vector (AODV) and 6LoWPAN SPEED [**BPP+11**].

In the WSN and RFID integrated solution proposed in Section 4.1.2 all the RFID-based capabilities have been abstracted as network resources. In order to provide this function the RFID based software components have been implemented in Erika OS as system drivers. For the HR node several functions to set hardware parameters have been implemented, as well as functions for configuring the serial communication interface and for managing the EPC memory of read tags. The HT node driver consists of several functions able to configure the semi-passive tag parameters and to get values from possible sensors embedded into the tag.

The LS software is based on PyoT. As described in Section 3.2 PyoT allows to define events handlers to automatically react to network notifications. More specifically the LS implements a Pwn, permitting a decentralized handling of events notified by HR nodes.

4.1.4. Performance evaluation

The performance of the system have been evaluated on a real deployment reflecting the scenario depicted in Figure 4.3. Both the HR and the H actuator nodes have been installed close to the entering door of a AUC, while inside the area three 6LR nodes have been deployed, as well as a 6LBR and a laptop working as LS. We conducted two experiments with the aim of

evaluating: *i)* the system response time and its accuracy in authorizing the access in the AUC, and *ii)* the system response time in logging AUC identification data on the HT node.

System response time and accuracy in authorization control

In the first experiment we evaluated the response time of the system to an access authorization request. When the HR node detects the presence of the HT node it sends an authorization request to the LS. The request is then processed by the LS and a message is sent to the actuator node. The response time is measured as the time between the detection of the HT node by the HR node, and the reception of the actuation message by the actuator node. We evaluated the response time as a function of the total number of transmission hops necessary for sending the various requests from HT to LS and from LS to H. The number of communication hops has been modified by forcing a static routing in the 6LR nodes, i.e., a static routing table has been defined for all nodes for each experiment. In the case of two hops, no 6LR nodes are allowed to forward messages, and HR and H communicate directly with the 6LBR.

The results of the performed experiments are summarized in terms of mean (95% confidence interval) and standard deviation values in Table 4.1. The figures reported in the table have been obtained performing one thousand authorization requests for each considered network configuration.

In Figure 4.7 the response time probability density functions for the network configurations of two and five hops are shown.

TABLE 4.1.
Response time as a function of the number of hops.

Number of hops	Response time [ms]	
	μ	σ
2	147.59±1.54	24.75
3	150.79±2.02	29.72
4	156.18±1.58	32.04
5	165.32±3.76	58.40

The time values reported in Table 4.1 include the processing time spent by each node, and the network stack delays for receiving and sending CoAP messages. They also include the time required by PyoT on the LS to process the event and trigger a message transmission in response. As expected the whole response time increases, in average, as a function of the number of hops. The overall time in authorizing the entrance in the AUC is compatible with the requirements of the use case considered.

In the system design, an event is sent from HR to LS only when a worker is wearing all the necessary safety equipments, which means that all the passive tags T detected by HR are in the equipments list stored in the HT node. This approach permits to have a certain percentage of

false access rejections, while no false access acceptances are allowed. It must be stressed that, while the absence of false acceptances depends on the system design, possible false rejections strictly depends on the accuracy of the hardware equipments used for reading the RFID tags. RFID readers able to reach higher output power levels, as well as directive antennas with higher gains can greatly reduce the phenomenon, thus significantly reducing the possibility of a false rejection. However, the detailed phenomenon can be experienced in a real scenario, thus its impact has been evaluated when one or two safety equipments are necessary to enter in the AUC area. The system false rejection rate has been evaluated by creating an authorization event, e.g., all the tags are installed in the HR range at a distance of one meter from the antenna, and counting the number of reading attempts necessary to generate the event. The results of the performed analysis have been reported in Table 4.2 for the two considered cases. For each of them one thousand experiments have been performed.

As shown in the table, the false rejection rate is equal to 0 % after four attempts when one passive tags and the HT have to be read. When two passive tags and the HT node are necessary to generate the authorization event the rejection rate reaches a value of 1.5 % after six attempts.

False rejections in generating the authorization event increase the overall response time of the system. Combining the response time results in sending the authorization event with the additional delay for generating the event itself, a figure of merit about the overall system response time can be obtained. Considering the maximum delay of a five hop communication, and the highest event generation time experienced in the case of one HT node and one T tag the overall response time is less than 400 ms, which slightly increases in case of two T tags are used. In both cases the value is acceptable for the considered use case.

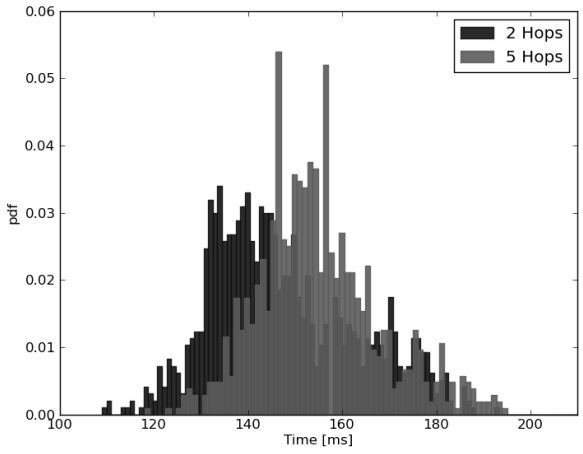


FIGURE 4.7. Response time pdfs for the network configurations of two and five hops.

TABLE 4.2.
False rejection rate vs. number of reading attempts.

Number of reading attempts	False rejection rate [%]	
	HT=1, T=1	HT=1, T=2
1	10.8	49.3
2	2.6	24.9
3	0.2	13.6
4	0.0	6.7
5	0.0	2.8
6	0.0	1.5

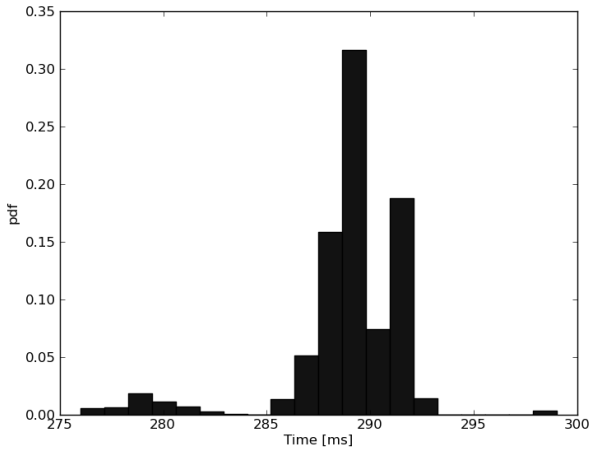


FIGURE 4.8. Response time pdf for logging location data.

System response time in logging location data

As previously described, by accessing the *log_location* resource of the HT node, a per-AUC tracking application can be implemented. According to the system design, the AUC identification data are stored in the EPC memory of the HT node. The *log_location* request is sent immediately after the *open_door* request, as shown in Figure 4.4. The logging time is defined as the time between the reception of the *log_location* request by the HR node and the time when the HR node receives an acknowledgment indicating that the write operation on the HT node is successful.

The results, shown in Figure 4.8, represent the time required for sending an EPC write memory request by the RFID reader, and for reading an acknowledgment in the memory of the semi-passive tag embedded in the HT node. With the adopted hardware solutions, by

installing HT at distance of one meter from the antenna, the average experienced logging time is 289.41±1.73 ms (95% confidence interval). Again, the results have been obtained by performing one thousand experiments. Similarly to the previous experiment, the HR may need to perform some attempts in order to complete the writing operation. This behavior can be observed in Figure 4.8, where in the pdf distribution two main Gaussians can be noticed, the first one with a peak around 280 ms and the second one with a peak around 290 ms. Better performance can be obtained by using improved RFID hardware equipments.

The rest of the chapter presents a different IoT application targeted to the Intelligent Transportation use case. Although applied to a different scenario, the programming abstractions described in the previous chapter confirm to be a valid tool supporting the development of IoT applications.

4.2. Intelligent Transportation Systems

The development of smart cities relies on the success of transportation systems providing efficient control and management using advanced sensing infrastructure and networking. In the context of Intelligent Transportation Systems the Internet of Things enables the creation of novel processes or to increase the efficiency of the existing ones. Another key element for the success of large-scale and interoperable smart infrastructure is the adoption of standard network architectures and common data formats, allowing the Machine-to-Machine (M2M) [**KLKY14**] paradigm to emerge. Thanks to M2M, devices communicate and cooperate with each other without human intervention. In order to fully enable the M2M paradigm, a complete network architecture with well-defined interfaces is required, providing full interoperability among heterogeneous systems. A strong standardization effort in this direction is provided by the European Telecommunications Standards Institute (ETSI) within the Smart Machine-to-Machine communications (SmartM2M) Technical Committee[1], working on requirements, networking, protocols and security aspects related to M2M communications. Despite the standardization efforts, effectively building M2M applications in the IoT domain remains a challenging task, partly due to the lack of common tools and because of the limited resources available on typical sensor devices [**JHVdV+09**]. Most importantly, state-of-the-art solutions are only designed to support static systems, in which the M2M communication endpoints are fixed and associated to each node during the application's design phase. However, IoT systems exhibit a higher level of unreliability, for example due to nodes often connecting and disconnecting at run-time. Moreover, since nodes are usually battery-powered, it is important to consider mechanisms to reconfigure the system and replace the function of nodes failing due to battery depletion.

Efficient solutions for moving an M2M endpoint from one node to another are missing. Moreover, the current M2M architecture defines a strict separation between data producers and consumers: the former typically reside on resource-constrained nodes, the latter on resource-rich gateways and full-fledged network applications. As a consequence it is difficult to exploit the

[1]Technical Committee Smart Machine-to-Machine communications. `https://portal.etsi.org/TBSiteMap/SmartM2M/SmartM2MToR.aspx`

benefits of in-network data processing in terms of energy and network traffic reduction. Also, network applications become more complex, having to deal with raw, non-aggregated data.

In order to tackle with these issues we present the ICSI M2M Middleware, developed within the Intelligent Cooperative Sensing for Improved traffic efficiency (ICSI) project[2] co-funded by the European Commission within the FP7 program. The main objective of ICSI is to develop a new Intelligent Transportation System (ITS), within a Smart City scenario, in which distributed and pervasive vehicular and sensor network technologies coexist. The presented ICSI M2M Middleware on the one hand is able to provide in-network event composition by leveraging on a RESTful publish/subscribe model. On the other hand it enables the dynamic reconfiguration of ETSI-M2M-compliant nodes by using standard IoT protocols.

Several works on data aggregation techniques in Wireless Sensor Networks (WSN) exist in the literature [**RV06**, **WJTL12**]. However, we focused our effort on the architectural solution and on the interfaces supporting generic "in-network" data processing in M2M networks. Existing data aggregation schemes may be easily applied to the ICSI M2M Middleware architecture.

4.2.1. ETSI M2M communication paradigm

In this section the ETSI M2M communication paradigm is discussed by underlining the role of constrained sensor devices in the system architecture. Within the IoT vision and thanks to the reference protocol stack described in Section 1.1, network applications can exploit the RESTful architecture to communicate with resource-constrained devices. In order to increase the level of interoperability of IoT systems, RESTful resources and access methods can be defined in a cross-service manner. A framework supporting M2M communication on the IoT should consider that sensor devices may be powered off for long periods of time due to energy saving policies. Also, control procedures such as access methods, and remote management procedures should be abstracted according to the RESTful paradigm.

A number of solutions have been proposed in order to address the requirements above. Among them a strong candidate is the ETSI M2M, since it constitutes the output of a working group formed by experts from a number of diverse ICT areas. The ETSI M2M[3] addresses the requirements presented above and consists in a horizontal framework enabling applications to use data independently of their location and of the protocols needed to access them. It consists of a distributed system composed of M2M Applications and a M2M framework called Service Capability Layer (SCL). M2M Applications can reside in a Device Application (DA), a Gateway Application (GA) or in a Network Application (NA). The high-level M2M architecture distinguishes within Device and Gateway Domain and Network Domain, as reported in Figure 4.9. According to the ETSI definition, a Device (D or D') is an equipment that may collect a set of actuators and sensors with embedded computing and communication capability. A Gateway instead aims at translating and transferring information between two or more communicating

[2]Intelligent Cooperative Sensing for Improved traffic efficiency (ICSI) `http://www.ict-icsi.eu/` November 2012.

[3]ETSI TS 102 690. Machine-to-Machine communications (M2M); Functional architecture, October 2013.

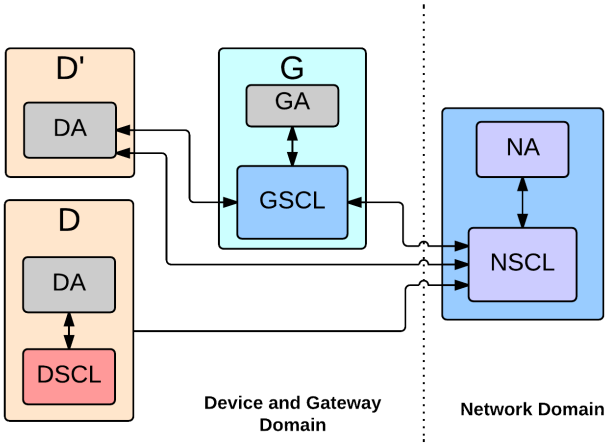


FIGURE 4.9. M2M architecture for different deployment scenarios.

entities, or at performing some routing and multiplexing function between the communicating entities. A Network Node is instead a generic communication entity which does not fall into the two aforementioned categories.

The M2M Service Capabilities Layer enables communication between M2M Applications, abstracting the distributed system as a set of resources and operations to be executed on them. The Service Capabilities Layer is present in the Network Domain as Network SCL (NSCL), in M2M devices as Device SCL (DSCL) and in M2M gateways such as Gateway SCL (GSCL). While a Gateway and a Network Node always include a Service Capability Layer, a Device may (D) or may not (D') host a local SCL. In the latter case it accesses the M2M framework through a GSCL. While full-fledged sensors are capable of implementing the DSCL feature (D devices), the D' option provides the best solution in terms of complexity for resource-constrained devices.

In order to support a common data representation, ETSI recommends three data serialization formats: JavaScript Object Notation (JSON), Extensible Markup Language (XML), and Efficient XML Interchange (EXI) [**SKPK11**]. While JSON is a data format widely used for the interactions between servers and web applications, XML has been recognized as the de-facto standard for data representation and exchange. If on the one hand it guarantees a great flexibility, on the other hand it exhibits a large intrinsic redundancy. The Efficient XML Interchange format is a compact XML representation, currently being standardized by the World Wide Web Consortium (W3C). It is designed to support high-performance XML applications for resource constrained environments, significantly reducing bandwidth requirements and improving encoding/decoding performance [**CGB+11**].

4.2.2. The ICSI use-case

The main goal of ICSI is to provide a platform for the deployment of cooperative systems, based on vehicular network and wireless sensor network communication technologies, with the aim of enabling a safer and more efficient mobility in both urban and highway scenario. In this

section we consider the WSN use-case and we introduce the main features and requirements of the ICSI M2M Middleware.

As part of the roadside network, the ICSI WSN is responsible for collecting and aggregating ITS-related events. The network is composed by low-cost visual sensors nodes collecting information on parking slots availability and traffic flows (e.g., as proposed in [**SPB+14**, **SPGP12**]). The collected data are then forwarded towards the ICSI Gateway and shared by a data distribution platform with the rest of the system. In order to cope with the problem of temporary occlusion of the field of view, it is possible to deploy different visual sensors monitoring the same scene, as depicted in Figure 4.10. When an event, e.g., a parking slot becoming free or busy, is detected by more than a single sensor it is necessary to aggregate the information to provide a final decision on the status of the slot. A middleware aggregating the information inside the

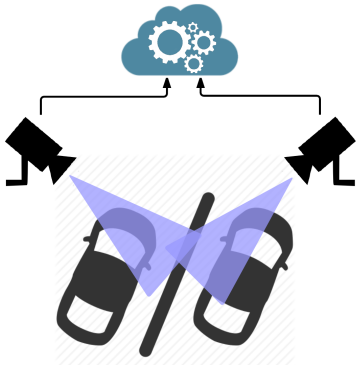


FIGURE 4.10. Aggregation of data from multiple visual sensors monitoring the same scene.

WSN brings a twofold advantage: first, it increases the robustness of the information, filtering out incorrect notifications. Second, it reduces the number of messages addressed to the upper layers. Existing solutions for in-network aggregation in the WSN domain typically select one node in a static fashion to perform the aggregation. However, if the selected node fails, e.g., due to battery depletion, the system becomes unavailable. We tackle with this problem by proposing a dynamically reconfigurable system, based on standard interfaces, capable of moving the data aggregation task from node to node at runtime.

ICSI WSN Architecture

Figure 4.11 shows the architecture of the ICSI WSN segment. All WSN nodes adopt the IoT-ready stack presented in Section 1.1, including IEEE 802.15.4, 6LoWPAN, RPL, and CoAP. A 6LoWPAN Border Router connects the sensor devices to the Internet and it is responsible for handling network traffic to and from the IPv6 and IEEE 802.15.4 interfaces. The architecture also includes 6LoWPAN wireless routers, i.e., nodes that can be added to the deployment in order to extend the communication range and enhance the reliability of the system. The ICSI Gateway implements the ETSI M2M GSCL, allowing M2M devices to create resources when new traffic-related events are detected.

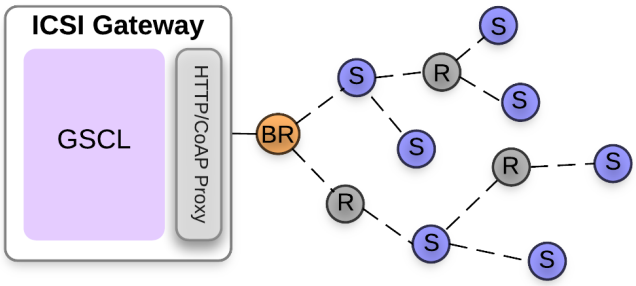


FIGURE 4.11. The ICSI WSN architecture, including sensors (S), wireless routers (R), a 6LoWPAN boder router (BR) and the ICSI Gateway.

CoAP has been designed with the goal of being easily used in REST architectures. Indeed, CoAP can easily inter-operate with HTTP through an intermediary proxy which performs cross-protocol conversion. Also, the HTTP/CoAP proxy can be configured to perform a conversion of the payload representation in a stateless fashion, e.g., EXI to XML for messages addressed to the Gateway, and from XML into EXI for messages addressed to resource-constrained nodes. We propose to extend the role of the proxy by enabling additional data compression mechanisms. Sensor nodes can transmit some invariant information in a compressed form and the Proxy can decompress them before sending it to the GSCL. The details of this technique will be presented in Section 4.2.4.

4.2.3. The ICSI M2M Middleware

In order to respond to the communication requirements of the ICSI project we developed the ICSI M2M Middleware. The system has been designed following an hybrid approach, making use of a RESTFUL design in order to provide standard communication interfaces and of software virtualization concepts to enable dynamic reconfiguration and in-network event composition.

The ICSI M2M Middleware supports three operational phases: *i)* Configuration; *ii)* M2M setup; *iii)* Monitoring. The sensor nodes act both as CoAP clients and servers depending on the specific operational phases. In the monitoring phase the sensors collect mobility information related to parking slots and traffic flows and send event notifications to the ICSI Gateway. During the M2M setup and the monitoring phases the sensors act as CoAP clients, encoding the application setup messages and the notifications into ETSI M2M packets addressed to the GSCL component of the Gateway. With reference to Figure 4.9, sensor nodes act as a D' device and implement the interface to communicate with the GSCL, as described in Section 4.2.1. During the configuration phase the sensors act as CoAP servers, exposing a set of resources representing the configuration parameters related to the vision algorithm and the in-network processing features of the nodes.

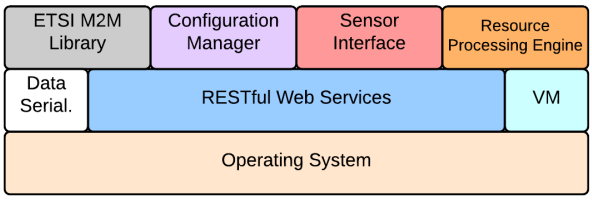


FIGURE 4.12. Middleware software archtecture.

The ICSI M2M Middleware is instantiated on each node of the WSN segment, which is in charge of gathering ITS-related parameters (e.g., number of passing cars, status of parking spaces) from the field by leveraging on vision sensors. The high level software architecture of the ICSI M2M Middleware is shown in Figure 4.12. The core modules of the middleware are placed on-top of the Operating System (OS) and make use of basic networking, virtualization and data serialization services provided by embedded software libraries.

The main components of the middleware are: *i)* the Resource Processing Engine (RPE), taking care of in-network event composition; *ii)* the ETSI M2M library, supporting standard communication with the Gateway; *iii)* the Sensor Interface, a software library abstracting the functionality of on-board sensors; *iv)* the Configuration Manager, enabling the configuration of system's functions. The basic services used by the middleware, the selecetd OS, and the middleware components are presented in the following.

Operating System and basic services

As a basic support for the ICSI M2M Middleware we selected Contiki, an open-source operating system targeted at IoT systems, while the RESTful Web Service component is provided by Erbium [**KDD12**]. The virtualization capability of the middleware in supported by PyMite [**Dea03**], a lightweight Python interpreter on which tasks and applications can be instantiated at run-time by uploading Python bytecode to the nodes. PyMite supports a subset of the Python 2.5 syntax and is designed to execute on 8-bit microcontrollers with resources as limited as 64 KiB of program memory and 4 KiB of RAM. The Data Serialization Library component provides data serialization services in two formats: JavaScript Object Notation and Efficient XML Interchange. Contiki natively provides a JSON library, while an EXI serialization library [**KED11**] has been ported to the system.

ETSI M2M library

In order to facilitate the communication with the ICSI gateway, we designed an M2M communication library for Contiki. The library uses the services provided by the data serialization libraries and exposes a set of APIs to encode/decode M2M messages. The M2M library is used both in the M2M startup phase and in the monitoring phase. During the startup phase the *Application* and *Container* M2M resources can be created on the GSCL. In the monitoring phase the *content Instances* resources (containing the traffic-related information) are created on the right *Container* resource, depending on the application type.

Resource Processing Engine

The Resource Processing Engine is the module of the middleware responsible for collecting events from various sources and performing in-network processing of events. The RPE can subscribe to local and remote events. Local events are generated directly by the Sensor Interface. Remote events can be generated by any of the nodes running the ICSI M2M Middleware. In both cases the subscription to the events is performed using the CoAP Observe mechanism. The basic events are then aggregated according to a user-defined function, expressed as a Python script, and the result of the computation is forwarded to the GSCL on the ICSI Gateway using the ETSI M2M library. The output of the aggregation is also published through the configuration manager. The RPE is based on T-Res [**APP13**]. We extended T-Res by allowing a T-Res node to send the output of the processing to M2M resources, encoding messages using the M2M library and the EXI data format. The RPE provides several benefits to the ITS use-case. First it offers a way to dynamically update the data processing function: a new aggregation algorithm having better accuracy or extracting new features from the raw sensor data can be upgraded after the deployment. Second, it allows the dynamic reconfiguration of the input sources and of the output destination of the data. A faulty sensors can be removed from the list of input sources, and if a new Gateway becomes available it can be added to the output list. Third, it allows the complete virtualization of the M2M endpoint. The data-processing function can be moved at run-time from one node to another, e.g., to a node having a better connectivity or a larger residual energy, completely decoupling the source of the M2M information from the single physical sensor.

Configuration Manager

The Configuration Manager is the component responsible for the global configuration of the ICSI M2M Middleware. Each configurable component is registered through the RESTful Web Service and provides the APIs to allow remote configuration. This component provides access to a wide range of functions, including: *i)* RPE configuration, i.e., input/output resources and processing; *ii)* Over-the-air software update; *iii)* Energy Policy Reconfiguration (duty cycle configuration); *iv)* Networking and general purpose maintenance features.

Figure 4.13 summarizes the APIs exposed by the Configuration Manager module for the parking slot use-case. Since the module relies on the RESTful interface provided by the lower layers, the configuration phase can be performed using a simple, standard CoAP client (HTTP client if the proxy is used). In order to facilitate the configuration tasks, PyoT has been installed on the Gateway. As described in the previous chapter PyoT allows to interact with a set of CoAP resources using macroprogramming scripts, abstracting the resources as Python objects. Moreover, it supports the creation and instantiation of the T-Res tasks providing in-network aggregation of visual sensor data.

4.2.4. Performance evaluation

In this section we evaluate the performance of the ICSI M2M Middleware. The first set of experiments focuses on the performance of the data serialization and ETSI M2M libraries.

URI	Methods
/cfg/	GET
/status	GET
/battery	GET
/period	PUT | GET
/firmware	PUT | GET
/reset	PUT
/tasks	GET | POST
/park	POST
/pf	GET | PUT
/is	GET | PUT
/od	GET | PUT
/sensor	GET
/park_output	GET

FIGURE 4.13. The APIs exposed by the Configuration Manager in the parking slot monitoring use-case.

Then we report on the preliminary performance evaluation of the ICSI M2M Middleware in a laboratory testbed.

Data Serialization

In order to assess the performance of the ICSI M2M Middleware we first conducted a comparative study of the data serialization formats recommended by the ESTI M2M working group. In particular, we focused on the comparison of EXI and JSON[4]. In order to align our work with the ETSI standard, we conducted performance measurements for an explicitly declared set of messages deduced from the ETSI CoAP M2M Interoperability Test Suite [**ETS**]. The comparison served as a base to select the best serialization method for the ICSI use case.

The experimental setup consists of a wireless sensor network composed of Wismote sensor nodes, emulated in Cooja. The sensor nodes represent the communication endpoints of a generic M2M scenario. Figure 4.14 shows the simplest configuration, comprising three sensor nodes, acting as client, server, and 6LoWPAN border router. The client and the server are programmed to execute an M2M interaction using the test library we developed implementing the ETSI M2M test suite. CoAP endpoints exchange messages using our serialization library.

Performance Metric

The performance of the two ETSI M2M data serialization formats viz., JSON and EXI are compared according to the following criteria:

1. *Performance over ETSI M2M CoAP Interoperability Tests*: This measurement focuses on the execution time, channel usage and energy consumption of several M2M operations.
2. *Compression gain*: This measures the percentage of compression achieved.

[4]EXI is a light-weight realization of the XML, thus we omit the comparison of XML

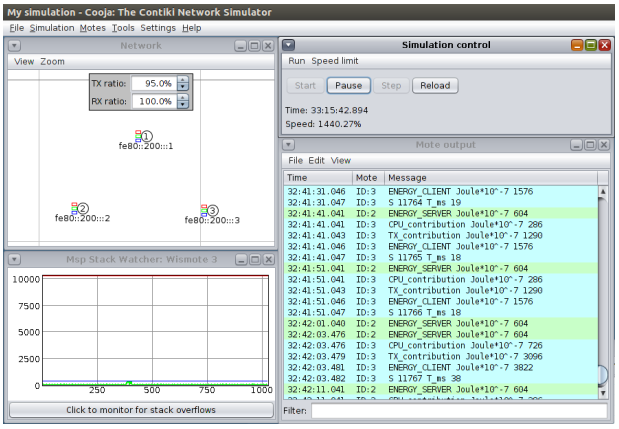


FIGURE 4.14. The data serialization experimental setup

3. *Serialization complexity*: This measures the complexity with respect to the memory usage, serialization and deserialization times.

For evaluating the EXI format two configurations are considered, corresponding to Schema-Enabled (SE) and Schema-Less (SL) serialization. The "Schema-Enabled" compression mode (also known as "schema-informed") can achieve better compression performance compared to the Schema-Less mode, but requires both the encoder and the decoder to share an XML schema. In addition, the EXI SE uses schema information to improve compactness and efficiency.

The ETSI M2M CoAP Interoperability Tests consist of the execution of a set of operations (create, retrieve, update) on ETSI M2M resources (*application*, *subscription*, *contentInstance*). For instance in our tests we performed an update operation on sensor nodes to manipulate the sensor resources (e.g., temperature and humidity) mapped over ETSI M2M resources. In our setup, we configured one CoAP endpoint to simulate a M2M Device Application and the other endpoint as the M2M Gateway. The ETSI M2M standard defines different possible profiles for message instances. We tested different message instances sent between the CoAP endpoints, which differ in terms of length and complexity. We categorized the message instances according to Profiles 1 to 4, where Profile 1 is shorter with less complexity and Profile 4 is longer with more complexity.

We show an example result from the *applicationCreate*. The *applicationCreate* is mapped on a CoAP POST message, and the measured execution time interval includes the serialization time, the delivery time of the request, and the delivery time of the response on the network.

Figure 4.15 shows *applicationCreate* execution times, using the different Profiles previously defined. When Profile 1 is adopted, we use just one CoAP block of 64 bytes with all data formats considered. The EXI SE has a lower execution time compared with JSON and EXI SL, resulting in an overall faster *applicationCreate*. The same observation is made with Profiles 2 to 4, where EXI SE outperforms the other data format configurations. With respect to the other formats, Schema-Enabled EXI has a clear advantage which increases with payload complexity: the gain

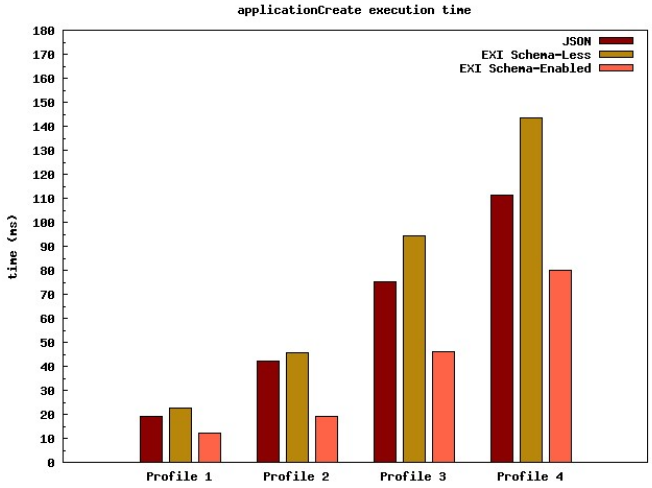


FIGURE 4.15. *applicationCreate*: Execution time.

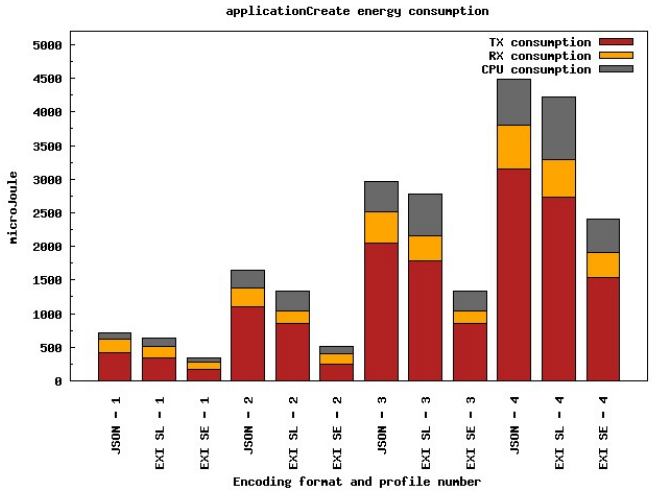


FIGURE 4.16. *ApplicationCreate*: Energy consumption.

in efficiency in terms of transmission time of a smaller payload largely compensates its encoding time.

In Figure 4.16 we show the energy consumption of an *applicationCreate* operation with respect to transmission, reception and CPU energy consumptions. From this figure, we can observe that EXI SE performs better compared with EXI SL and JSON. Even if EXI Schema-Less performs worse in time, it brings a benefit in comparison to JSON from an energy standpoint. This is due to the fact that a smaller payload implies lower radio usage, which has an higher impact on power consumption than the one deriving from the CPU usage. Table 4.3 shows that

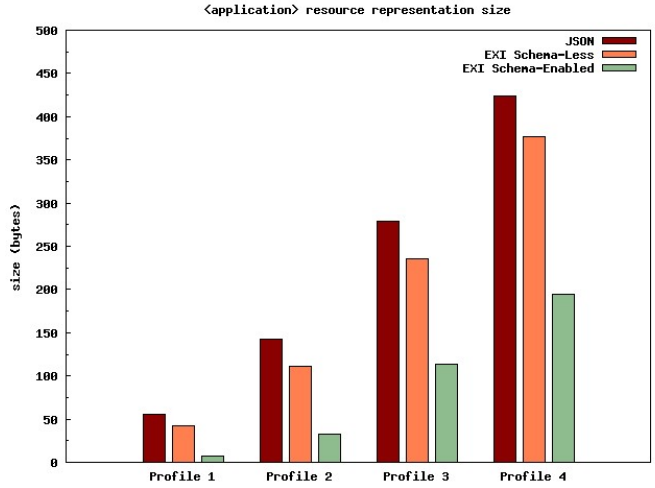


FIGURE 4.17. Compression gain on size of serialized data: *application* resource serialization.

EXI SL reduces channel usage on average by 15% with respect to JSON, while EXI SE reaches an average of 50%.

TABLE 4.3. *applicationCreate* channel usage

Profile	JSON	EXI	
		Schema-Less	Schema-Enabled
	bytes	bytes	bytes
1.	245	232 (-5%)	155 (-36%)
2.	640	469 (-27%)	181 (-72%)
3.	1135	953 (-16%)	471 (-58%)
4.	1679	1448 (-14%)	870 (-48%)

Compression Gain:

To compare the compression gains, we show example results for the serialization of two ETSI M2M resources i.e., *application* and *subscription*. In Figure 4.17 and 4.18, we can see that the EXI SE achieves more compression gain compared with both EXI SL and JSON. Whenever there are large data items with highly repetitive structures we would expect EXI SL and EXI SE to become closer in performance. This is due to EXI SL dynamic learning mechanisms: the compressor builds up grammar structures when a given structure is processed the first time in such a way that they can then be referred to indirectly when such structure is subsequently encountered, without re-encoding the structure item.

We report on the serialization complexity measurements for an *applicationCreate* operation in terms of time required to perform serialization and deserialization. Tables 4.4 and 4.5 reveal that JSON performs better compared with EXI SE and SL. Similarly, Table 4.6 shows the memory usage. The estimate of the stack size at run time include EXIp, Contiki OS and its

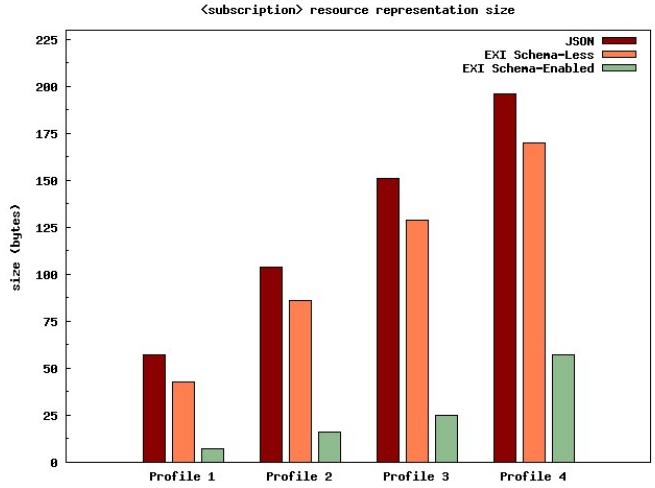


FIGURE 4.18. Compression gain on size of serialized data: *subscription* resource serialization.

TABLE 4.4. *application* resource. Serialization time complexity

application resource serialization time			
	JSON	Schema-Less	Schema-Enabled
Profile	(ms)	(ms)	(ms)
1.	0.17 ± 0.02	5.64 ± 0.03	2.50 ± 0.01
2.	0.42 ± 0.01	14.80 ± 0.07	6.07 ± 0.03
3.	0.62 ± 0.02	31.5 ± 0.2	15.01 ± 0.07
4.	1.10 ± 0.01	48.0 ± 0.2	23.1 ± 0.1

TABLE 4.5. *application* resource. Deserialization time complexity

application resource deserialization time			
	JSON	Schema-Less	Schema-Enabled
Profile	(ms)	(ms)	(ms)
1.	0.27 ± 0.01	3.60 ± 0.01	2.05 ± 0.01
2.	0.73 ± 0.01	9.70 ± 0.04	4.45 ± 0.02
3.	1.09 ± 0.01	19.17 ± 0.09	8.46 ± 0.04
4.	1.86 ± 0.01	27.5 ± 0.1	12.05 ± 0.06

communication stack, namely: IEEE 802.15.4, 6LoWPAN, and Erbium (CoAP). On average EXI SE and SL use up to 40% and 30% of the stack respectively. Out of the initial *Wismote* RAM size of 16 KB, 8 KB is used by EXIp and 5.7 KB is used by Contiki OS and the communication stack. Although it has a lower serialization complexity, JSON generates larger packets, leading to longer execution time for data transfer on the network channel and possibly more retransmission on a lossy network.

TABLE 4.6. Memory usage - *applicationCreate*

Memory usage - *applicationCreate*			
	JSON	Schema-Less	Schema-Enabled
Text + Data (ROM)	5 KB	43 KB	50 KB
Data + BSS (RAM)	210 B	8 KB	9 KB
Stack at run-time (RAM)	(500 ± 70) B	(680 ± 40) B	(650 ± 40) B

TABLE 4.7. Summary of Results

Criteria	JSON	EXI Schema Less	EXI Schema Enabled
memory occupation			
ROM	1	2	3
RAM	1	2	3
applicationCreate			
Execution Time	2	3	1
Energy Consumption	3	2	1
Channel Usage	3	2	1
applicationRetrieve			
Execution Time Sc. 1	3	2	1
Execution Time Sc. 2	2	3	1
Energy Consumption	3	2	1
Channel Usage	3	2	1
applicationUpdate			
Execution Time	-	-	1
Energy Consumption	3	2	1
Channel Usage	3	2	1
subscriptionCreate			
Execution Time	2	3	1
Energy Consumption	-	-	1
Channel Usage	3	2	1
contentInstanceRetrieve			
Execution Time Sc. 1	3	2	1
Execution Time Sc. 2	-	3	-
Energy Consumption	-	-	1
Channel Usage	-	-	1

To sum up, EXI SE outperforms both JSON and EXI SL configurations in terms of energy consumption, channel usage and the execution times. However EXI has significantly higher memory requirements. For completeness, we summarize in Table 4.7 the results obtained from our experiments for all the ETSI M2M resources i.e., *application*, *subscription* and *contentInstance*. We rank the performance from 1 to 3, where 1 ranks as the best performance and 3 ranks as the worst. The data serialization methods considered are placed in relative order whenever possible. The dash ("-") signifies situations where, according to the chosen benchmark, evaluation of the results is not needed. From our results, we made the following observations:

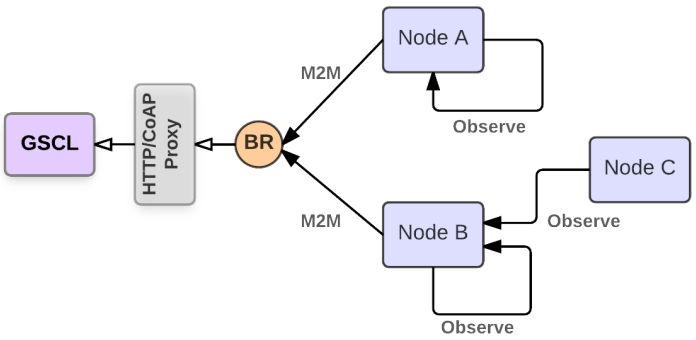


FIGURE 4.19. Experimental setup in the laboratory testbed, including a border router (BR), three sensor nodes (A, B, C), and the notebook hosting the emulated Gateway.

1. EXI SE performs better in terms of energy consumption, compared with JSON and EXI SL. Similarly, with respect to the execution time, EXI SE performs better in the majority of cases. This is however achieved at the cost of a higher memory occupancy, which can still be accommodated on a sensor node. Moreover endpoints need to share grammar structures.
2. The average performance of EXI SL is comparable with JSON; in some cases it performs better than JSON with respect to energy consumption. In our implementation, EXI SL execution time is better to an equal extent compared with JSON especially when the serialization time is negligible.
3. With respect to channel usage, in all scenarios considered EXI performs better compared with JSON, especially when EXI SE is used.

Following the observation on the performance of the different serialization methods supported by the serialization library, we decided to adopt the EXI SL mechanism in the ICSI project. EXI SL provides better performance with respect to JSON. Also, it does not require the communication endpoints to share any static information, such as the XLM schema, greatly simplifying the deployment and maintenance phases.

Testbed

In order to assess the performance of the ICSI M2M Middleware in a realistic scenario we implemented a simple testbed, emulating the parking slot use case described in Section 4.2.2. The testbed includes three sensor nodes, one 6LoWPAN border router, the HTTP/CoAP proxy, and a HTTP server emulating the GSCL.

The sensor nodes have been programmed in order to emulate the generation of events related to parking slots availability. The experimental setup, depicted in Figure 4.19, shows the relationship between the sensor nodes. While *Node A* is configured to monitor only local events,

Node B subscribes both to local events and to events generated by *Node C*. The basic events are then aggregated by the RPE using a simple averaging function and eventually sent to the GSCL through the proxy as new *ContentInstance* resources. The proxy and the GSCL are hosted on the same laptop. We measured the event transmission latency, i.e., the time between the event generation on the RPE and the creation of the M2M resource on the GSCL. The measured latency includes the time required to:

1. encode the events in a M2M message using the EXI compression;
2. send the M2M CoAP message to the Proxy, using the *Blockwise* mechanism when necessary;
3. convert the CoAP/EXI message to HTTP/XML on the proxy;
4. transmit the HTTP packet to the HTTP server emulating the GSCL.

Table 4.8 presents the results of the experiment: *Sensor to Proxy* is the time required to complete steps 1 and 2, while *Sensor to GCSL* is the overall event notification latency. The start time is taken when *Node B* receives the notification from the local RPE. The second timestamp is taken when the proxy receives the complete representation of the resource sent by *Node B*. Finally we take the last timestamp when the HTTP message is received and parsed by the HTTP server. We measured the latency using three different M2M message types, varying the size of the *ContentInstance* representation, that is the XML document containing the M2M notification. The table includes the size of the original XML message, prior the EXI compression. The results show that latency grows with the number of CoAP blocks. Also, since every message transmission requires an acknowledgement, the greatest contribution to the overall latency is due to the CoAP *Blockwise* transfer to the Proxy. However, the resulting latency is fully compatible with the dynamics of the considered ITS applications.

TABLE 4.8. ICSI middleware notification latency.

CoAP Size [bytes]	CoAP Blocks [#]	XML size [bytes]	Sensor to Proxy [ms]	Sensor to GSCL [ms]
104	2	262	168.07 ± 0.24	176.01 ± 0.27
155	3	313	219.16 ± 0.29	227.79 ± 0.42
228	5	386	279.27 ± 0.35	287.06 ± 0.38

In order to reduce the number of CoAP blocks required for each notification and improve the efficiency of the communication, we extended the role of the proxy. A proxy is a CoAP endpoint that can be tasked by CoAP clients to perform requests on their behalf. In our architecture the Proxy is a "forward-proxy" and is explicitly selected by clients. CoAP requests to a forward-proxy are made as normal requests to the forward-proxy endpoint, but they specify the request URI in a different way: the request URI in a proxy request is specified as a string in the *Proxy-Uri* Option, while the request URI in a request to an origin server is split into the *Uri-Host*, *Uri-Port*,

Uri-Path, and *Uri-Query* Options. Hence, the URIs of the GSCL resources need to be encoded as plain text in the CoAP message, thus occupying a significant amount of space in the packet. This can lead to excessive fragmentation in case of large resource representations (for example a M2M *Content Instance*). Our Proxy is based on the Californium [**KLS14**] implementation. We extended the proxy by adding two features: *i)* Content type translation: the Proxy translates the payload from EXI to XML and vice versa. For the EXI/XML conversion we used the Java OpenExi library in conjunction with Californium. *ii)* M2M URI decompression. The M2M URI decompression permits a significant gain in efficiency by allowing the CoAP clients to remove the static portion of the request URI from the Proxy-Uri Option string. The full URI is then reconstructed by the Proxy during the message translation phase using a conversion table. Considering the GSCL configuration used in the ICSI project, the URI compression mechanism makes available up to 40 additional bytes for every CoAP packet. In Table 4.9 we report on the memory occupation of the ICSI middleware on a Seed-Eye board, the WSN device selected for the ICSI project. The system requires 44.80% of the ROM memory and only 26.50% of the RAM, leaving a significant space available for the user-defined data aggregation functions.

TABLE 4.9. ICSI M2M Middleware memory requirements.

ICSI Middleware Memory Occupation		
RAM	34672 Bytes	26.50%
ROM	238076 Bytes	44.80%

4.3. Conclusions

In this chapter we presented two IoT-based applications targeted to different scenarios. The diversity of the application domains demonstrates the flexibility of the IoT technologies to different contexts. However, in order to further simplify the development and the deployment of IoT applications in the real world, it is useful to adopt abstractions and tools. Both the applications take advantage of the abstractions concepts described in Chapter 3. In the first part of the chapter an integration between WSN and RFID technologies in the IoT scenario is presented. By representing sensors, actuators and RFID related data as network resources, addressable through CoAP protocol methods, a seamless integration between the two technologies is achieved. The resulting system provides a technological solution in which the outstanding pervasiveness of RFIDs and the advanced sensing and communication features of WSNs are effectively merged together. The proposed solution is applied to the use case of a restricted access zone control system in a Smart Factory environment. The system is able to guarantee a safe access to workers based on the presence of safety equipments.

In the second part of the chapter we presented the ICSI M2M Middleware. The middleware is capable of supporting M2M communication on resource-constrained IoT systems, while offering a wide range of reconfiguration capabilities through a RESTful interface. The in-network

processing feature permits to reduce the number of messages exchanged in the low power network, with benefits on the power consumption and on the complexity of network applications. The ICSI M2M Middleware is targeted on the ICSI use-case of a visual sensor network in an ITS scenario. However, thanks to the RESTful design, it is general enough to support different application scenarios, including systems made of heterogeneous sensors and actuators. An extensive performance evaluation is planned for the field trial phase of the ICSI project. However, the preliminary results gathered in a laboratory testbed show the feasibility of the proposed solution on resource-constrained devices. Moreover, the measured processing and networking latencies are fully compatible with the requirements posed by the ICSI use-case.

As a further step towards the development of abstractions for the IoT, the next chapter presents Virtual Resources, a set of architectural concepts to resolve the tension between effective development and efficient operation of IoT applications. The Virtual Resources architecture contrasts the traditional Cloud-centric designs by giving developers the ability to push slices of the application logic down to the IoT network.

CHAPTER 5

The Virtual Resource abstraction for the IoT

EMERGING IoT architectures exhibit recurring traits: resource-limited sensors and actuators with RESTful interfaces at *one* end; full-fledged Cloud-hosted applications at the *opposite* end. The application logic resides entirely at the latter, creating performance issues such as excessive energy consumption and high latencies. To ameliorate these, Virtual Resources allows developers to push a slice of the application logic to intermediate IoT devices. *Virtual sensors* allow one to process data acquired by a set of sensors according to programmer-provided functions, and to export the results through a RESTful interface. *Virtual actuators* operate dually, by providing a RESTful interface to operate a set of actuators according to programmer-provided functionality. Virtual Resources thus create a continuum between physical resources and Cloud-hosted applications, which developers can freely exploit. For example, they can push sensor data processing to IoT devices close to the physical sensors, reducing the amount of data to transmit and thus saving energy. We describe the key concepts of Virtual Resources and their realization in a CoAP prototype atop resource-constrained devices. Experimental results gathered through cycle-accurate emulation indicate that Virtual Resources enable better performance than Cloud-centric architectures, while retaining the RESTful interaction pattern. For example, energy consumption in representative scenarios improves up to 40%, while control loop latencies reduce up to 60%.

5.1. Introduction

The Internet of Things is expected to play a key role in a range of domains, from factory automation to health-care [**AIM10**]. These applications are enabled by embedded sensors and actuators able to exchange data with the larger Internet.

Applications and devices. Building automation is a paradigmatic scenario where IoT technology is employed. The system's goal is typically to intelligently control electrical appliances such as lights and HVAC (Heating, Ventilation, and Air Conditioning) to reduce energy expenditures, while retaining the user comfort. To this end, applications gather data from sensors; for example, to measure the appliances' generated loads or to detect people's presence, and use these data to affect the environment by controlling the relevant actuators.

In most scenarios, sensed data are not used in raw form. Rather, they are typically processed through multiple stages until a higher-level information is obtained that is useful to take smart decisions. For example, the appliances' generated loads are aggregated based on their location;

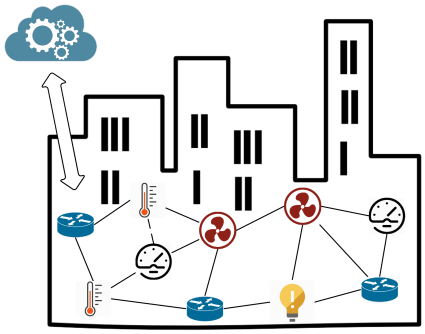


FIGURE 5.1. Cloud-centric IoT.

for example, public spaces as opposed to private offices, as these demand different control strategies. The commands sent to actuators also depend on their characteristics. For example, some appliances are not critical to a building's operation, such as those providing accessory services to the inhabitants [**Whi13**], and may be safely dimmed or switched off to save energy.

Adopting IoT technology for these applications can break the isolation of systems, and enable a more integrated approach. For example, to perform load balancing, different buildings may be jointly controlled by matching their overall energy consumption against that of their neighborhood. Moreover, offering the ability to sense or to actuate through standard-compliant communication networks enables a better re-use of the deployed devices, which may be seamlessly accessed by multiple applications. To lower costs and to ease installations, however, these devices typically run on batteries and communicate wirelessly. As a result, they can only feature 8- or 16-bit microcontrollers, few kB of RAM, and low-bandwidth radios [**JHVdV+09**]. Further, because of the limited communication range affordable with little energy, they often form *multi-hop* networks to ensure overall connectivity through intermediate devices operating as message routers.

These observations are not germane solely to building automation. For example, smart-city applications such as intelligent lighting, traffic control, and structural monitoring exhibit similar traits [**AIM10**]. In this context, for example, IoT technology may allow different public institutions to share the sensing infrastructure, offering a multitude of advanced services to the citizens.

State of the art. Despite the lack of well-established solutions to architect IoT software, specific solutions are distinctly emerging. One such approach arguably gaining momentum is the so called "Cloud-centric IoT" [**KMO12**], intuitively depicted in Figure 5.1. Sensors and actuators expose application-agnostic elementary functionality through RESTful interfaces. In contrast, the application logic is entirely deployed in the Cloud. By doing so, applications benefit from virtually-unlimited computing resources, whereas developers enjoy the range of development tools and integrated services made available by Cloud providers.

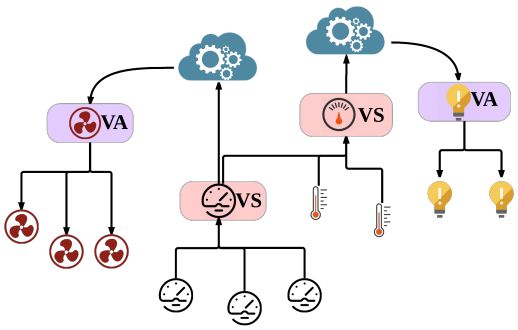


FIGURE 5.2. VIRTUAL RESOURCES applied to building automation. (VA is a virtual actuator, VS is a virtual sensor).

The Cloud-centric architecture, however, also comes with disadvantages. For example, in the building automation scenario, the application logic is unlikely interested in the individual readings of each and every power meter that records the energy consumption in public areas. Rather, only the sum of those readings is relevant to determine whether the energy consumption is excessive. Using a Cloud-centric architecture, the application needs to probe every sensor one by one, to compute the sum afterwards at the Cloud. This likely increases the network traffic within the resource-constrained network. As radio communication is the most energy-draining functionality on mainstream IoT devices, excessive network traffic severely impacts a device's lifetime.

In the same scenario, based on sensor data, the application may decide to dim the lights in public areas to reduce a building's energy consumption. As before, using a Cloud-centric architecture, this requires a separate call to every individual light controller, which would again impact the energy consumption of the IoT devices as in the case of the power meters. Worse, latencies would grow as well, as the queries need to travel from the Cloud across a complex infrastructure before reaching the IoT devices in the field.

Contribution and road-map. To remedy these issues while retaining RESTFUL interactions, we present VIRTUAL RESOURCES: an architectural solution for IoT applications that contrasts the distinct separation between physical devices and Cloud-hosted applications. Using VIRTUAL RESOURCES, developers can relocate slices of the application logic to any intermediate IoT device, such as those used as message routers inside the wireless network. This occurs through the concepts of *virtual sensor* and *virtual actuator*, depicted in Figure 5.2.

Using a *virtual sensor*, developers can acquire data from a set of physical sensors, process the data according to a programmer-provided processing function, and export the results through a RESTFUL interface akin—or even identical—to a physical device. Developers in building automation may define a "total-energy-public-areas" sensor built to provide the sum of the readings of all power meters in public areas, and deploy the virtual sensor on any IoT device between the physical sensors and the Cloud. The Cloud-hosted application interacts only with one virtual sensor to obtain the data.

Similarly, using a *virtual actuator*, developers can offer a single entry point for the Cloud-hosted application to control the lights in public areas, through a RESTful interface similar to that of a physical actuator. In the aforementioned scenario, for example, developers may define a "lights-public-areas" actuator to drive all lights in public areas at once, and deploy the virtual resource on any IoT device. The Cloud-hosted application can perform a single call to the *virtual actuator* instead of a varying number of calls depending on how many light controllers are deployed in public areas.

Using Virtual Resources thus provides a range of key benefits:

- Network resources are better utilized. The ability to push a slice of the application processing to intermediate IoT devices enables, for example, to reduce the amount of data coming from sensors. Because of the devices' characteristics, reducing network traffic prolongs the system lifetime. Similarly, lower network traffic is less likely to generate congestion within the IoT network, improving on the application's perceived latency.
- The Cloud-hosted application becomes simpler. For example, driving one actuator for all lights in public areas is easier than a varying number of individual actuators. Notably, the set of resources representing inputs (outputs) for a *virtual sensor* (*actuator*) is evaluated only at run-time. Therefore, changes in such sets are dealt with transparently w.r.t. the Cloud-hosted application.
- Rather than polluting the Cloud-hosted application with low-level concerns, Virtual Resources allow developers to separate the latter out and to move it to other devices. This may help achieve a better separation of concerns. For example, the company manufacturing the IoT devices may also provide a library of virtual resources useful in paradigmatic scenarios, such as the the "total-energy-public-areas" sensor, used by domain-experts when implementing the high-level application logic at the Cloud.

The remainder of the chapter unfolds as follows. Section 5.2 illustrates our design of Virtual Resources, which provides a RESTful interface also to manipulate the architecture's elements besides interacting with them. Section 5.3 describes the concrete use of Virtual Resources through the programming support we provide. We illustrate the underlying implementation in Section 5.4, which includes custom heuristics to determine where to deploy a virtual resource in a generic multi-hop wireless network. Our quantitative evaluation, reported in Section 5.5, demonstrates significant performance improvements in energy consumption and application latencies using Virtual Resources. We conclude the chapter with a brief survey on efforts close to ours in Section 5.6, and by offering concluding remarks in Section 5.7.

5.2. Design

We design the concrete realization of Virtual Resources in a way that manipulating its elements is accomplished using RESTful interactions as well.

Virtual Resource Directory	Methods
/periodic_sensor	GET | POST
/vs_instance1	GET
/period	PUT | GET
/processing	PUT | GET
/sliding_window_sensor	GET | POST
/vs_instance2	GET
/processing	PUT | GET
/window	PUT | GET
/simple_actuator	GET | POST
/va_instance1	PUT
/processing	PUT | GET

FIGURE 5.3. Virtual resource directory example.

Our design revolves around three concepts, hierarchically organized: *i) templates*, *ii) instances*, and *iii) configuration* resources. Virtual resource templates abstractly specify the services offered by a virtual resource, fostering re-use. Virtual resource instances are the operational counterpart of templates, and are derived from templates, that is, every instance is a sub-resource of the template it is derived from. A generic instance can be configured by means of configuration resources, located in the next level of the hierarchy. This hierarchical organization is reflected in a virtual resource *directory*, exemplified in Figure 5.3, whose goal is to provide an entry point for the creation, configuration, and use of the virtual resources, by means of RESTful interactions.

Virtual resource templates. The templates specify three aspects: *i)* the resources of interest associated with the virtual one, for example, what actuators are the target output of a *virtual actuator*; *ii)* the structure of the corresponding instances, that is, the operations offered by a virtual resource's interface and its configuration sub-resources; *iii)* the distributed behavior of the virtual resource, for example, whether a *virtual sensor* pulls data from the input resources or the latter periodically push data.

As for point *i)*, the input/output resources concurring to create a virtual one are defined by specifying a list of desired attributes to be found in the resources of interest. Such a list is taken as a Boolean constraint to be matched against application-level attributes of other resources. For example, a *virtual sensor* may be defined to consider as input all exiting resources whose `type` attribute equals `power_meter`, and whose `location` attribute matches `public_areas`.

The template also defines the configuration resources of all corresponding instances, as described in point *ii)*. These resources allow to change or to tune the behavior of a specific instance. For example, a configuration resource may be defined to set the period at which a *virtual sensor* offers a new reading. Every instance always features at least one configuration resource that binds the *processing function* applied to input data or to output commands. For example, the processing function for a *virtual sensor* may perform a form of aggregation over the input data. For a *virtual actuator*, the processing function may perform the command translation necessary to interact with the output resources.

As for point *iii)*, the template also encodes the distributed behavior of the virtual resource. This corresponds to an underlying implementation of distributed functionality, typically carried

out by a programmer with specific knowledge of the communication protocols in use. The encoding of what specific interaction pattern to employ; for example, push vs. pull in the case of a *virtual sensor*, is also part of this implementation. Here again is an opportunity for better separation of concerns. In traditional architectures, these aspects are entangled in the main application's processing. Virtual Resources cleanly separate this from the high-level logic, allowing developers with different backgrounds to take care of different functionality.

From templates to instances. At run-time, the virtual resource templates are explicitly published on the virtual resource directory. This operation makes them available to create virtual resource instances.

The virtual resource directory of Figure 5.3 contains three virtual resource templates, namely a *periodic-sensor*, a *sliding-window-sensor*, and a *simple-actuator*. Every template must support the GET and POST methods. The GET method returns a description of the template itself and of the interfaces of the derived virtual instances. This enables run-time discovery of the available virtual resources. The POST method performs the actual creation of the virtual instance, creating a new sub-resource of the template. The operation returns the URI of the new resource. The required configuration resources are automatically created as sub-resources of the instance.

Every template can host an arbitrary number of instance sub-resources. The instances offer the methods defined in the corresponding template. The example in Figure 5.3 includes three instances, `vs_instance1`, `vs_instance2`, and `va_instance1`, namely, two virtual sensors and one virtual actuator. Instances of virtual sensors generally support a GET method to return the (virtual) sensor reading, whereas instances of virtual actuators typically support a PUT method to update the state of the target actuators. After the creation of an instance, the application can further configure its behavior by updating the configuration sub-resources, using PUT methods. Virtual resources also support a DELETE method to destroy an instance, which makes it disappear from the directory.

Once the instances are created and possibly configured, the application can use their services through their RESTful interfaces, exactly like if these were bound to physical devices. In doing so, the binding between a virtual resource and its input/output resources is evaluated dynamically, only when a service is requested on an instance. That means that the actual input or output set of a virtual resource may freely change at run-time; for example, as new nodes join or some fail because of battery depletion, and yet the main application would keep operating transparently to these dynamics.

Although we hitherto considered that the input/output data of virtual resources be physical devices, this is not a strict requisite. A virtual resource may use another virtual resource as input/output, creating hierarchies. This is as simple as creating a template that matches the attributes of other virtual resources. Moreover, virtual resources may act as RESTful proxies for legacy technology or non-IP sensing networks, masking the heterogeneity of the target devices. For example, a *virtual sensor* may use inputs from a database of historical sensor readings, whereas a *virtual actuator* may output commands on an industry-strength wired bus [**Fle**] by means of a proper translation. This would additionally offer a standard-compliant means to integrate non-IoT networks.

```
1 metersPublic = resource_set(Type='power_meter', location='public_area')
2 VsPeriodic(Input=metersPublic, name='per_power_meter_public')
3 lightsPublic = resource_set(Type='light_control', location='public_area')
4 VaSimple(Output=lightsPublic, name='lights_control_public')
```

FIGURE 5.4. Example PyoT code to publish virtual resource templates onto the directory.

```
1 metersTempl = Resource.get(name='per_power_meter_public')
2 vs = metersTempl.POST('vs')
3 vs.fun.PUT(sumCode)
4 vs.period.PUT('60')
5 lightsTempl = Resource.get(name='lights_control_public')
6 va = lightsTempl.POST('va')
7 va.fun.PUT(translationCode)
```

FIGURE 5.5. Example instantiation of virtual resources and subsequent configuration.

5.3. Programming

We use the Constrained Application Protocol (CoAP) at application level. CoAP is an application-layer UDP-based protocol designed by IETF to enable service-oriented interactions with resource-constrained devices. CoAP implements RESTFUL interactions, resource abstraction, and URIs, while avoiding most the complexities and overhead of HTTP. It also supports built-in discovery of services and resources.

We implement the virtual resource directory as a CoAP server. Virtual resources are represented as CoAP resources. These support the RESTFUL operations described earlier, making it possible for applications to manipulate virtual resources through CoAP methods. We use PyoT, described in Chapter 3, as support to the run-time execution of virtual resources. PyoT abstracts sensors and actuators as a set of software objects qualified by application-level attributes; for example, the type and the location, which are instrumental to define the input/output resources for templates. PyoT objects can be used interactively using a shell, which provides a complementary means to manipulate virtual resources, in addition to explicit CoAP calls.

Instantiating virtual resources. Figure 5.4 shows a sample PyoT program to publish two virtual resource templates onto the directory. The code defines a set of input resources for a *virtual sensor* in line 1, using a built-in `resource_set()` method. This takes as input a list of key-value pairs that determine the constraints the resources of interest must match. In this example, the set of resources includes the sensors operating as "power meters" in "public areas".

The set of resources identified in `metersPublic` is used as a parameter to `VsPeriodic` in line 2, which publishes a *periodic-sensor* template on the directory with a unique name. Our PyoT implementation of the *periodic-sensor* template asynchronously collects data from the input resources, independent of whether the virtual resource is called. When the latter receives a GET request, the virtual sensor returns the value obtained by applying the processing function on the data of the last time period. The code unfolds similarly in lines 3 and 4 for the virtual actuator instantiated from the *simple-actuator* template, which targets the "light controllers" in

```
1 def control_app():
2   while True:
3     meter_values = vs.GET()
4     new_output = control_fun(meter_values)
5     va.PUT(new_output)
```

FIGURE 5.6. An example control application using virtual resources.

```
1 Actuators = get_actuator_list()
2 setpoint = get_setpoint()
3 if type(setpoint) == Power:
4   setpoint = power2lux(setpoint)
5 if len(Actuators) > 0:
6   newSetpoint = setpoint / len(Actuators)
7   set_actuator(newSetpoint)
```

FIGURE 5.7. Processing function example for a *virtual actuator*.

"public areas". Our PyoT implementation of the *simple-actuator* template applies the processing function to the command received through a PUT request, and forwards the results to the target resources.

Figure 5.5 exemplifies the actual instantiation. In line 1, the code obtains a reference to the template of the *periodic-sensor* published earlier. This occurs by querying a generic `Resource` object. Alternatively, one may query the directory to discover the available resources dynamically. In line 2, we create the actual instance of virtual resource, by passing the name of the new object as a parameter of the POST operation. The newly-created instance is then configured in line 3 and 4 by updating the available configuration sub-resources: `fun` for the processing function, and `period` for the sampling period. Similarly, in lines 5 to 7, a *virtual actuator* named `va` is instantiated and configured.

Tying things together. Figure 5.6 shows an example application, possibly running on the Cloud, that uses the created virtual resources. The application uses the instances named `vs` and `va`—corresponding to those created in Figure 5.5—to retrieve values from the virtual sensor and to send commands to the virtual actuator. Figure 5.6 makes it apparent that the Cloud-hosted application is extremely simplified using VIRTUAL RESOURCES. The code only includes the high-level logic, whereas the lower-level details are delegated to the virtual instances and hidden from the developers.

To express how to manipulate input and output data for virtual resources, developers implement dedicated processing functions. These are coded in Python, as they also run atop PyoT. We provide a specific API to allow programmers to access input and output data within a processing function. For a *virtual sensor*, the API gives access to the list of values retrieved by the input sensors. For a *virtual actuator*, it allows to obtain the list of currently available output devices and to define a setpoint for them.

Figure 5.7 shows an example processing function executing a command translation on the *simple-actuator*. In this case, the function divides the setpoint given to the virtual actuator by the

number of active output actuators, after applying a translation between different measurement units. Functions `get_actuator_list()`, `get_setpoint()` and `set_actuator()` are part of the API we provide to VIRTUAL RESOURCES developers. They serve to retrieve the list of active output resources, to read the command sent to the virtual actuator, and to send a new command to the output resources, respectively. Similar APIs also exist for virtual sensors. Importantly, since the actual binding with input/output resources is dynamic, their number, as returned for example by `len()` in Figure 5.7, may dynamically change between two executions of the same processing function.

5.4. Run-time Support

Our prototype targets WisMote devices, equipped with an MSP430 micro-controller with 16 kB of RAM and 256 kB of program memory. The nodes run a network stack that includes CoAP, 6LoWPAN to provide IPv6 connectivity over low-power wireless, and IEEE 802.15.4 at link layer.

Distributed processing. The key functionality at run-time is to manage the distributed execution of processing functions at the instantiated virtual resources. Enabling this functionality on resource-constrained devices is a challenge per se. These devices do not run general-purpose operating systems, and re-configuring the running code may, in many cases, require a complete replacement of the deployed binaries [**WZC06**].

We use T-Res [**APP13**] to allow developers using VIRTUAL RESOURCES to dynamically allocate processing functions to arbitrary nodes in the IoT network. T-Res offers a run-time environment that enables in-network processing in CoAP networks. A T-Res *task* is a deployable unit of processing that can be dynamically installed on CoAP-enabled nodes. T-Res tasks are characterized by their input sources, the data processing they perform, and the destination of their output. Both the input sources and the output destination must be CoAP resources and are specified by means of their URIs. Using the RESTFUL interface of T-Res nodes, PyoT allows us to configure and re-configure the processing functions used by the created virtual resource instances. Upon instantiating a virtual resource, PyoT delivers through T-Res the processing function possibly set by the programmer to the node where the virtual resource is going to execute. The corresponding Python is uploaded to the node, and the input sources and output destination of the T-Res task are configured using the URIs of the corresponding input/output resources of the virtual instance.

We extend both PyoT and T-Res to support the RESTFUL design of VIRTUAL RESOURCES. Specifically we implement the virtual resource directory on-top of PyoT and enable the creation of VIRTUAL RESOURCES templates and instances through its shell. As for T-Res, we implement generic support to the execution processing functions.

Instance placement. The performance of a system using VIRTUAL RESOURCES is influenced by what device T-Res chooses to execute the processing of a virtual resource. This is essentially due to the way the underlying routing protocols operate. The standardized solution for IPv6-enabled IoT devices is RPL [**ITN13**], the Routing Protocol for Low-Power Lossy Networks. RPL

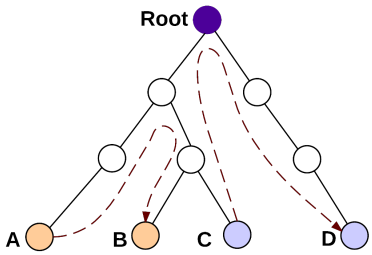


FIGURE 5.8. RPL graph and example routing between nodes.

superimposes a Directed Acyclic Graph (DAG) atop the multi-hop physical topology, as shown in Figure 5.8. The DAG is rooted at a single device, typically the one operating as the border router that bridges to and from the larger Internet.

When two devices communicate using RPL, messages travel only along the links in the DAG. If source and destination share an ancestor other than the root, this can shortcut the path, as is the case when transmitting from node A to node B in Figure 5.8. In contrast, if source and destination do not share an ancestor other than the root, the message needs to travel up to the latter before being forwarded downwards to the destination. This is the case when transmitting from node C to node D in Figure 5.8.

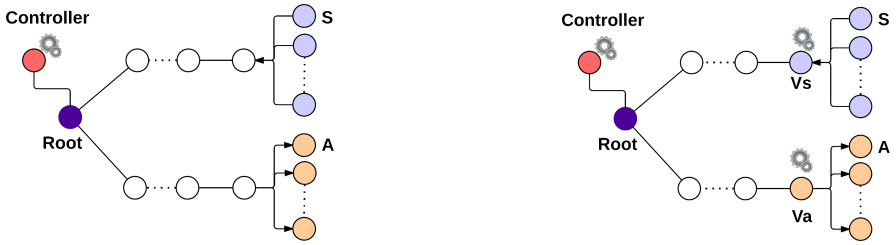
RPL is indeed optimized for scenarios where most of the traffic goes through the root. However, this is not necessarily the case using VIRTUAL RESOURCES, which instead generate peer-to-peer traffic among arbitrary nodes; for example, between a virtual sensor and its input resources. If the device hosting the virtual sensor and its input resources happen to be located like nodes C and D in Figure 5.8, the potential savings in network traffic may disappear.

We thus develop a simple heuristic to drive the placement of virtual resource instances onto the physical devices, applied when RPL has converged to a stable state. This does happen in many scenarios akin to those we target [**ITN13**]. We take as input the current RPL graph and the position of the existing physical resources on it. We then determine the positioning of the newly-created instances based on their input or output resources[1], based on a few simple policies:

- **P1:** virtual sensors (actuators) are positioned on the first common ancestor found by walking the graph from the input (output) resources to the root;
- **P2:** if multiple such ancestors exists, we take the one farthest from the root;
- **P3:** if common ancestors other than the root do not exist, the virtual resource is placed on an available node closest to the root.

P1 ensures that data coming from sensors is processed, and thus reduced in size, as soon as possible on the way to the Cloud, or viceversa that data coming from the Cloud and addressed to actuators is demultiplexed as close as possible to the output devices. **P2** break ties by attempting

[1]The heuristic may also re-evaluate the positioning of existing virtual resources in case the underlying RPL graph changes.



(a) Cloud-centric design.

(b) Virtual Resources design.

Figure 5.9. Application designs used throughout the evaluation. (VA is a virtual actuator, VS is a virtual sensor, A is a physical actuator, S is a physical sensor).

to amplify the effects of **P1** on a higher number of hops from/to the root. Finally, **P3** is a fall-back solution in case the common ancestor is indeed the root, which is however seldom usable because its resources are taken by the border routing functionality.

In Section 5.5, we quantitatively measure the effects of applying these policies on a random placement of virtual resources. Such simple policies already provide significant improvements; nevertheless, this is in fact an instance of the well-known task allocation problem [**SSXY06**], where a body of work already exist. We plan to leverage the existing literature to design a provably optimal solution.

5.5. Evaluation

We aim at understanding the benefits and performance of Virtual Resources over the Cloud-centric architecture. To this end, Section 5.5.1 describes the settings enabling the comparison, whereas Section 5.5.2 reports on the results.

5.5.1. Setting and Metrics

We realize two functionally-equivalent versions of the building automation application. One design employs a Cloud-centric architecture, as in Figure 5.9a. The application logic residing on a controller node directly accesses a varying number of physical sensors and actuators. The other design uses Virtual Resources, as in Figure 5.9b. We instantiate a virtual sensor and a virtual actuator to process sensor data and to drive the actuators, respectively. The virtual sensor uses a *sliding-window-sensor* template, which computes an average over a sliding window of sensor values asynchronously collected from the physical resources.

Besides qualitatively comparing the implementations, we run experiments using Cooja/M-SPSim [**EÖF+09**]: a wireless network simulator catering for cycle-accurate executions of Wis-Mote devices. We simulate a network of 21 nodes, representative of existing IoT installations in medium-size buildings [**Whi13**] These include a border router acting as the RPL root, a

variable number of physical sensors and actuators, a variable number of intermediate nodes operating as message routers, and a control node that models the Cloud-hosted application. The implementation on the latter is written in C/Contiki [**DGV04**], which is directly supported by Cooja/MSPSim. Differently, running the application in an actual Cloud would require taking measures across a hybrid system, unnecessarily complicating the setup. The scope of our work, indeed, does not stretch to the standard Internet.

We first run simulations on top of a controlled network topology, akin to the one in Figure 5.9b. This means that physical resources are placed at the leaves of the RPL graph, and virtual resources are placed on a node that is both a common ancestor of, and closest to the physical resources. This somehow represents a favorable case for Virtual Resources, which is however not that unrealistic. Indeed, it is reasonable that devices used as input/output of virtual resources be also physically co-located; for example, all power meters on the same floor, and that it would be possible to place the virtual resource nearby. Nevertheless, this setting is solely instrumental to compare the trends—not the absolute performance—of Virtual Resources against the Cloud-centric architecture, *without* the bias due to the physical topologies. To this end, we vary the number of physical sensors and actuators and the hop-distance between the RPL root and the virtual resources.

Next, we perform experiments with arbitrary topologies, to understand the absolute performance in settings that arguably represent a *worst-case* for Virtual Resources. Indeed, the underlying physical topology is randomly generated, with the only constraint of ensuring overall (multi-hop) connectivity, and there is initially no control over the placement of virtual and physical resources. This means that, for example, two sensors that may be somehow related at the application level, such as a power meter and the light controllers on the same floor, may be placed totally apart in the RPL graph. We then apply the heuristic of Section 5.4 to reposition the virtual resources, and measure the improvements compared to the initial placement.

To assess the performance, we measure: *i)* the *control loop latency*; *ii)* the *inter-actuator latency*; and *iii)* the *energy consumption*. The control loop latency is the time between the start of the control loop at the controller and the time when a command is last received by an actuator. The inter-actuator latency measures the time between the first and the last command reception at the actuators, giving an indication of how uniformly the nodes affect the environment. The energy consumption is measured only when we run experiments using a radio duty-cycling (RDC) mechanism, which is responsible for turning the radio on and off depending on the traffic.

CoAP messages are sent reliably, that is, packets are re-transmitted if an acknowledgment is not received within a timeout. This suffices to complete all 500 iterations of the control loop we test in every setting.

5.5.2. Results

Qualitative comparison. Both versions include a setup phase. In the Cloud-centric design, this consists in discovering what physical sensors and actuators are available to execute the control loop. This processing is mandatory and must be executed periodically. Indeed, the set of available sensors and actuators may change over time; for example, because some devices deplete

their batteries or new devices are dynamically added, and the application must be made aware of these changes. Generally, using a Cloud-centric architecture, the functionality to maintain a catalogue of active devices along with their relevant application-level characteristics is necessarily part of the application's processing.

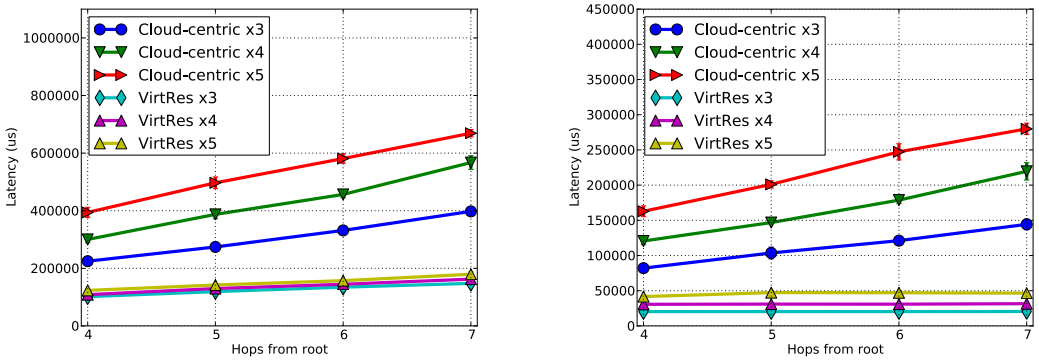
Using Virtual Resources, the setup includes the processing to create and configure the virtual instances. The setup may not be necessary, for example, if another application has already defined the same virtual resources and published them on the virtual resource directory. This way, the virtual resources would be immediately usable. More importantly, in the worst case this processing is performed only once, as managing the changes in the input/output resources is completely delegated to the Virtual Resources run-time support. This spares developers from including this functionality in the application processing.

Besides the observations above, interacting with two (fixed) virtual resources turns out simpler than accessing a varying number of physical resources. This reflects, for example, in the code complexity, which we indicatively measure here in LLOC although our implementation of the application logic does not employ standard Cloud technology, as described in Section 5.5.1. Using Virtual Resources, the application processing amounts to 96 LLOC for the control loop, plus additional 24 LLOC optionally required to instantiate and configure the virtual resources. The latter serve to create resources that become part of the directory and may be shared with other applications. In contrast, the Cloud-centric implementation requires 167 LLOC that only serve a single application.

Controlled topologies. We first analyze results when running the system with a 5 second control period. This greatly overestimates the dynamics of building automation systems, which typically control phenomena with slow dynamics, such as temperature or light. Control loop periods are therefore in the range of minutes. We intentionally push on this dimension to stress the systems. Nevertheless, the trends and observations we discuss next do apply—sometimes even with greater evidence—also when setting larger control periods. We momentarily disable radio duty-cycle to obtain measures of control loop latency and inter-actuator latency not affected by the added delays of energy saving mechanisms.

Figure 5.10 illustrates a sample of our results to understand the trends at stake. Figure 5.10a shows that using Virtual Resources significantly decreases the control loop latencies. The improvements grow as the hop-distance between the controller node and the physical resources increases. As this happens, indeed, the beneficial effect of the local processing performed at the virtual resources progressively amplifies. Using Virtual Resources, the system shows better scalability as the number of physical resources grows: the curves corresponding to a different number of physical sensors grow more slowly using Virtual Resources. Similar observations apply to Figure 5.10b, with the only difference that inter-actuator latencies remain constant using Virtual Resources, because the *virtual actuator* is always one hop away from the physical ones.

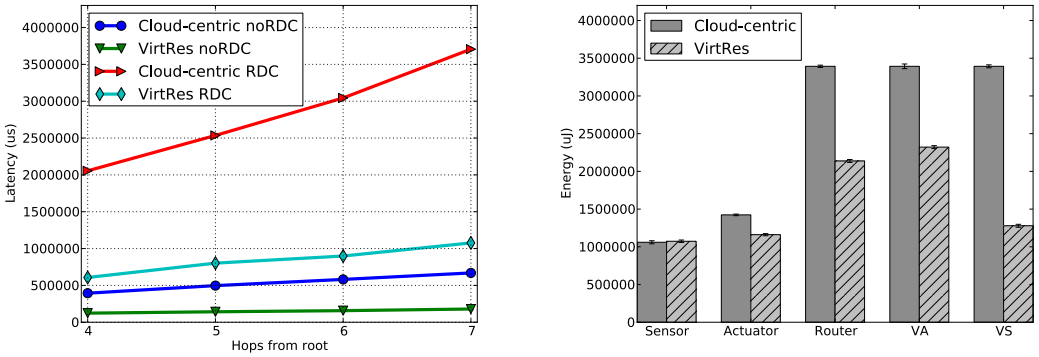
In contrast, Figure 5.11 studies how the performance changes in a more realistic configuration that uses radio duty-cycling to prolong the system's lifetime, in a sample setting with 5 physical resources. Figure 5.11a shows that the improvements in control loop latency amplify in favor



(a) Control loop latency.

(b) Inter-actuator latency.

FIGURE 5.10. Controlled topologies with 5 seconds period and no radio duty-cycling. *Control loop and inter-actuator latencies decrease using* VIRTUAL RESOURCES*, while the system scales more gracefully.*

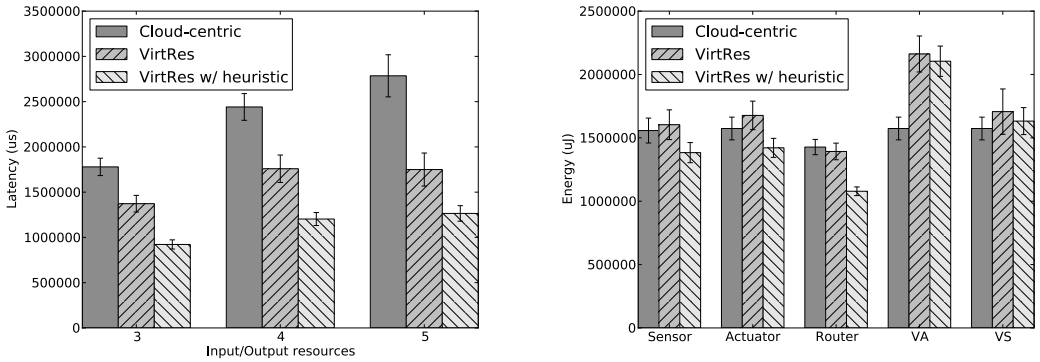


(a) Control loop latency.

(b) Per-round energy consumption.

FIGURE 5.11. Controlled topologies with 5 seconds period, using 5 physical resources. *The gains in control loop latency amplify in favor of* VIRTUAL RESOURCES*, which also enable energy savings for message routers and the controller node.*

of VIRTUAL RESOURCES. This is essentially because the per-hop delay grows when using radio duty-cycling; therefore, the impact of reduced traffic becomes more significant. Figure 5.11b illustrates the gains in energy consumption depending on the processing performed by a node. Intermediate nodes operating as message routers need to forward fewer messages using VIRTUAL RESOURCES, so they consume significantly less energy. The nodes running virtual sensors or virtual actuators, compared in Figure 5.11b to the highest energy consumption recorded when using the Cloud-centric design, also improve their energy efficiency. Physical resources consume about the same energy in that their processing is similar in the two designs.



(a) Control loop latency.

(b) Energy consumption.

FIGURE 5.12. Random topologies with 5 seconds period and radio duty-cycling. *Control loop latencies are more than halved with the heuristic placement; energy consumption reduces as well.*

Random topologies. Figure 5.12 summarizes the results obtained from more than 70 different randomly-generated wireless topologies. As already mentioned, these represent a worst-case for VIRTUAL RESOURCES, as the placement of virtual and physical resources is initially totally random.

The control loop latency, shown in Figure 5.12a, shows improvements for VIRTUAL RESOURCES even when considering the fully random placement. After applying the heuristic of Section 5.4 to re-position the virtual resources, the results further improve and the variability reduces. Ultimately, the control loop latency is more than halved using VIRTUAL RESOURCES. Similar considerations also apply to the inter-actuator latency in the same settings—not shown here for brevity—albeit the improvements are not as high. In both cases, the gains are enabled by the traffic reduction brought by VIRTUAL RESOURCES, while still maintaining the RESTFUL interactions.

Figure 5.12b reports demonstrates the energy improvements of VIRTUAL RESOURCES over the Cloud-centric architecture. As before, nodes operating as message routers reduce the energy expenditures because of the lower traffic load enabled by VIRTUAL RESOURCES. Different than before, however, also the physical sensors and actuators consume less energy when using VIRTUAL RESOURCES with the heuristic placement. As their location is randomly decided, it may happen that they are *not* placed at the leafs of the RPL graph, as before. In these cases, they often need to double as message routers as well, hence the improvements of the latter apply to them too.

Finally, again as a mere indication given we are not employing standard Cloud technology at the controller node for these experiments, we measure the processing load at this device. Using VIRTUAL RESOURCES, we record a 50% improvement in this figure as well, mainly due to simpler processing in the high-level application logic.

5.6. Related Work

Service-oriented architectures are being widely applied to abstract the "things" as software services [**THIG11**]; yet, very few approaches enable to relocate slices of the application logic inside the IoT network, as we do. Because of this, they miss out on the performance advantages we demonstrated in Section 5.5.

In this area, a work close to Virtual Resources is, for example, that of Mitton et al. [**MPPT12**], who present a concept of sensor virtualization applied to a smart-city use case. However, they do not provide the ability of moving part of the application logic inside the IoT network. Similar observations apply to the concept of "physical mashup" [**GTW10**], and to systems such as IBM's Node-RED (`nodered.org`), where sensor virtualization components can indeed be created, which however execute on a machine outside the IoT network. We did present a notion of virtual sensor and virtual actuator for stand-alone wireless embedded networks in earlier work [**CMP06**]. However, we did not tackle the problem of Internet integration, and designed the system based on custom languages and interfaces. Moreover, the equivalent of Virtual Resources processing functions had to be pre-programmed on the nodes, rather than dynamically allocated.

In contrast, the ability to dynamically push a slice of the application inside the IoT network raises the question of where to place the processing function. The heuristic we presented in Section 5.4, albeit already effective as we demonstrated in Section 5.5, is certainly not optimal. Several more sophisticated algorithms do exist [**BB03**], which we plan to adapt to the Virtual Resources run-time.

5.7. Conclusion

We presented the Virtual Resources architecture, which contrasts the traditional Cloud-centric designs by giving developers the ability to push slices of the application logic down to the IoT network. This is possible through the notions of *virtual sensor* and *virtual actuator*. By doing so, Virtual Resources provide several benefits to developers, including better utilization of network resources that results in higher energy efficiency and lower latencies, a simplification of the application logic at the Cloud, and better separation of concerns throughout the development process. We designed a RESTful interface to manipulate virtual resources and a CoAP-based prototype providing dedicated programming support. Our results indicate that Virtual Resources achieves a 40% improvements in energy consumption and a 60% improvement in control loop latency, while retaining RESTful interactions.

CHAPTER 6

Conclusions

THIS thesis discussed the role of programming abstractions in the development of Internet of Things systems. We introduced the new possibilities and the new requirements presented by multimedia applications in the context of Wireless Sensor Networks. First, we investigated the application scenario of Intelligent Transportation Systems, which is paradigmatic to describe the need for open systems, based on Internet standards. The adoption of 6LoWPAN and of the REST paradigm is just the starting point to achieve the expected impact of IoT technologies on mass scale. In the following we discussed on the necessity of using abstractions and tools to facilitate the management of networks of Smart Objects efficiently and scalably. We proposed PyoT, a system allowing to apply the macroprogramming paradigm to IoT resources. Thanks to the Resource abstraction programmers can easily develop modular and re-usable applications. In the following chapter we described two applications of IoT technologies taking advantage from the abstraction mechanisms previously described. The concepts proposed by PyoT proved to be crucial to bring the applications on the field. The Smart Factory scenario permits to augment the safety in industrial environments in which the access to dangerous areas is subject to restrictions. Differently, a new Intelligent Transportation System scenario demonstrates how adopting IoT standards supported by abstraction mechanisms it is possible to reach a better versatility of the applications and an increased ease of use and maintenance. On the one hand, thanks to the abstraction of the processing of the nodes, it is possible to dynamically reconfigure M2M systems, responding to the dynamic features of IoT networks. On the other hand the configuration and maintenance operation are facilitated.

Following the growing interest in Cloud-centric IoT architectures we focused on the problems related to such model and we proposed an alternative scheme based on the virtualization of IoT resources. Through the concept of virtual sensors and virtual actuators, software developers can effectively focus on the high-level functions of the applications. Virtual Resources hide low-level details, such as the interaction pattern with the sensors and actuators and the management of the resources of interest. As a result, the use of Virtual Resources promotes the separation of concerns throughout all the development phases. Most importantly it permits to conciliate the Cloud model with the distributed *in-network processing*: the first represents the most promising solution for the management of large-scale IoT infrastructures; the latter allows to retain good performance in terms of reduction of number of messages, energy saving, and latencies.

Bibliography

[AAB+12] A. Azzarà, D. Alessandrelli, S. Bocchino, P. Pagano, and M. Petracca. Architecture, Functional Requirements, and Early Implementation of an Instrumentation Grid for the IoT. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, June 2012.

[AAB+14] A. Azzarà, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano. PyoT, a macroprogramming framework for the Internet of Things. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*. IEEE, 2014.

[AAP+12] D. Alessandrelli, A. Azzarà, M. Petracca, C. Nastasi, and P. Pagano. ScanTraffic: Smart Camera Network for Traffic Information Collection. In *Proceedings of European Conference on Wireless Sensor Networks*, pages 196–211, Trento, Italy, February 2012.

[AAPP14] A. Azzarà, D. Alessandrelli, M. Petracca, and P. Pagano. Demonstration abstract: PyoT, a macroprogramming framework for the IoT. In *Proceedings of the 13th international symposium on Information Processing in Sensor Networks*, pages 315–316. IEEE Press, 2014.

[ABP+13] A. Azzarà, S. Bocchino, P. Pagano, G. Pellerano, and M. Petracca. Middleware solutions in WSN: The IoT oriented approach in the ICSI project. In *Software, Telecommunications and Computer Networks (SoftCOM), 2013 21st International Conference on*, pages 1–6, Sept 2013.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[AM15] A. Azzarà and L. Mottola. Virtual Resources for the Internet of Things. In *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT), Milano (Italy)*, December 2015.

[APN+10] Daniele Alessandrelli, Paolo Pagano, Christian Nastasi, Matteo Petracca, and Aldo Franco Dragoni. Mirtes: middleware for real-time transactions in embedded systems. In *3rd IEEE International Conference on Human System Interactions (HSI)*, pages 586–593, 2010.

[APP13] Daniele Alessandrelli, Matteo Petracca, and Paolo Pagano. T-res: Enabling reconfigurable in-network processing in IoT-based WSNs. In *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*. IEEE, 2013.

[APP15] A. Azzarà, M. Petracca, and P. Pagano. The ICSI M2M Middleware for IoT-based Intelligent Transportation Systems. In *International Workshop on COoperative Sensing for Smart MObility (COSSMO)*, 2015.

[AvA08] B. Arief and A. von Arnim. TRACKSS approach to improving road safety through sensors collaboration on vehicle and in infrastructure. In *68th IEEE Vehicular Technology Conference (VTC)*, pages 1–5, 2008.

[BB03] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN*, 2003.

[BBM+11] B. Barbagli, L. Bencini, I. Magrini, G. Manes, and A. Manes. An End To End WSN Based System For Real-Time Traffic Monitoring. In *8th European Conference on Wireless Sensor Networks (EWSN)*, 2011.

[BCS12] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2), 2012.

[BPP+11] S. Bocchino, M. Petracca, P. Pagano, M. Ghibaudi, and F. Lertora. Speed routing protocol in 6lowpan networks. In *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1–9, Toulouse, France, September 2011.

[BT06] A. Bratukhin and A. Treytl. Applicability of RFID and Agent-Based Control for Product Identification in Distributed Production. In *Proceedings of International Conference on Emerging Technologies and Factory Automation*, pages 1198–1205, Prague, Czech Republic, September 2006.

[CCCT06] Wenjie Chen, Lifeng Chen, Zhanglong Chen, and Shiliang Tu. WITS: A wireless sensor network for intelligent transportation system. In *1st International Multi-Symposiums on Computer and Computational Sciences (IMSCCS)*, 2006.

[CCO+11] Matteo Ceriotti, Michele Corrà, Leandro D Orazio, Roberto Doriguzzi, Daniele Facchin, Gian Paolo Jesi, Renato Lo Cigno, Luca Mottola, Amy L Murphy, Massimo Pescalli, Gian Pietro Picco, Denis Pregnolato, and Carloalberto Torghele. Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels. In *10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN/SPOTS)*, pages 187–198, 2011.

[CGB+11] Angelo P Castellani, Mattia Gheda, Nicola Bui, Michele Rossi, and Michele Zorzi. Web Services for the Internet of Things through CoAP and EXI. In *Communications Workshops (ICC), 2011 IEEE International Conference on*. IEEE, 2011.

[CGN+05] Jason Campbell, Phillip B. Gibbons, Suman Nath, Padmanabhan Pillai, Srinivasan Seshan, and Rahul Sukthankar. IrisNet: an Internet-scale architecture for multimedia sensors. In *13th annual ACM international conference on Multimedia*, pages 81–88, 2005.

[Che] C. Chen. Design of a Child Localization System on RFID and Wireless Sensor Networks. *Journal of Sensors*, 2010.

[CMP06] Pietro Ciciriello, Luca Mottola, and Gian Pietro Picco. Building virtual sensors and actuators over logical neighborhoods. In *Proceedings of the international workshop on Middleware for sensor networks*, pages 19–24. ACM, 2006.

[Dea03] W. Dean. PyMite: A Flyweight Python Interpreter for 8-bit Architectures, March 2003.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[DRC+10] D. De Donno, F. Ricciato, L. Catarinucci, A. Coluccia, and L. Tarricone. Challenge: towards distributed RFID sensing with software-defined radio. In *Proceedings of International Conference on Mobile Computing and Networking*, pages 97–104, Chicago, USA, September 2010.

[EÖF+09] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. COOJA/MSPSim: Interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[eri] Erika Enterprise RTOS. `http://erika.tuxfamily.org/`.

[ETS] ETSI. Machine-to-machine communications (m2m); interoperability test specification for coap binding of etsi m2m primitives. `http://www.etsi.org/deliver/etsi_ts/103100_103199/103104/01.01.01_60/ts_103104v010101p.pdf`. Accessed: 15-Sept-2014.

[Fle] ISO 17458—FlexRay Communications System. `goo.gl/kZhgYY`.

[GBL+00] P. Gai, E. Bini, G. Lipari, M. Di Natale, and L. Abeni. Architecture For A Portable Open Source Real Time Kernel Environment. In *Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, Nairobi, Kenya, November 2000.

[GKP+11] Vikram Gupta, Junsung Kim, Aditi Pandya, Karthik Lakshmanan, Ragunathan Rajkumar, and Eduardo Tovar. Nano-cf: A coordination framework for macro-programming in wireless sensor networks. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011 8th Annual IEEE Communications Society Conference on*, pages 467–475. IEEE, 2011.

[GTW10] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010.

[HAAIKR11] Mohammad Sajjad Hossain, ABM Alim Al Islam, Milind Kulkarni, and Vijay Raghunathan. μsetl: A set based programming abstraction for wireless sensor networks. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 354–365. IEEE, 2011.

[HM06] Salem Hadim and Nader Mohamed. Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7(3), 2006.

[HMnVMn+11] José M. Hernández-Muñoz, Jesús Bernat Vercher, Luis Muñoz, José a. Galache, Mirko Presser, Luis a. Hernández Gómez, and Jan Pettersson. Smart cities at the forefront of the future internet. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6656:447–462, 2011.

[HSM09] S. Hussain, S. Schaffner, and D. Moseychuck. Applications of wireless sensor networks and rfid in a smart home environment. In *Proceedings of Communication Networks and Services Research Conference*, pages 153–157, Moncton, Canada, May 2009.

[ITN13] Oana Iova, Fabrice Theoleyre, and Thomas Noel. Stability and efficiency of RPL under realistic conditions in wireless sensor networks. In *Personal Indoor and Mobile Radio Communications (PIMRC), 2013 IEEE 24th International Symposium on*, 2013.

[JHVdV+09] Michael Johnson, Michael Healy, Pepijn Van de Ven, Martin J Hayes, John Nelson, Thomas Newe, and Elfed Lewis. A comparative review of wireless sensor network mote technologies. In *Sensors, 2009 IEEE*, pages 1439–1442. IEEE, 2009.

[JRO08] R. Jurdak, A. Ruzzelli, and G. O'Hare. Multi-hop rfid wake-up radio: Design, evaluation and energy tradeoffs. In *Proceedings of International Conference on Computer Communications and Networks*, pages 1–8, St. Thomas, USA, August 2008.

[KDD11] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. A low-power coap for contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 855–860, 2011.

[KDD12] M. Kovatsch, S. Duquennoy, and A. Dunkels. Erbium (Er) REST Engine and CoAP Implementation for Contiki, April 2012.

[KED11] R. Kyusakov, J. Eliasson, and J. Delsing. Efficient structured data processing for web service enabled shop floor devices. In *IEEE International Symposium in Industrial Electronics*, Hune 2011.

[KHH05] Mauri Kuorilehto, Marko Hännikäinen, and Timo D. Hämäläinen. A survey of application distribution in wireless sensor networks. *EURASIP J. Wirel. Commun. Netw.*, 2005(5):774–788, October 2005.

[KKFS09] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. Smart Objects as Building Blocks for the Internet of Things. *IEEE Internet Computing*, 14(1):44–51, December 2009.

[KLD12] Matthias Kovatsch, Martin Lanter, and Simon Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *Internet of Things (IOT), 2012 3rd International Conference on the*, pages 135–142. IEEE, 2012.

[KLKY14] Jaewoo Kim, Jaiyong Lee, Jaeho Kim, and Jaeseok Yun. M2M service platforms: survey, issues, and enabling technologies. *Communications Surveys & Tutorials, IEEE*, 16(1), 2014.

[KLS14] Matthias Kovatsch, Martin Lanter, and Zach Shelby. Californium: Scalable cloud services for the Internet of Things with CoAP. In *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

[KMO12] Matthias Kovatsch, Simon Mayer, and Benedikt Ostermaier. Moving application logic from the firmware to the cloud: Towards the thin server architecture for the Internet of Things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 751–756. IEEE, 2012.

[LBS08] H. Liu, M. Bolic, and A. Nayakand I. Stojmenovic. Taxonomy and challenges of the integration of rfid and wireless sensor networks. *IEEE Network*, 22(6):26–35, November 2008.

[LWS04] R. Lin, Z. Wang, and Y. Sun. Wireless sensor networks solutions for real time monitoring of nuclear power plant. In *Proceedings of World Congress on Intelligent Control and Automation*, pages 3663–3667, Hangzhou, China, June 2004.

[MCB+15] J Manyika, M Chui, P Bisson, J Woetzel, R Dobbs, J Bughin, and D Aharon. Unlocking the Potential of the Internet of Things. *McKinsey Global Institute*, 2015.

[MD09] A. Mitrokotsa and C. Douligeris. *RFID and Sensor Networks: Architectures, Protocols, Security, and Integrations*, chapter Integrated RFID and Sensor Networks: Architectures and Applications, pages 511–535. Wireless Networks and Mobile Communications. CRC Press, Taylor & Francis Group, 2009.

[MFHH05] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[MJK+10] Suhas Mathur, Tong Jin, Nikhil Kasturirangan, Janani Chandrasekaran, Wenzhi Xue, Marco Gruteser, and Wade Trappe. Parknet: drive-by sensing of road-side parking

statistics. In *8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 123–136, 2010.

[MMN+11] M. Magrini, D. Moroni, C. Nastasi, P. Pagano, M. Petracca, G. Pieri, C. Salvadori, and O. Salvetti. Visual sensor networks for infomobility. *Pattern Recognition and Image Analysis*, 21:20–29, 2011.

[MP06] Luca Mottola and Gian Pietro Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *Distributed Computing in Sensor Systems*, pages 150–168. Springer, 2006.

[MP11] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, V, 2011.

[MPPT12] Nathalie Mitton, Symeon Papavassiliou, Antonio Puliafito, and Kishor S Trivedi. Combining Cloud and sensors in a smart city environment. *EURASIP journal on Wireless Communications and Networking*, 2012(1), 2012.

[MPV11] L. Mainetti, L. Patrono, and A. Vilei. Evolution of wireless sensor networks towards the internet of things: A survey. In *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on*, pages 1–6, Sept 2011.

[Mul07] G. Mulligan. The 6LoWPAN architecture. In *Proceedings of Workshop on Embedded networked sensors*, Cork, Ireland, June 2007.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498. ACM, 2007.

[NS09] A. Nasir and B. Soong. Environsense: An integrated system for urban sensing using rfid based wsn's. In *Proceedings IEEE Region 10 Conference*, pages 1–5, Singapore, January 2009.

[OBB+13] Tony O'donovan, James Brown, Felix Büsching, Alberto Cardoso, José Cecílio, Jose Do Ó, Pedro Furtado, Paulo Gil, Anja Jugel, Wolf-Bastian Pöttner, Utz Roedig, Jorge Sá Silva, Ricardo Silva, Cormac J. Sreenan, Vasos Vassiliou, Thiemo Voigt, Lars Wolf, and Zinon Zinonos. The ginseng system for wireless monitoring and control: Design and deployment experiences. *ACM Trans. Sen. Netw.*, 10(1):4:1–4:40, December 2013.

[PAB+15] F. Pacini, A. Azzarà, S. Bocchino, F. Aderohunmu, M. Petracca, and P. Pagano. Poster abstract: Performance analysis of data serialization formats in m2m wireless sensor networks. In *Proceedings of the 12th European Conference on Wireless Sensor Networks*, 2015.

[PBA+13] M. Petracca, S. Bocchino, A. Azzarà, R. Pelliccia, M. Ghibaudi, and P. Pagano. WSN and RFID Integration in the IoT scenario: an Advanced Safety System for Industrial Plants. *Journal of Communications Software & Systems*, 9(1), 2013.

[PG07] Fernando Perez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[PSM+12] P. Pagano, C. Salvadori, S. Madeo, M. Petracca, S. Bocchino, D. Alessandrelli, A. Azzarà, M. Ghibaudi, G. Pellerano, and R. Pelliccia. A middleware of things for supporting distributed vision applications. In *Proceedings of the 1st Workshop on Smart Cameras for Robotic Applications (SCaBot)*, 2012.

[RR08] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly, 2008.

[RV06] R. Rajagopalan and P.K. Varshney. Data-aggregation techniques in sensor networks: A survey. *Communications Surveys Tutorials, IEEE*, 8(4):48–63, Fourth 2006.

[SBK13] Zach Shelby, Carsten Bormann, and Srdjan Krco. Core resource directory. 2013.

[SH09] Stanislava Soro and Wendi Heinzelman. A Survey of Visual Sensor Networks. *Advances in Multimedia*, 2009:1–22, 2009.

[SKPK11] John Schneider, Takuki Kamiya, D Peintner, and R Kyusakov. Efficient XML interchange (EXI) format 1.0. *W3C Proposed Recommendation*, 20, 2011.

[SPB+14] Claudio Salvadori, Matteo Petracca, Stefano Bocchino, Riccardo Pelliccia, and Paolo Pagano. A low-cost vehicle counter for next-generation ITS. *Journal of Real-Time Image Processing*, pages 1–17, 2014.

[SPGP12] Claudio Salvadori, Matteo Petracca, Marco Ghibaudi, and Paolo Pagano. On-board image processing in wireless multimedia sensor networks: a parking space monitoring solution for intelligent transportation systems. In *Intelligent Sensor Networks*, pages 245–266. CRC Press, 2012.

[SSXY06] Sancho Salcedo-Sanz, Yong Xu, and Xin Yao. Hybrid meta-heuristics algorithms for task assignment in heterogeneous computing systems. *Computers & operations research*, 33(3):820–835, 2006.

[THIG11] Thiago Teixeira, Sara Hachem, Valérie Issarny, and Nikolaos Georgantas. Service oriented middleware for the Internet of Things: A perspective. In *Towards a Service-Based Internet*. Springer, 2011.

[TLCC11] Yi-Hsuan Tu, Yen-Chiu Li, Ting-Chou Chien, and Pai H Chou. Ecocast: interactive, object-oriented macroprogramming for networks of ultra-compact wireless sensor nodes. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 366–377. IEEE, 2011.

[TRL+09] Arvind Thiagarajan, Lenin Ravindranath, K. LaCurts, Samuel Madden, Hari Balakrishnan, S. Toledo, and Jakob Eriksson. VTrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 85–98, 2009.

[VD10] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting smart objects with ip: The next internet*. Morgan Kaufmann, 2010.

[Vin06] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

[Whi13] K. Whitehouse. The rise of the intelligent building. *IQT Quarterly*, (3), 2013.

[WJTL12] Cheng Wang, Changjun Jiang, Shaojie Tang, and Xiang-Yang Li. Selectcast: Scalable data aggregation scheme in wireless sensor networks. *Parallel and Distributed Systems, IEEE Transactions on*, 23(10):1958–1969, 2012.

[WPA07] P. Wilson, D. Prashanth, and H. Aghajan. Utilizing RFID Signaling Scheme for Localization of Stationary Objects and Speed Estimation of Mobile Objects . In *Proceedings of International Conference on RFID*, pages 94–99, Grapevine, USA, March 2007.

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110. ACM, 2004.

[WTB+12] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, March 2012.

[WZC06] Qiang Wang, Yaoyao Zhu, and Liang Cheng. Reprogramming wireless sensor networks: Challenges and approaches. *Magazine of Global Internetworking*, 20(3), 2006.

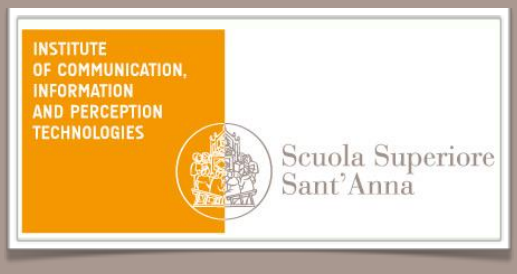
[WZWX11] R. Wang, Z. Zhang, J. Wang, and A. Xue. A new solutions for staff localization in chemical plant . In *Proceedings of International Conference on System Science and Engineering*, pages 503–508, Macao, June 2011.

[XSSG11] Z. Xiong, F. Sottile, M. Spirito, and R. Garello. Hybrid Indoor Positioning Approaches Based on WSN and RFID. In *Proceedings of International Conference on New Technologies, Mobility and Security*, pages 1–5, Paris, France, February 2011.

[XW08] Z. Xiaoguang and L. Wei. The research of network architecture in warehouse management system based on rfid and wsn integration. In *Proceedings of IEEE International Conference on Automation and Logistics*, pages 2556–2560, Qingdao, China, September 2008.

[YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3):9–18, 2002.

[ZLZ+08] L.Q. Zhuang, W. Liu, J.B. Zhang, D.H. Zhang, and I. Kamajaya. Distributed asset tracking using wireless sensor network. In *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1165–1168, Hamburg, Germany, September 2008.


INSTITUTE
OF COMMUNICATION,
INFORMATION
AND PERCEPTION
TECHNOLOGIES
Scuola Superiore
Sant'Anna