Design, implementation and experimentation of a protocol stack for the Internet of Things

Ph.D. Candidate: Angelo Paolo Castellani

Advisor: Prof. Michele Zorzi

Department of Information Engineering

University of Padova

A thesis submitted for the degree of

Dottorato di Ricerca in Ingegneria dell'Informazione

July 2012

A Dio,
per il Suo Amore.

A mia moglie Chiara,
per esser–ci sempre ed essere così speciale.

Ai miei figli, Marco, Miriam, Michele,
Marta Maria, Pietro, Francesco, Antonio,
Paolo, e Teresa ed a mio nipote Davide,
per la gioia che mi donano ogni giorno.

A mio padre Antonio e mia madre Marinella,
per avermi accompagnato a questa meta.

Abstract

The Internet of Things (IoT) is a novel paradigm that promises to offer us enhanced awareness of our surroundings through the introduction of processing, communication and sensing capabilities in everyday objects. Any object, which is now "smart", will help providing an augmented reality experience thanks to its machine–to–machine interactions with other smart objects and with the web services in the Internet cloud as well.

To make the IoT paradigm a reality, an interoperable, efficient and flexible Internet protocol stack is a key requirement; many research institutions and standardization bodies have been working to this end in the current years.

This thesis is centered around the technical challenges involved in the realization of such a protocol stack, including their definition and evaluation. Beyond the experimentation of current protocol proposals, this thesis also contains novel approaches, optimizations and architectures critical for building IoT nodes using cost, energy, and processing constrained devices, such as the ones readily available on the market for building IoT prototypes.

Implementation, simulation and experimentation of such technologies have revealed several issues preventing an efficient realization of such IoT systems. These issues have been addressed by the software and protocol architecture proposed in this thesis, which has proved to be more efficient than the state-of-the-art during simulations and field trials in realistic operation conditions.

Sommario

La Internet of Things (IoT), ossia la Internet delle Cose, è un nuovo paradigma tecnologico che promette di offrirci una migliore consapevolezza di cosa ci circonda aggiungendo agli oggetti di uso quotidiano capacità elaborative, comunicative e sensoriali. Ogni oggetto, reso così "intelligente", collaborerà nel provvedere all'utente un'esperienza di realtà aumentata, ottenuta grazie alle interazioni macchina–macchina con gli alri oggetti intelligenti, ed anche con i servizi presenti nella nuvola di Internet.

Per rendere la IoT una realtà, un requisito chiave è rappresentato dall'avere un insieme di protocolli Internet che siano al contempo interoperabili, efficienti e flessibili; molti istituti di ricerca e organismi di standardizzazione hanno lavorato a questo scopo durante questi anni.

Questo lavoro di tesi è centrato intorno alle complessità tecniche che minano la realizzazione pratica di tale protocol stack, includendo quindi la loro definizione e valutazione. Oltre la sperimentazione delle proposte protocollari correnti, questa tesi contiene nuovi approcci, ottimizzazioni e architetture critiche per la costruzione di nodi della IoT tramite dispositivi limitati in termini di costo, energia e capacità di elaborazione, come quelli che sono attualmente disponibili sul mercato per la costruzione di prototipi.

L'implementazione, la simulazione e la sperimentazione di queste tecnologie hanno rilevato numerose problematiche che possono prevenire la realizzazione efficiente di tale sistema IoT. Queste problematiche sono state gestite nell'architettura protocollare e software proposta in questa tesi, che infatti si è dimostrata essere più efficiente dello

stato dell'arte durante le simulazioni e la sperimentazione sul campo in condizioni operative realistiche.

Contents

CONTENTS

1

Introduction

The research presented in this thesis focused on the field of Internet of Things networking and, in particular, on the investigation of its relations with the well-known research topic of sensor networks, with the aim of producing a full-featured Internet-viable protocol stack. This stack has been realized as a proof-of-concept, demonstrating its feasibility on real devices and evaluating the performance of Internet-enabled smart sensor nodes.

This activity has been particularly inspired by the problem of defining a networking stack architecture for realizing efficient, interoperable and flexible smart objects.

In fact, in recent years the vision of ubiquitous, mobile, and pervasive computing systems has received extensive attention by the research community. As the result of a change in living habits worldwide, these days the demand for personal digital communication and computing devices is exponentially growing. Digital assistance is still limited in its interactions to regular Internet contents, which are in large part provided by humans feeding information systems with data, and where unattended operations are the exception rather than the norm.

In the future, a growing number of ordinary objects are expected to receive digital communication and computing extensions, to become *smarter* devices, helping their users in the interactions they have with them.

Any real-life system can be affected by this digital revolution, in fact, nowadays the application of the so-called Internet of Things (IoT) paradigm is proposed in many contexts.

In this thesis, the discussion will not be focused on a specific application, but rather we will discuss the technology as it could be applicable to multiple IoT scenarios. In fact, from a communication perspective, fundamental protocol stack characteristics are not impacted by specific use cases, and the whole dissertation has a benefit from a general purpose approach in its writing.

Practically speaking, however, in the aforementioned technical environment, a number of relevant technical issues have to be solved. The main problem are the severe constraints, in terms of amount of resources available on cheap constrained devices; since cost-effective nodes are desirably the majority of the hardware involved, system and protocol design is tightly limited to a very strict subset of feasible choices.

Moreover, the ubiquitous spatial deployment nature of smart devices, together with their inherent energy limitations, generally involve using an impaired communication infrastructure with characteristics similar to the ones available in the first years of the Internet computing. Additionally, nodes are required to share those limited communication resources with peer devices that cannot by themselves route their messages through the network without this active collaboration.

Last but not least, the importance of interoperable solutions has required special attention towards recent work done by standard organizations active in Internet protocols, and in particular a partecipation to the IETF standardization activity has been an integral part of this research.

The above grand vision has entailed the design of a system solution able to match real life hardware requirements of the involved devices, by active implementation and experimentation of all the proposed protocols and formats; this approach has required a large amount of time in the software design and evaluation, but has made it possible to gather an in-depth understanding of the whole problem, and has led to valuable architectural and practical guidance.

1.1 Structure of the thesis

This thesis work is structured as follows:

Chapter 2. gives an overview of seminal work done on lower layers of the system, required during our investigation and experimentation, and introduces relevant tools that have been vastly used throughout the thesis.

Chapter 3. is focused on prototyping the Internet of Things, by exploring the usage scenarios with their specific needs, and by eventually building the technical design of a working IoT system prototype.

Chapter 4. takes into account the impact and related complexity required for building functional IP networking on constrained devices, by proposing an efficient design of an IPv6/6LoWPAN stack. Moreover, it contains the design of a feasible congestion control approach for such networking environment allowing a scalable deployment of such technologies.

Chapter 5. contains the design and experimentation of application layer protocols for smart objects, starting from the seminal work done during the SENSEI project, up to the recent involvement in the subsequent standardization activities within the IETF. Moreover, it contains the discussion of an innovative implementation of an XML data encoder for IoT devices, and the related experimentation results.

Chapter 6. presents tools and technologies required to integrate IoT nodes into the ordinary HTTP-based Internet. Beyond considering the required protocol mappings, the chapter contains also the design of a novel web framework specifically targeting IoT application design.

Chapter 7. concludes the dissertation with some remarks and summarizes the most relevant results obtained in this work.

2

Background

In this chapter, we will provide an overview of the background work that has been conducted on link-layer technologies required for the investigation, implementation and testing of actual sensor network protocols.

First of all, in Section 2.1 we will discuss the design and deployment of the WISE-WAI testbed located at the University of Padova. This experimental facility allowed us to run real experiments, which guided us in the design of the protocol stack. Differently from many other similar testbeds, the WISE-WAI testbed spans across several buildings and floors of our campus, where the 2.4GHz ISM bandwidth suffers from a high number of users concurrently accessing it using different technologies.

In the second part of the chapter, we will present our original design of TinyNET, a tiny network framework for TinyOS. This piece of software allows the flexible integration of different networking components in the link-layer networking stack; we tested out its design upon the previously described testbed, to demonstrate its feasibility to easily realize advanced sensor network applications, containing routing and environmental monitoring capabilities.

2.1 The WISE-WAI Testbed [1]

The WISE-WAI Testbed is a large experimental platform of wireless sensor nodes. It has been designed in the context of the WISE-WAI project (see [21]) as a flexible and reconfigurable platform to test algorithms and protocols for WSNs,

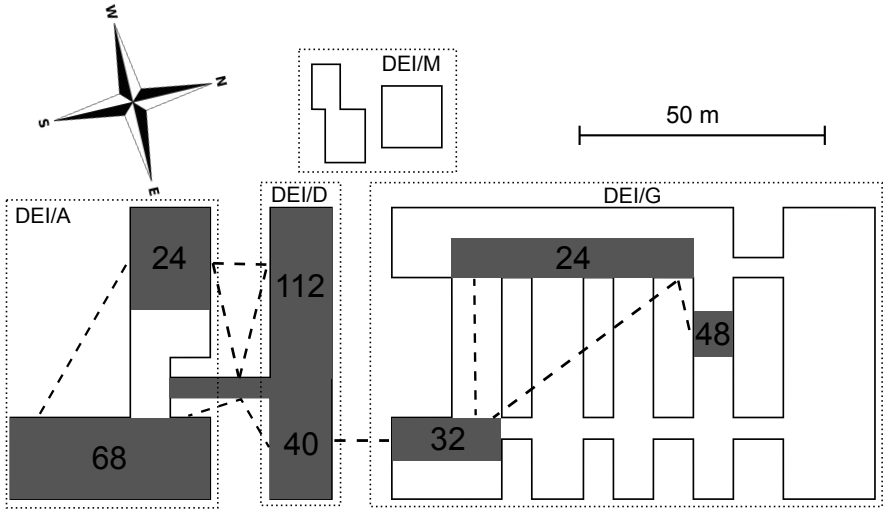

Figure 2.1: WISE-WAI Testbed Overview

and has been endowed of networking devices and solutions which allow wired communication with sensor devices; having a wired connection is a requisite for debugging or quick reprogramming purposes. The resulting network features an intermediate density of 10 to 20 nodes within the coverage area of any sensor, in case different (e.g., lower) densities should be needed, this number can be tuned by acting on the maximum transmit power of the nodes. We deployed the testbed so that every node is connected to the network backbone through a Universal Serial Bus (USB) cable. USB also provides power supply, this avoids battery wastage and continuous replacements during setup and test phases. However, during actual operations communications take place only through the wireless channel. The USB backbone also provides a cheap and fast way to log data for debugging, of performing general management and of programming nodes as well.

The areas currently covered with sensors entail lab rooms and offices, and are located over two separate buildings, and over two and three floor on these buildings, respectively. The connection between the buildings is completely wireless, although the configuration of the network may be changed so that separate sinks are assigned on each building: these nodes would have the special function to translate communications from the wireless to the wired domain, thereby reducing the wireless traffic if necessary. However, for most applications, the radio link

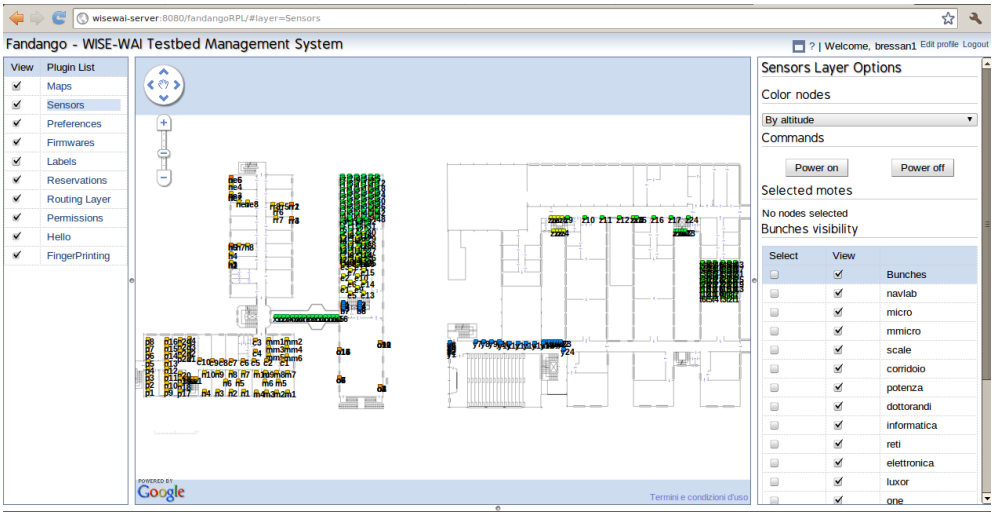

Figure 2.2: A screenshot of the web interface to the WISE-WAI testbed. The figure shows nodes of different colors, depending on the area where they are deployed. Each node has a unique label placed next to it (labels are more clearly visible when appropriate zooming and panning are operated). The plugins attached to the interface can be activated or deactivated through the check-boxes on the left, whereas the parameters and functions of the currently selected plugin (in this case the Routing plugin) are available on the right.

is kept active, so that the size of the network can be fully exploited.

It is worth noting that some of the sensors had to be modified in order to provide sufficient radio coverage, especially in bottleneck areas: the modification included the installation of an external antenna connector, the removal of a capacitor on the sensor board, the placement of a new capacitor in a different position, enabling the usage of the external antenna, and the actual installation of a displaced monopole antenna attached to the board via an antenna cable.

The network has been enriched with new management interfaces based on the well known Google Web Toolkit (GWT), which provides user friendliness through one of the most popular user interfaces to date (the same of Google Maps), and helps create and maintain complex front-end applications written in Java and Javascript. Ultimately, this allows to map network nodes to their installation places, and features the advantages of a modular environment enabling the development of customized applications, as well as of a large beta testing

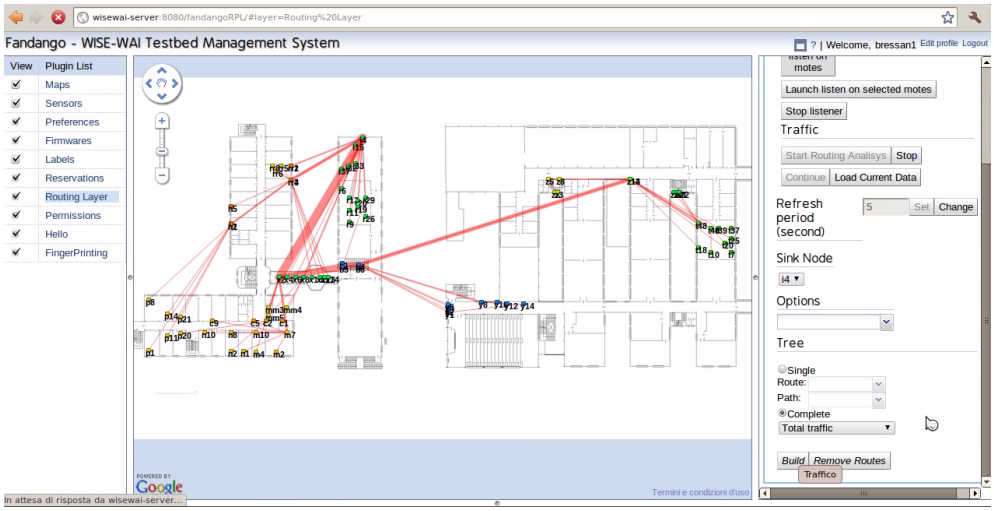

Figure 2.3: A screenshot of the Routing plugin ran over the Fandango interface. Only a subset of the nodes has been selected to run the application. The routing paths are depicted hop-by-hop by means of a red line connecting subsequent relays over the path. Some features of the lines can be associated to routing performance metrics (e.g., the thickness to the load over the respective link, the color to packet error rate).

community. GWT provided the framework for both Google Maps implementation and the integration of many other popular Google services such as Calendar, which we employed to provide resource booking functions. Whereas GWT perfectly supports the creation of client-side applications, on the server side we opted for Java Enterprise Edition (Java EE) in order to achieve an efficient and reliable server platform: in particular, we used the new, lightweight Java EE 6 Web Profile to create web applications, which we combined with the popular open source database MySQL. The applications allow users to build plugins capable of embedding graphics to show network resources and/or data. In particular it is possible to create interactive elements such as network nodes that can be clicked on to activate data representation plugins: an example can be seen in Figure 2.2. The interface and server-side platform are hosted on an Apache Tomcat server, a web container that provides a running platform for Web applications developed in Java. The graphical interface to the testbed has been named Fandango.

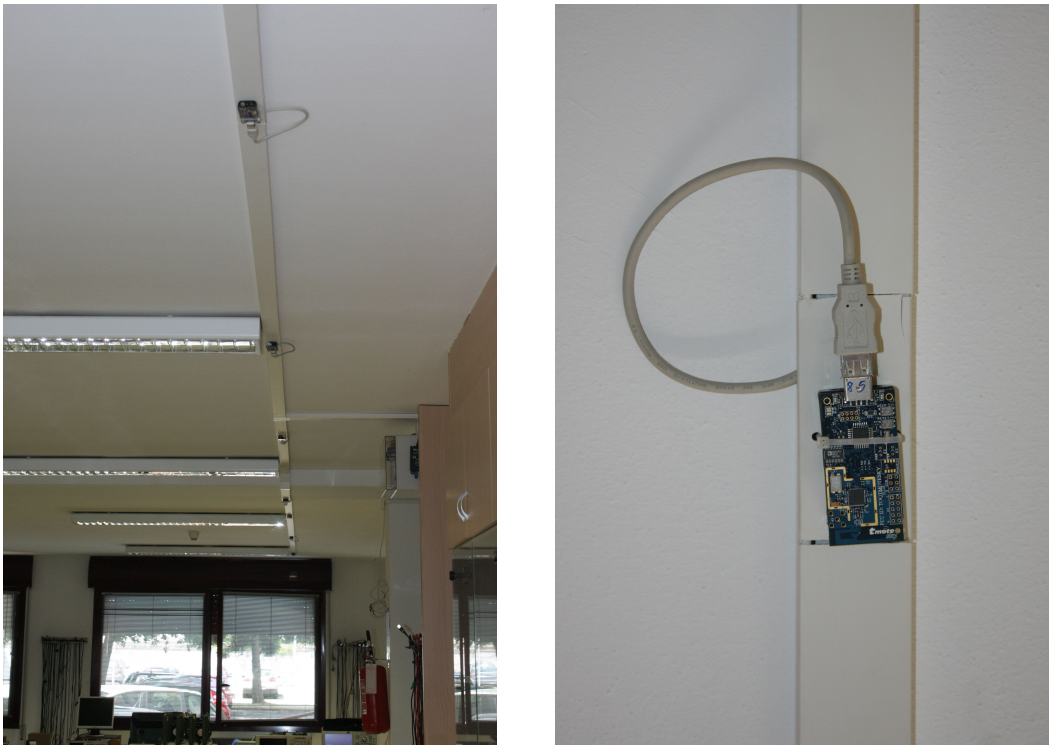

Figure 2.4: Indoor WISE-WAI Testbed Deployment

2.1.1 Related Work

In this section, we will provide a brief historical perspective, which guides the reader from the dawn of the Wireless Sensor Network (WSN) era to the latest developments of the IoT. In particular, we will focus on the most representative efforts on creating autonomous WSN testbeds and on integrating those networks in the Internet.

One of the earliest examples of sensor testbed has been Sensor Web [22], which provided a low cost implementation of wireless sensor networks based on commercial-off-the-shelf components. Sensor Web interconnected a few of the first WSNs and provided users with an ad hoc graphical environment for visualizing data.

In 2005 Sensor Web was enhanced with a distributed geospatial infrastructure based on a service oriented architecture [23]. This infrastructure leveraged on gateways capable of translating the proprietary communication protocol stacks

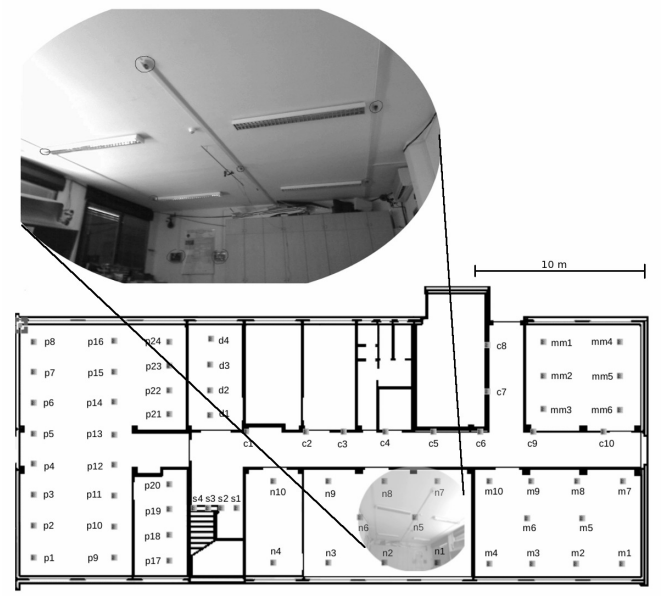

Figure 2.5: Node displacement map, with a close-up on the arrangement of nodes within one of the rooms.

of the different WSNs into Internet compliant messages based on both the HyperText Transfer Protocol (HTTP) and the eXtensible Markup Language (XML) for exchanging and enhancing information, respectively. From our experience, while this solution is successful in integrating many different networks, it is still dependent on the specific WSN realization, thus lacking the seamless interoperability that will be the main requirement for the IoT.

By 2005, another important testbed had been realized: MoteLab [24]. MoteLab is the first attempt to provide a research platform for independent researchers to test their own applications. In particular, MoteLab leveraged on TinyOS's hardware abstraction layer concepts, thus offering uniform interfaces for application design. However, the platform, which was also available for download, was intended for testbed management only and did not provide Internet connectivity to the nodes, nor web interfaces for node interactions.

In 2006 two other testbeds were realized: Kansei [25] at The Ohio State University and SignetLab at the University of Padova [26]. The former was designed with objectives similar to MoteLab, while the latter was our own first contribution to the IoT. Kansei allowed users to test their own software on the WSN

testbed. Although smaller, SignetLab not only allowed users to test their own applications, but also provided them with a simple Java Management framework which let users deploy custom plugins to control applications and interact with devices.

Again in 2006, Microsoft created the first portal website for real-time real-world sensor data, SensorMap [27]. SensorMap leveraged on geo-centric web services such as Windows Live Local and Google Maps to provide APIs to visualize spatially and geographically related data over a map interface.

Later integration examples, both from 2008, are SensorScope [28] and Smart Space Network [29]. At about the same time, we were extending the SignetLab sensor testbed to the whole Department of Information Engineering (DEI), now counting about 350 devices [1]. Also, in 2009 a RESTful architecture [30] was proposed, as well as two concrete implementations based on the Sun SPOT platform and on the Ploggs wireless energy monitors. In 2010, [31] proposed a federated testbed approach to interconnect heterogeneous hardware by virtualizing the physical testbed topology.

Concluding our chronology, Pachube [32] is possibly the most successful IoT integration framework into the Web and in 2011 reference [33] describes the Web of Things (WoT) architecture and best practices based on the RESTful principles that are similar to those that we leveraged on in the following section.

2.1.2 Technical Description

Testbed has been deployed according to a hierarchical organization, whereby all sensors are connected, via USB hubs, to tiny embedded computers that act as Node Cluster Gateways (NCGs, see Figure 2.7). USB connections are not used for actual communication, but rather provide power supply and log data for debugging purposes. During actual operations, communications take place only through the wireless channel. For the quick deployment of applications to be executed on the nodes, USB cables can also convey new application data. The NCGs are core elements of the network hierarchy, and interact with the nodes both in the upstream (node-to-gateway) direction, e.g., for reporting debug and log messages, and in the downstream (gateway-to-node) direction, e.g., to

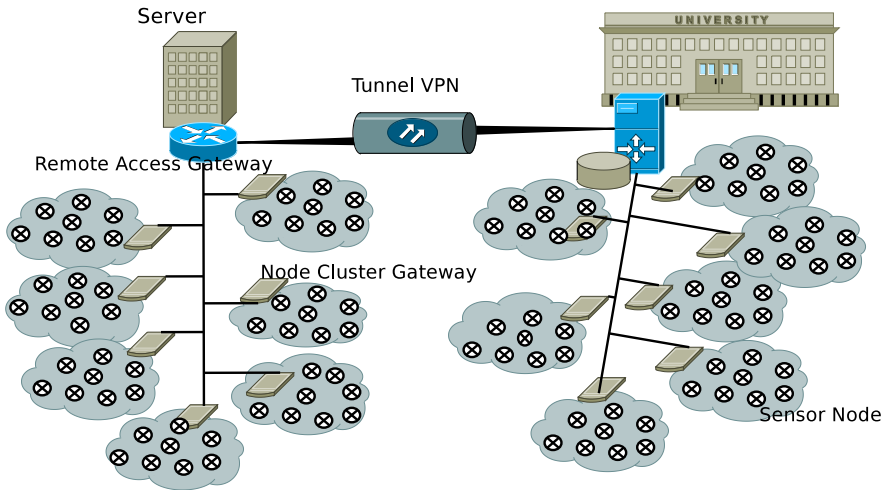

Figure 2.6: WISE-WAI Testbed Architecture

reprogram, reset, and power up or down the nodes. The latter functionality is provided by fully USB 2.0-compliant hubs and proves particularly useful by doubling as a sort of hard sensor reset. This is accomplished by powering off the port to which the sensor is attached. Thanks to this function, the sensors need not to be manually disconnected, in case they should not respond to software reset commands. NCGs can be reached from a central server through Virtual Private Network (VPN) connections, in order to carry out management tasks; otherwise, their presence is transparent to the user, who interacts with the network as though he were directly communicating to the sensors. While command line scripts are available for this purpose, an HTTP interface has also been developed to ease node programming, as well as other management tasks (power on/off, information retrieval, and so forth).

The aforementioned architecture (server–NCGs—USB hubs–sensors) is scalable and easy to extend; furthermore, its components can be easily replaced in case of faults. As anticipated, NCGs are a key component of our network hierarchy. They are small computers of size 15 cm $\times$ 15 cm, bearing limited power supply requirements, which can be supported through the Power-over-Ethernet (PoE) standard. The nodes have been deployed in the buildings of the Department of Information Engineering at the University of Padova, Italy. Part of the testbed is shown in the map in Figure 2.5, and counts more than 350 nodes

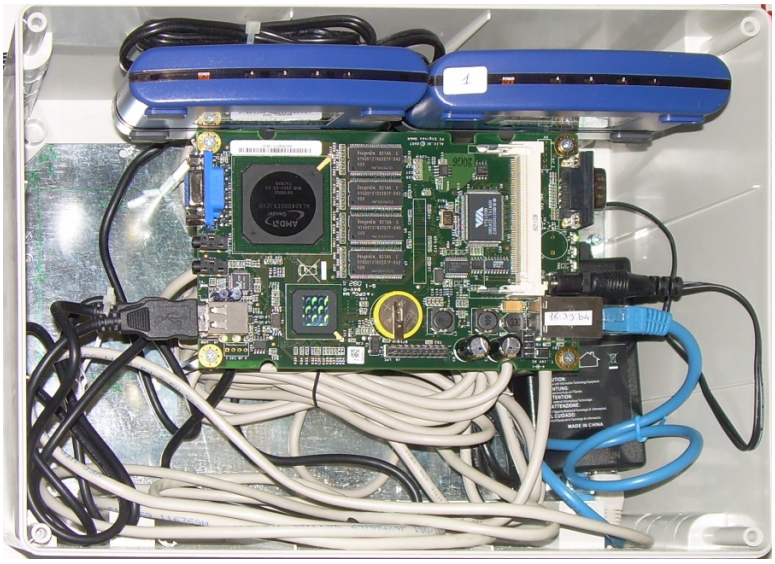

Figure 2.7: An example of Node Cluster Gateway (NCG), showing the embedded computer at the center, and the two USB 2.0-compliant hubs at the top.

currently deployed.

In the following we give a brief description of the components of the testbed following a bottom-up approach.

2.1.2.1 Sensor nodes: TelosB

The wireless embedded sensors we chose for use in our testbed are the TelosB nodes [34] (Figure 2.8), a widely used platform enjoying wide community support and usually used as a reference in performance evalutations. Here we recall that TelosB platforms are low-power IEEE 802.15.4 wireless nodes communicating with a maximum transmission power of 1 mW within the 2400-2480 MHz bandwidth, at a maximum bit rate of 250 kbps. This choice has been made since this platform is undergoing widespread use around the world, and therefore it is constantly supported and upgraded. TelosB nodes come equipped with temperature, humidity and light sensors. They can be directly connected to other devices through an embedded USB port.

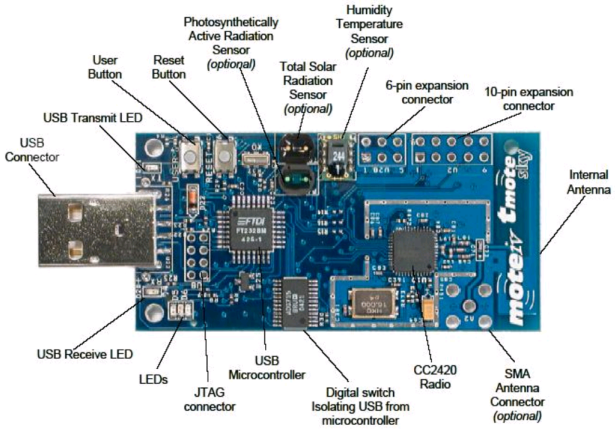

Figure 2.8: The TelosB sensor board showing a breakdown of the main components.

TelosB node – TmoteSky: summary of features

- Cpu MSP430 Texas Instruments 8 MHz, RAM 10 KB, ROM 48 KB.
- ZigBee: 250 Kbps, standard 2.4GHz IEEE 802.15.4.
- Temperature, humidity and brightness sensors.
- USB installation.

2.1.2.2 USB hub

Sensor nodes are connected to the gateway through USB self-powered hubs. As anticipated before in this report, the hubs serve many purposes: primarily, they supply power to the nodes, and replicate USB ports to extend the connectivity capabilities of NCGs. Moreover, self-powered hubs allow to toggle power supply on a per-port basis, allowing, e.g., to switch on or off any sensor connected to a specific hub port via a straightforward software command. While the latter

individual port power control support is a standard requirement for every self-powered USB 2.0 hub, most off-the-shelf products do not adhere to full USB 2.0 specifications, and therefore implement only a subset of features that are expected to be most widely used. However, the capability to switching power on or off for single USB ports is a critical requirement for a fully functional and reliable testbed, because it enables hard node resets via remote controls. For these reasons, a number of USB hubs have been carefully inspected and tested: only certified USB 2.0 hubs with reliable performance have been chosen for use in the testbed. Here we summarize the features of the Linksys Proconnect 4-port USB 2.0 hub, that is representative of the equipment we use in our testbed.

Linksys Proconnect 4-Port USB 2.0 Hub: summary of features

- Supported standards: OHCI, UHCI, USB1.1 and 2.0
- 1 USB Type B Root Port
- 4 USB Type A Device Ports
- LEDs signaling power and activity
- Individual port power control and overcurrent compensation

2.1.2.3 Node Cluster Gateways (NCG)

The main part of our hierarchical structure is made up by NCGs, that connect sensors via hubs and communicate with the rest of the network through an Ethernet backbone. The NCGs are embedded PCs with reduced capacity computation, that run a GNU/Linux kernel augmented with all libraries and tools required to control the nodes (mainly, TinyOS-2.x and related add-ons) and a Java Virtual Machine. At this phase of the project, we have not forced ourselves to strictly define the tasks that should be carried out by the NCGs and to the centralized network server: this decision will be made at a later stage, after a careful assessment of the capabilities and requirements of the testbed. A more detailed description of the NCG hardware follows.

Figure 2.9: Outdoor WISE-WAI Testbed Deployment

Alix 3c2 - Fanless embedded AMD motherboard: summary of features

- CPU: AMD Geode LX800 500 Mhz
- SDRAM: 256 MB on board
- 2 USB Hosts
- 1 RS232 COM port
- 1 Rj-45 Ethernet LAN port
- 1 expansion slot for Compact flash memory cards

2.2 TinyNET [2, 3]

Wireless Sensor Networks (WSNs) have emerged as a promising paradigm for a number of smart applications to be implemented in the near future. These applications are growing beyond simple data collection, localization and information retrieval services, to incorporate increasingly complex features such as, e.g., smart sensing, assisted navigation, and sensory extension. It can be foreseen that many solutions by different distributors will undergo full-fledged development and find their way to the market, e.g., see [35].

From a developer's point of view, it is very convenient to create new software based on the reuse of as many program components as possible, taken from both open-source and proprietary repositories. However, as noted in [36], very few applications are actually built based on reusable components: in fact, the most widespread approach is to implement ad hoc, monolithic blocks that deliver the required services. From these macro-blocks, it becomes difficult to distinguish which components provide a given set of functionalities. While this usually bears greater efficiency at execution time and a slightly more contained memory footprint, a software block-based approach can achieve comparable efficiency and footprint while bearing the further advantage of greater modularity and broader system-level view [36].

A first step toward the resolution of these problems is moved by introducing TinyNET, a modular framework for TinyOS that *i*) makes it easier to build applications by reusing software modules; *ii*) provides any protocol and application with a layered network interface that encompasses the whole stack, while still allowing cross-layer operations and exchange of parameters; *iii*) allows fast reconfiguration of applications through new protocols and functionalities, that transparently become a part of the layered network stack. Our framework operates on top of TinyOS but below the user application modules, and is completely transparent (in the sense that TinyOS module binding directives are intercepted and used to place any module within the framework).

TinyOS is a powerful platform to build applications for WSNs, due to its limited memory consumption and to its cross-platform support; its design is based upon tiny components, whose interfaces are linked using a highly optimized C

dialect (nesC [37]). This paradigm has proven to be effective when building a system with shared, highly reusable components, and helps reduce the final binary image size.

The communication abstraction employed in TinyOS is the Active Message (AM) model [38]. The AM header is composed of 1 byte, the AM type, identifying the user-level message handler. The rest of the packet is composed of the payload to be passed on to the handling process. The AM paradigm allows to share the radio interface, by binding applications to a single AM type of the Active Message subsystem. Applications employ available interfaces to control the radio subsystem, e.g., to power it on/off and, by means of platform specific commands, read link quality indicators (TX power, RSSI, LQI). Directly putting an application in control of the radio subsystem is a valid approach only if the application itself is very simple; in case a more complex system should be built, a top-down approach is preferred, which requires to design the architecture and modules of the system before developing the system itself. However, network applications are usually designed as holistic modules which are tightly integrated into TinyOS itself, bearing hardly reusable parts and usually incorporating platform-specific code. This is also due to the intrisic structure of TinyOS, whose architecture do leave an extensive freedom in the design of components, which unfortunately results in frequent design of components as monolithic and platform-specific. Furthermore, there is no logical network architecture available for TinyOS, which represents a drawback to open contributions of network protocols and applications. TinyNET is a network framework designed to help fill the gap between TinyOS itself and any kind of networked system built on top of it.

2.2.1 Related Work

With a few exceptions, most architectures proposed for WSNs are created to comply with a particular requirement, or to support a specific protocol feature. For example, the authors in [39] face the challenges of energy management in WSNs by treating energy as a fundamental design primitive. Their architecture is composed of three parts, namely a user interface for specifying an energy policy, a monitoring system to control energy usage, and a management module to enforce

the energy policy. The use of expressive language to specify the energy policy enables easier user interaction.

The Tenet architecture [40] has been specifically designed to support tiered architectures, where slave (low-tier) nodes are only in charge of gathering information, whereas the complexity of system-level, computationally-intensive tasks (such as data fusion) is concentrated on high-tier master nodes, which usually own a non-volatile power supply. It is worth noting that this is in line with the Router/End Node paradigm seen, e.g., in the ZigBee standard [41]. Tenet subdivides sensing tasks into tasklets, each of which specifies the sensing operation to be carried out by low-tier motes, as coordinated by masters. Tasks are flooded to all motes upon user input.

The SP architecture proposed in [42] aims at providing a link layer abstraction to all protocols, by means of a shared message pool (formed of data to be transmitted in packets) and a shared neighbor table, which holds a summary of neighbor information which is made available to all protocols, instead of having each protocol maintain its own. The SP approach allows to bind the standard interfaces of the higher layers of the protocol stack to the link layer: the effectiveness of this approach is explained in [43]. Chameleon [44] also targets the design of a reconfigurable architecture, that allows applications to transparently adapt to different MAC, routing and transport protocols. The key feature of Chameleon is a universal header format which is based on packet attributes rather than bit fields.

The approach chosen in [36] is slightly different: the authors propose a MAC Layer Architecture (MLA), which aims at subdividing usual MAC-layer functionalities into atomic operations, so that existing as well as new protocols can be programmed based on a large library of reusable components. Each component (either hardware-dependent or -independent) is instantiated into TinyOS: when properly connected, these software blocks allow the creation of MAC protocols that are entirely analogous to those found in the literature, and yield the same performance (e.g., throughput) while bearing only a slightly larger memory footprint. An approach similar to [36] is also followed in [45], where the authors propose COPRA, a communication processing architecture based on protocol processing stages and engines, i.e., components that perform basic operations

and can recursively become part of larger structures to carry out more complex tasks. A survey of other ongoing projects regarding networking abstractions in TinyOS as of a few years ago can be found in [46].

SensorStack [47] is a solution to provide an abstraction of communication services to the upper layers, in order to facilitate data-centric communication. It relies on an information broker based on the publish-subscribe paradigm, and aims at providing simple interfaces and efficient use of memory to share cross-layer parameters, as well as the support for notifying complex events to related protocols. Similarly, Cross-Layer Optimization Interface (CLOI) [48] provides an interface to exchange data between protocols; this interface is also implemented in the form of data structures such as message pools and neighbor tables.

Unlike the previously cited approaches, our TinyNET architecture works at a lower level. We focused more on the reusability of any software block, rather than of specializing the architecture to support a certain network task or application. In this light, the most similar approaches are [36, 45]: however, there are also some differences, in that [36] focuses on MAC protocols, while [45] requires specific protocol engines and stages to encompass network functionalities. In our case, instead, the framework's main task is to let the user promptly instantiate, switch and connect any kind of open or proprietary TinyOS-based software, with special regard to creating multiple instances of the same components and to transparently multiplexing protocols over the same interface. In this regard, it is worth noting that most of the previous architectures, e.g., [39, 40] could be integrated seamlessly as part of the TinyNET framework. This also applies to the MAC components of [36]. Finally, it is worth noting that the Contiki operating system [49] also implements an adaptive networking architecture for WSNs through the Chameleon/Rime stack [44]. However, as most applications developed to date have been programmed in TinyOS, TinyNET presents considerable advantages, as it yields equivalently solid network architecture, modularity and extensibility to present and future TinyOS applications. Also, while Contiki has a fixed ROM occupancy of 40 kB, TinyNET and TinyOS present a much smaller footprint, see Section 2.3.

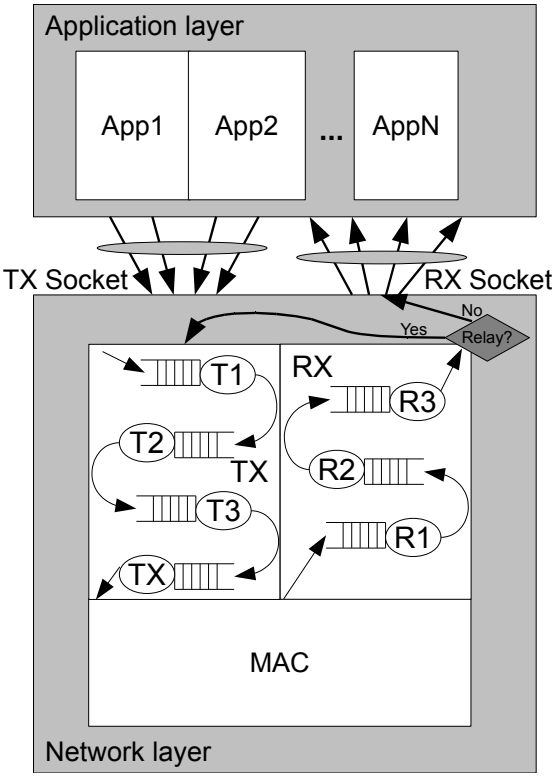

Figure 2.10: A sketch of the TinyNET architecture.

2.2.2 Architecture

TinyNET exploits nesC to split any networked system into two parts: the application layer and the network layer. The *application layer* is similar to TinyOS's standard developing entry point, with the additional feature that every application module represents a single, independent process in the network system. Utility interfaces have been built to perform such operations as radio state control and channel selection; in addition, the control of the radio subsystem has been centralized, thus providing the ability to intercept any control request. In turn, this common entry point facilitates the development of resolution techniques for concurrencies in radio control (e.g., through independent locks to be imposed on the radio by those applications that require exclusive access to this resource). The *network layer* is instead a novel layer which contains any modules that require full access to every packet received and/or transmitted by the node. The network layer is a direct development entry point for new network protocols to be inserted

in the framework, transparently to applications. This layer also supports ordered access of protocols to transmitted/received packets, and provides full control over the packet itself, e.g., any module can change the field structure of the packets if required. To ease the differentiation among the application and network layer, they are programmed inside separate files, allowing system integrators to easily combine application and network components.

The development of TinyNET has posed various design challenges, mainly in order to select a minimal yet sufficient set of inter-component interfaces, which has to be clean and practical for applications, yet powerful for network modules. Moreover, significant attention has been paid to preserving support for cross-layer interactions, in that any network module can access and process information contained in other modules (this feature is natively available in TinyOS). A hardware abstraction layer has been introduced to access specific chip features, in order to provide cross-platform support and access to low-level hardware components. Moreover, the development has been carried out on top of TinyOS in a completely independent fashion, in order to favor the porting of TinyNET to newer TinyOS versions. (Even the TinyNET code is placed in a different folder tree with respect to the rest of TinyOS.) As shown in Figure 4.1, the application packets to be transmitted are taken in charge by the network layer and are scheduled across network modules.[1] After passing through all linked network modules, the packets are sent to the MAC module to be scheduled for transmission.[2] The reception of packets takes place following the reverse order, i.e., when the MAC module signals the reception of a new packet, this packet is processed by all RX network modules (in reverse order with respect to the transmission phase, see Figure 2.10) and is eventually passed on to the application corresponding to the AM type of the received message.

2.2.3 Inter-component interfaces

Four different kinds of interfaces are required in TinyNET.

[1] In the current implementation, the number of network modules has been fixed to three for simplicity.

[2] The MAC *module* manages channel access independently of actual MAC/routing *protocols*. It is used to implement, e.g., ALOHA vs. CSMA.

Application layer interfaces—Transmission and reception interfaces are required by applications to access the framework. In general, these are an expansion of current TinyOS `AMSend` and `Receive` interfaces.

```
interface TX {
 command message_t* send(
   message_t* pkt,
   am_addr_t dst,
   uint8_t len,
   uint8_t power,
   uint8_t prio,
   bool swap );
 event void sendDone(
   message_t* pkt,
   error_t status );
}
```

The `send` command has been extended over the standard `AMSend.send`, with new per-packet TX power and scheduling priority attributes. Moreover a bool `swap` parameter is passed, to request a `message_t*` pointer to a free buffer in exchange to the `message_t` buffer passed for transmission. This can improve memory utilization in nodes with multiple applications that concurrently access the network subsystem. At the current stage, the `Receive` interface is identical to TinyOS 2.1's, and has been renamed `RX`, in order to support later modifications.
Network layer interfaces—Network layer modules require full access to the packet, and can thus access the internal framework data structures used by scheduling components. The internal transmission buffer is defined as follows:

```
typedef struct txring_buffentry_t {
  am_addr_t src;
  message_t* pkt;
  txpkt_state_t state;
  uint8_t power;
  uint8_t prio;
  error_t error;
  uint8_t swapped;
```

```
} txring_buffentry_t;
```

where `pkt` is a pointer to the `message_t` holding the actual packet to be transmitted: swapping this pointer with a different one allows the network module to rewrite the entire packet from scratch. The variable `state` holds the current scheduling state of the packet, representing which network modules it has already stepped through; `power` is the power level at which the packet should be transmitted, `prio` is the scheduling priority, `error` stores a general purpose error code and `swapped` tells if the buffer space of the current packet has been swapped. Moreover, the source address of the packet is also stored in the structure (`src` field), to distinguish between the originator of the packet and the current relay (which is set by TinyOS). The full `txring_buffentry_t*` structure pointer is passed to every transmit network module which implements the `ProcessTXPacket` interface:

```
interface ProcessTXPacket {
 command error_t process(
   txring_buffentry_t* txbuf );
 event void processed(
   txring_buffentry_t* txbuf,
   error_t error );
}
```

Network modules handle one packet at a time: they receive the input packet through the `process` command, and signal back the processing completion using the `processed` event.

```
typedef struct rxring_buffentry {
  message_t* pkt;
  rxpkt_state_t state;
  error_t error;
  uint8_t opt;
} rxring_buffentry_t;
```

Analogously, an RX buffer structure is defined, storing the `pkt message_t*` pointer, the `state` variable of the current processing step, and an `error` variable holding any error codes faced during packet processing. Additionally a persistent

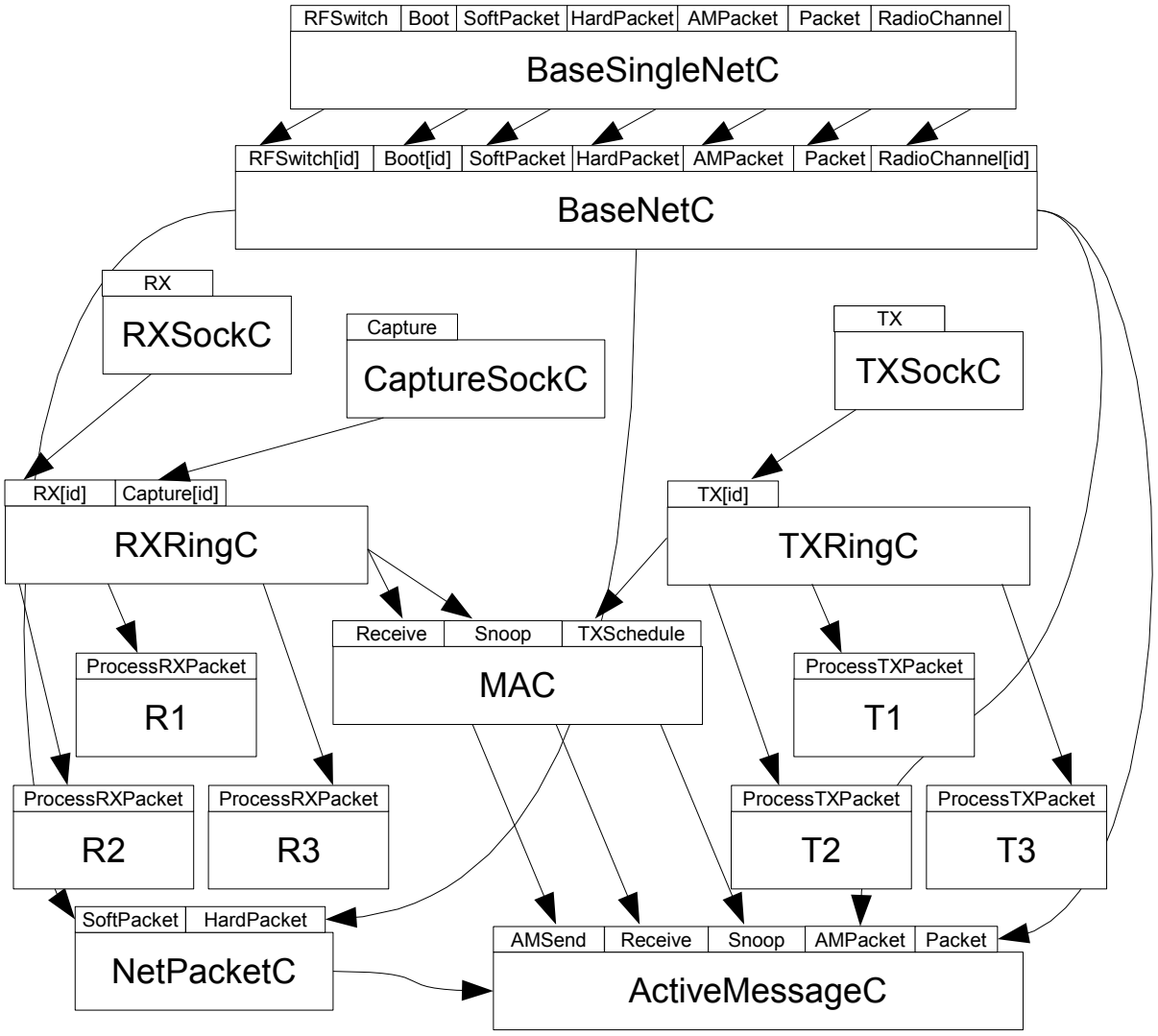

Figure 2.11: Scheme of the wiring of TinyNET components.

per-packet `opt` variable is provided for internal module use. A `ProcessRXPacket` interface is provided, analogously to the aforementioned `ProcessTXPacket`.

Apart from the described TX/RX processing interfaces, two more specific interfaces are required to build a practical network layer: a `Route` interface and a `TXSchedule` interface.

```
interface Route {
 command bool forward(rxring_buffentry_t* rxbuf);
 command bool isForMe(rxring_buffentry_t* rxbuf);
}
```

The `Route` interface is required to handle the delivery process of received packets. More specifically, it allows a routing module to implement custom logic to choose if a packet should be further relayed over a multihop path (returning the `forward` command). Moreover, the `isForMe` command can be implemented to decide whether a packet must be delivered to the applications running on the local node. The `TXSchedule` interface, instead, requests a packet transmission slot to the MAC module.[1]

```
interface TXSchedule {
  command error_t schedule(
    uint8_t id,
    uint16_t dst );
  event txring_buffentry_t* doTX(
    uint8_t id );
  event void TXdone(
    txring_buffentry_t* txbuf,
    error_t error );
}
```

Using this interface, the MAC module can be asked to reserve a slot for transmission of packet `id` to node `dst`. When the transmission can eventually take place, the MAC module fires a `doTX` event, which will return the pointer to the TX buffer to be transmitted. Upon transmission end, a `TXdone` event is fired to return the result of the operation.

[1] This is required to support reservation-based slotted access protocols: unslotted protocols may allow access right away or according to specific rules.

Hardware abstraction interfaces—At the current stage of development only CC2420-based motes are supported, and the supplied hardware abstraction is bound to the CC2420 TinyOS implementation. When more radio chips are supported, the interface definitions and conventions will be refined. The first interface required to abstract from hardware-specific components is `HardPacket`:

```
interface HardPacket {
  command uint8_t getPower( message_t* p_msg );
  command void setPower(
    message_t* p_msg,
    uint8_t power );
  command int8_t getRssi( message_t* p_msg );
  command uint8_t getLqi( message_t* p_msg );
}
```

The per-packet TX power level, receive RSSI or LQI are extracted from the radio subsystem using this interface. As reported before, the returned values are currently interpreted as in the `CC2420Packet` module:

```
interface RadioChannel {
  command error_t set( uint8_t channel );
  event void setDone(
    uint8_t channel,
    error_t error );
  command uint8_t get();
}
```

The `RadioChannel.set` command allows to set the operating radio channel of the RF transceiver, according to the IEEE 802.15.4 standard; upon completion of the command, a `setDone` event is propagated. The channel currently in use can be identified by using the `get` command.

Legacy application layer interfaces— To facilitate the migration to TinyNET, a set of standard TinyOS network interfaces is provided: `AMSend`, `Receive`, `Packet` and `AMPacket`. These interfaces are sufficient to translate former TinyOS applications to TinyNET, by instantiating TinyNET components instead of standard TinyOS components.

2.2.4 Technical description

The path tree of TinyNET contains the following folders: `sys` (framework core modules); `interfaces` (interface definitions); `modules` (actual implementation of network, MAC, and application modules); `platforms` (collection of platform-specific components); `lib` (reusable components, useful to implement common network modules); `6lowpan` (porting of TinyOS's 6LoWPAN implementation to TinyNET); `examples` (sample usage files demonstrating TinyNET); `install` (installation procedures and utility files).

The `sys` directory contains all the components implementing the actual framework. As shown in the wiring scheme reported in Figure 2.11, the `BaseSingleNetC` component is the basic module every application should instantiate to signal its own presence as part of the framework; the instantiation allows every radio-related event (power on, channel change, radio subsystem boot) to be exposed through the offered interfaces. `BaseSingleNetC` actually instantiates `BaseNetC`, and binds it to the application with a unique `net_app_id`; the `BaseNetC` component is the network layer definition file, which is in charge of wiring all network layer components, of loading `RXRingC`, `TXRingC` and `ActiveMessageC`, and of selecting and wiring `MacC` with the three receive and transmit modules, `R{1,2,3}` and `T{1,2,3}`, respectively.

The `RXRingC` component is in charge of passing on every packet received by `MacC` to all receive network modules, and ultimately of delivering the packet to the application to re-queue it for transmission. Similarly, the `TXRingC` component is in charge of handling transmit packets to every TX network module. Furthermore, it reserves a transmission slot from the MAC module, handles the packet to that module when the slot is available, and signals back to the application when the packet has actually been transmitted. The `MacC` component has full direct access to the TinyOS radio subsystem and is in charge of every transmission and reception.

2.3 TinyNET Proof-of-Concept running over the WISE-WAI testbed

Our first experience with the framework focused on simple tests to measure overhead and basic functionalities. The `BlinkToRadio` application has been ported to TinyNET as a reusable application module using native `TX`/`RX` interfaces. Hence, `BlinkToRadio` can be loaded by simply instantiating and wiring the component in the application layer definition file. When the firmware is built using the described application and no network modules, the overhead due to the framework size can be measured in comparison to a plain `BlinkToRadio` binary. As to ROM occupancy, the use of TinyNET increased the `BlinkToRadio` size by 3.5 kB, reaching a total size of 15 kB. However, it should be noted that this overhead is fixed, and depends neither on which nor on how many applications are loaded, and is also independent of how many network modules have been wired to the framework. The RAM occupancy overhead depends on scheduling queue buffer sizes as set up in configuration files, plus about 60 B of static variables allocated by the framework.

After testing TinyNET's memory footprint, we wish to experience the practical advantages yielded by usage of TinyNET, as compared to the standard TinyOS programming approach. To this end, we have built a more complex system, featuring several application, networking and communication modules. A multi-hop environmental monitoring and querying system using 6LoWPAN has been chosen in this regard, as it is complex enough to prove the advantages brought about by using the TinyNET framework. We highlight that the focus of the work was to prove that TinyNET allows easy and straightforward implementation of these modules, compared to TinyOS, rather than on collecting performance metrics related to the system itself.

The following sections will present the components of our system architecture in more detail.

2.3.1 IPv6/6LoWPAN stack

Research on the integration between standard Internet services and WSNs, as well as the introduction of novel concepts such as the Internet of Things bestow larger

importance on protocols that allow easy connection between WSN islands and Internet. 6LoWPAN plays a key role in this regard, as it is specificlly designed to make any (even tiny) object addressable from anywhere on the Internet through IPv6; the burden of typical IPv6 processing and header sizes, which are not optimized for the wireless channel, is alleviated by compressing the IPv6 header; this is achieved, e.g., by avoiding repeating patterns and useless or redundant fields, while still allowing use of the full IPv6 address space breadth. Such simple protocols, that yet make packet communications compatible to Internet protocols shifts the perspective of WSNs, which are expected to become part of the world-wide network along with any other kind of connected smart object.

TinyOS features a lightweight 6LoWPAN implementation, which can be found in the path `lib/net/6lowpan`; this implementation provides the minimum working set of features required by IPv6 specifications: ICMPv6 Echo Request/Reply, header compression and UDP socket support. The implemented features and exposed interfaces fit the requirements of our proof-of-concept application: for this reason, the 6LoWPAN implementation provided with TinyOS has been ported to TinyNET. Through the 6LoWPAN component, any node can assign itself any IPv6 64-bit prefix to be added to its MAC address (`TOS_NODE_ID`); additionally, a node can send UDP packets to any IPv6 address and listen over one or more UDP ports.

In TinyNET, 6LoWPAN sits on top of the whole framework, behaving as a standard application. This way, 6LoWPAN can make straightforward use of any available link layer and network protocol (e.g., routing and security). By using legacy TinyOS support interfaces, porting 6LoWPAN to TinyNET has been very easy, as the only changes required involved the instantiation of some components and the setup of the proper wiring to the 6LoWPAN subsystem. This is a further clue of how TinyNET straightens up development work when integrating objects into a more complex application.

2.3.2 Routing network module

A simple routing protocol based on hop count (HC) descent has been implemented: basically, a node with HC equal to n always relays packets to a neighbor exhibiting HC equal to $n - 1$. While this might be a suboptimal strategy [50], it

is sufficiently effective and simple to serve as a proof-of-concept component. The HC information has to be renewed periodically: to this end, each node sets its own HC to an arbitrarily high value, and the sink starts a HC flooding procedure by sending an advertising packet with HC equal to 0; all nodes receiving the messages set their HC equal to 1, and choose the sink as their next hop. The procedure is recursively repeated, as every node broadcasts its hop count (say n), and its neighbors set their own HC to the minimum between the current HC and the value read from the packet plus one. When the node's HC is actually updated (the packet carried a smaller value than currently held by the node), the receiver selects the packet sender as its next hop toward the sink. This is only one way to choose the next hop: other choices that lead, e.g., to some cost optimization [50] can be applied as well. In order to handle dead nodes and topology modifications, an `age` variable is associated to any chosen relay. Each time a node propagates its HC, it also increments the `age` of its relay by one. When `age` gets bigger than a preset `MAX_AGE`, the current next hop becomes outdated, and the node is required to perform a further relay choice upon reception of a HC update packet by a neighbor; in any case, the `age` of a relay is also set to zero any time a HC packet is received by that relay.

The protocol described above supports node-to-sink communication, but does not apply to sink-to-node routing, because the sink itself has no knowledge about which path to go through in order to reach the node. A simple solution to this shortcoming is to have any node, including the sink, remember which neighbor is relaying the packet sent from a specific source. By dynamically building a {relay,source} least recently used (LRU) cache table, any path can be walked in a reverse, sink-to-node direction. To accomplish the described tasks, a routing header is required, which carries information about the final destination of the packet, the chosen next hop, and the original source of the packet (which is also required in order to to build the route from sink to node).

Implementing the described protocol in TinyNET requires that the network module provides three interfaces: `ProcessTXPacket`, used to build the routing header in the packets queued for transmission (in order to keep implementation simple, the routing metadata has been appended to the outgoing packet); `ProcessRXPacket`, required to extract the routing footer appended by the transmitting node; `Route`, which updates the LRU table when a packet is queued for

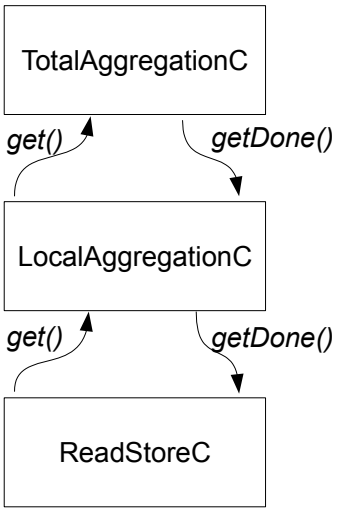

Figure 2.12: Interconnection between modules of the application performing sensor reading collection and aggregation.

further relaying or delivery to the application.

2.3.3 Environmental monitoring and querying application

The application built upon the described system performs environmental monitoring and supports single node querying. The application concept is very simple: the monitoring component `ReadStoreC` periodically samples values provided by on-board sensors, and stores them into the RAM using a circular buffer of fixed size equal to `N_PKT`. The sampling interval is fixed to `READ_INTERVAL` (and can be tuned by acting on the variable).

Asynchronous to the gatherer of sensor readings samples, a second component (`LocalAggregationC`) accesses the circular buffer, and stores a synthetic representation of the values on the flash memory integrated on the sensor node board. This way, the only permanent trace of past readings is kept in a compressed form, and provides useful information about reading history (which can be accessed using proper queries), without wasting the flash memory by, e.g., storing all samples. As a compressed representation of the readings, we chose their average value.

The networked component (`TotalAggregationC`) is the only module in charge of network-related operations, such as listening for incoming requests and report-

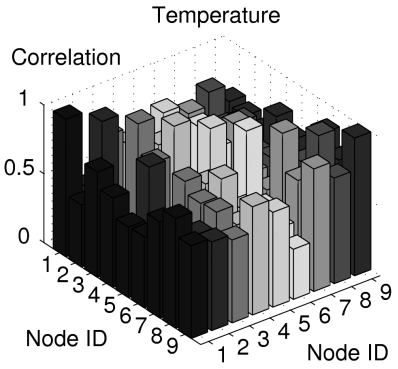

Figure 2.13: Correlation among the time series of temperature readings for nodes m1 through m9 in Figure 2.5.

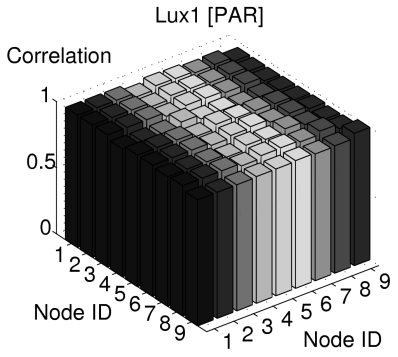

Figure 2.14: Correlation among the time series of luminosity readings for nodes m1 through m9 in Figure 2.5.

ing data back to the sink. The latter operation is performed periodically by all sensors, but it should be noted that the sink itself can query any specific sensor at any time, if needed. The 6LoWPAN support discussed before also enables queries to be originated by any host on the Internet toward any node in the network; the converse is also true, i.e., sensor readings can be conveyed to any host on the Internet.

In order to achieve communication efficiency and scalability, the nodes progressively aggregate the sensor readings while routing packets throughout the network. As commonly done in many similar approaches , we have organized the nodes into an aggregation tree. The tree is formed in such a way that hierarchically higher nodes aggregate the readings received by the sensors from lower hierarchy levels. Hierarchical connections are devised so that nodes expected to yield correlated measurements are at the same hierarchy level and report to the same head node. Such nodes are said to form ma *group*. For example the nodes in the same room report to a node which is also inside the room, and which can decide if and how to aggregate the data coming from its siblings. In turn, further levels of aggregation are possible; for example head nodes representing groups of sensors within rooms on the same floor or wing of a building may report to another head node, which occupies a higher hierarchy position; this node receives information on a per-group basis, and can thus decide whether or not to fur-

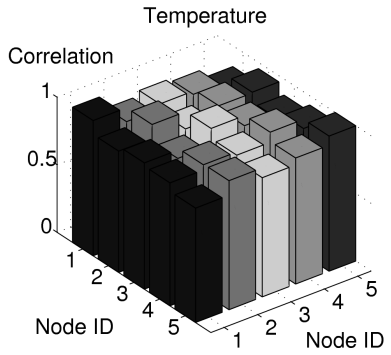

Figure 2.15: Correlation among the time series of temperature readings for nodes c1 through c5 in Figure 2.5.

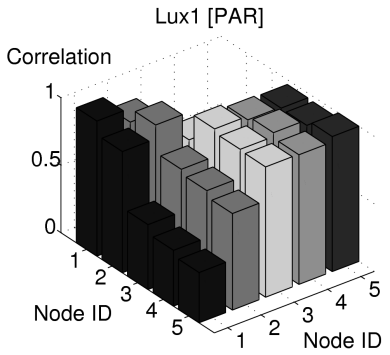

Figure 2.16: Correlation among the time series of luminosity readings for nodes c1 through c5 in Figure 2.5.

ther aggregate the readings, depending on whether the end user requires a coarse or fine reading. We note that this structure is scalable and fast to replicate. All aggregation operations applied to group readings are again demanded to the `TotalAggregationC` component.

The aforementioned application elements are connected through a custom interface `ReadData`, supporting asynchronous replies to `get` commands by means of `getDone` events. Therefore, upper layer modules propagate queries in a top-to-bottom direction whenever data is required, thereby limiting the further occupation of the sensor RAM. We recall that the average of past readings are available on the flash memory of the node, whereas a limited amount of recent, non-averaged readings can be retrieved from the node circular buffer.

Examples of the data retrieved by the application can be found in Figures 2.13 through 2.16. In particular, Figures 2.13 and 2.14 respectively show the correlation among the time series of temperature and luminosity readings output by sensors m1 through m9, in the bottom-right room of the map in Figure 2.5. Figures 2.15 and 2.16, instead show the came correlation metrics taken over the readings of sensors c1 through c5, along the corridor. From the graphs, we infer that the correlation among the luminosity levels perceived by nodes in the room (Figure 2.13) is very high, meaning that this metric can be aggregated and represented with an average among readings with little if any loss of information. The same applies to the temperature readings of nodes in the corridor (Figure 2.15).

The reason behind the very high correlation is that the room features very uniform lighting from windows and sometimes ceiling lights, which leads to very similar sensor readings. The same is verified for the corridor temperatures, which tend to be uniform across the corridor itself, with slightly larger deviations, which are small enough to make the temperature information amenable to be represented with an average value. The luminosity in the corridor (Figure 2.16) behaves differently: in this case, there are no windows, and lighting comes from rooms, window-doors, and side corridors as well, making the luminosity non-uniform, and more so for nodes placed farther from each other (e.g., node c1 and c5, which have the lowest luminosity correlation. The same argument applies to the temperature readings in the room (Figure 2.13), where this time the source of different heat levels is the equipment placed in the room itself.

While these results are just samples of which data can be gathered from our sensor network, they indeed suggest that aggregation is a viable and effective option for environmental data, making the environmental monitoring application an effective part of our proof-of-concept TinyNET application.

3

Prototyping the Internet of Things

The gravitational core formed by the concepts of Web 2.0 and the Internet of Things (IoT) is shifting Web applications and services concepts towards wider integration and accessibility, in light of an anytime, anywhere, anything internetworking paradigm. The future Internet builds on these bricks to make up a dynamic entity, yielding novel means of interaction with services, other users, and the environment. Wireless Sensor Networks (WSNs) have been recognized as a very important block of this internetworking concept. Tiny, distributed objects as they are, WSNs constitute a reasonably cheap sensory extension to Internet-connected devices; moreover, their computational capabilities allow for further (though possibly limited) use flexibility and functional expansion. Any kind of next-generation Internet-enabled portable device will set up advanced interactions with the "things" making up the new IoT, resulting in a pervasive infrastructure of fixed and mobile heterogeneous nodes, seamlessly providing, exploiting or sharing context-based services and applications. In particular, capturing the context and surroundings of devices will constitute a key component, making such operations as "Googling" physical reality possible and common [51]. Such a wide perspective requires stable foundations, starting from a widely agreed upon protocol and communication infrastructure, which has currently been identified in the IPv6/6LoWPAN protocol suite [52]. By integrating any object into the IP infrastructure, 6LoWPAN is an important enabling technology allowing

to merge newer and older Web services, as well as to support the cited IoT interaction paradigm, while still running everything over the widespread Internet infrastructure.

As part of the SENSEI [53] consortium and in the context of the WISE-WAI project [21], at the University of Padova we early put this vision into practice, by channeling experience in the field of wireless sensor networking towards the realization of a scalable and easily extendable network structure, which is basically formed of three classes of nodes (base stations, mobile nodes and specialized nodes) running compatible code but providing different functions and carrying out different tasks.

In this chapter, our work in the design of a IoT system based upon sensor nodes will be presented.

Nodes in our WISE-WAI testbed (see Chapter 2) have been flashed with an IPv6-compatible stack, which makes them directly addressable from any Internet-capable device. The nodes have been programmed to support diverse services, from environmental parameter monitoring to localization, and are in turn supported by lower-level functionalities allowing, e.g., to switch the application being run on the node, to change the class/role of a node, to spread software changes and updates over the air, or to perform low-level resets in case of malfunctions [1].

Offering such services through our network provided us with an opportunity to realize part of the IoT vision and focus research efforts in the field: in addition, it demonstrates the advantages of the IoT in the management as well as everyday occurrences of University life, as explained in Sec. 3.2.1. In fact, thanks to our early experience, we built our vision for a full-featured, doable and interoperable IoT protocol stack.

3.1 Related work

The interconnection of WSNs to the Internet has been widely researched in the last few years. At the beginning of the WSN era, researchers focused on the development of dedicated systems, where highly specialized but non-standard protocols were used within the sensor network, whereas one or multiple gateways were used to translate messages and ultimately connect the WSN to the external IP world. While these systems were generally efficient in the specific application scenario

they were designed for, they lacked flexibility: developing new applications on top of them was therefore time-consuming and cumbersome, as it required modifications to the specialized protocols within the WSN. As a remedy to that, the 6LoWPAN standard has recently been proposed as a viable method to bring IPv6 to WSNs [54, 55] so that sensor nodes can be natively addressed and connected through the IP protocol. This has obvious advantages such as rapid connectivity and compatibility with pre-existing architectures, plug-and-play installation of WSNs, rapid development of applications as well as the possibility of integrating with existing Web services developed for standard IP networks.

Web services are extensively and successfully used mechanisms in Information Technology (IT) systems; they may be defined as techniques to develop interoperable and distributed applications exploiting Web standards like HTTP. As discussed in [56], sensor networks can greatly benefit from their usage, as Web services allow the integration of WSNs into any system that is built on standard IT components, e.g., industry/home automation as well as home energy management systems. TinyREST [56] efficiently implements Web services on WSNs by carefully minimizing the overhead introduced by the transport layer whilst using data formats such as XML and WSDL.

The authors of [57] have recently demonstrated that SOAP-based Web services are doable for WSNs. The RESTful approach is however currently preferred for these networks due to its lightweight character [58]. RESTful has the sensor resource as its main abstraction and every resource in every sensor node is linked and retrieved through a Uniform Resource Locator (URL). In addition, standard HTTP methods such as GET, POST, PUT and DELETE are used respectively to acquire, modify or delete the value of a given resource.

Recent research efforts explore the feasibility and the performance limits of the RESTful approach for Web services on top of 6LoWPAN for WSNs. These studies aim at improving the usability of WSNs, making them suitable for complex installations, while retaining the flexibility of IP-based networks. A recent paper [58] presents an IP-based WSN where nodes send data using Web services. We show that this approach is doable for resource constrained sensor nodes in terms of acquisition time for the sensor data and power consumption. In addition, they prove that TCP can be supported under particular network settings. A similar approach has been presented in [59] where WSN resources are integrated

into IP-based networks exploiting Web services.

In this chapter, the seminal work done on designing a web service architecture for the IoT will be presented. Resources are handled according to the RESTful approach and binary encoded XML is used to reduce the transmission overhead. In addition, we exploit standard interfaces for access (Resource Access Interface, RAI) and publication (Resource Publication Interface, RPI) of resources. The peculiarity of our work is that of presenting an actual system, using standard protocols to offer various WSN services to both the administrative staff and regular users of the University (i.e., students and professors). The resulting architecture will be able to provide network services to a specific, custom-designed base-station, as well as to generic mobile nodes accessing the network using standard protocols.

3.2 Scenarios and Use Cases

While aiming at providing a network stack for IoT devices, a constant comparison with real-life application scenarios was required. In fact, we used a number of realistic target use cases as a reference that guided us in the definition of the characteristics that are required for the protocol stack.

Along these lines, those application scenarios will be discussed together with their specific characteristics.

3.2.1 University [4]

A University offers many application scenarios for demonstrating the advantages brought about by the IoT in real life. In particular, at least three service categories can be offered in a University facility: *i*) *Office automation*: the services belonging to this category are automated applications, meant to help, e.g., the management staff; *ii*) *Teaching*: this category includes all functionalities that can be exploited by full registered users only; *iii*) *Guest*: visitors can access this class of services to retrieve basic information (e.g., to navigate around buildings) or to locate people.

These services involve quite diverse technology, security and quality requirements. For instance, the first category includes such services as door access control, which aims at granting access to qualified persons and therefore needs

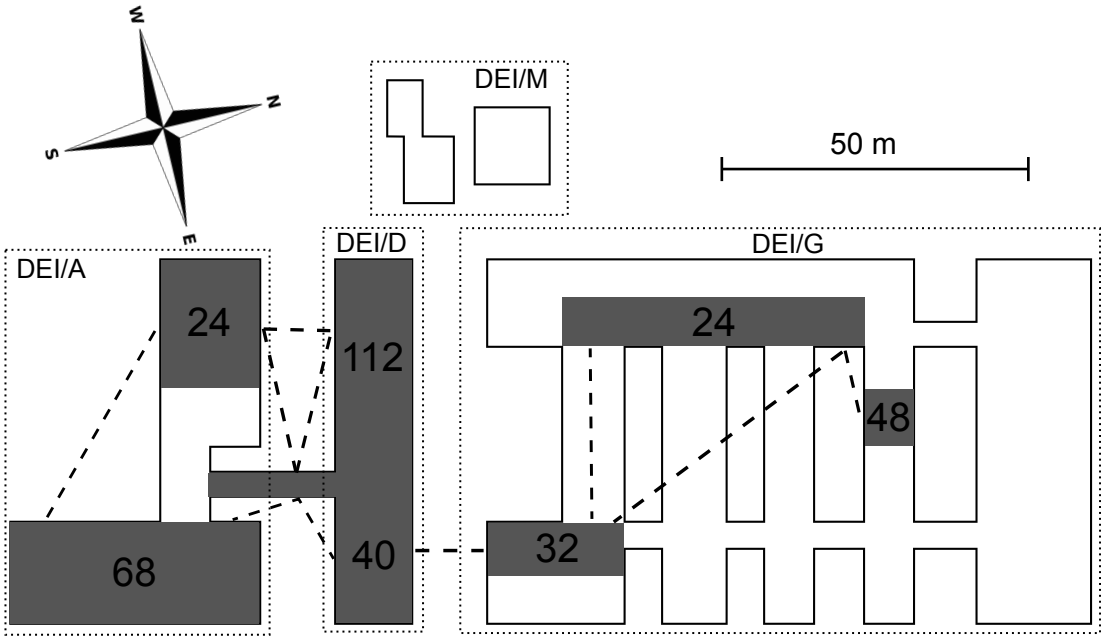

Figure 3.1: Graphic representation of the testbed at the University of Padova.

integration with identification technologies (e.g., RFID) and a reliable backbone network, Ethernet in our case. Moreover, this application calls for complex security features, since the network needs both to authenticate the users entering a certain area and to avoid granting passage to forged IDs. On the contrary, guest services do not need a particularly high security level, but must be able to manage a large number of users, hence requiring high scalability. In fact, in order to route visitors through the building, the system needs to run a reliable positioning system while, at the same time, reporting information (such as location) to the users. This can be performed with low-power radio interfaces, such as the IEEE 802.15.4, so that users only require a USB dongle for setting up communications with the fixed network.

Even though our testbed can offer each of these services, in this chapter we only focus on those enabled by the wireless sensor network backbone. In its actual configuration, outlined in Figure 3.1, our testbed consists of more than 350 static sensor nodes and 100 more nodes that can be used both as mobile stand-alone devices or as USB devices for laptop connection.

By using Synapse++ [60], a fast and reliable over-the-air reprogramming sys-

tem, the static backbone can be programmed with any of the following applications: *i*) Web Services; *ii*) Localization; *iii*) Experimental protocols. Experimental protocols and Localization can be installed on demand in an arbitrarily large fraction of the network; however, Web Services [7] constitute the default application being run by the nodes using the Binary Web Service (BWS) protocol.

In more detail, web service enabled nodes communicate through the IEEE 802.15.4 radio using the IPv6/UDP [52] protocol stack, thus enabling interoperability between our testbed and the Internet: in other words, it will be possible from any browser to open a page on the IP address of a specific node and look up/subscribe to its offered services, or read its sensor data. The majority of our nodes offer baseline sensing capabilities (light, temperature and humidity) as well as some management parameters (battery level, transmission power); however, nodes installed in specific locations are programmed with additional services: for instance, sensors in the proximity of the teaching rooms can be asked for the schedule of that room as well as a booking service; offices have sensors that can record the name of the visitor and the time of the visit; furthermore, the administrative staff can use the testbed in order to read the temperature in the whole building, and steer it to a comfortable level for employees and students.

We can connect our testbed to available actuators, such as heaters, air conditioners, light switches, etc. and a number of new devices, such as an automated door key lock with RFID, routing monitors for visitors and so on. In terms of smart automation, we can develop control mechanisms to monitor the environmental status of the buildings and generate reports if critical or abnormal conditions are found. The communication aspect is of fundamental importance, as we aim at connecting every device using standard mobile protocols either by browsing through gateway nodes or by directly accessing nodes through native BWS. Thanks to the aforementioned possibilities, our wireless sensor network becomes much more than an experimental research tool, turning it into a complete infrastructure allowing University users to experience typical IoT services.

3.2.2 Smart Grid [5]

Since the end of the twentieth century, several factors have started to change the energy scenario: the foreseen oil shortage brought to the forefront research efforts

for new and renewable energy sources; the increasing demand for energy called for a drastic improvement of the efficiency of the energy production and distribution plants, and a new attention to the environment changed the behavior of many energy players, leading them towards a more "green" attitude of judicious use of the energy resources.

A first consequence of these trends has been a model change in the energy market. From a monopolistic, single-provider scenario, the market is going through a number of intermediate phases featuring several players, mostly providers and vendors, and is expected to ultimately approach an open model where consumers may themselves become producers, thanks to the availability of cheap photovoltaic panels and other affordable and easy-to-use renewable energy sources. This new market model is clearly more dynamic, due to its distributed nature, and also because the instantaneous availability of energy will depend on sunlight, wind or other similarly intermittent sources. The behavior of the users is also undergoing substantial changes, as energy providers are aggressively promoting energy usage during off–peak hours by offering cheaper tariffs.

Central administrations are fostering this change by committing large amounts of money to support more efficient energy distribution and management models. For example the U.S. Presidency has already invested billions of dollars on grid-related projects [61], including research, development and assessment. The overall objective of this operation is to improve energy efficiency, by promoting energy savings and the adoption of alternative energy sources.

The Smart Grid (SG) is the technological paradigm that is being proposed to satisfy the aforementioned needs: SGs are expected to spread the intelligence of the energy distribution and control system from some central core to many peripheral nodes, thus enabling a more accurate monitoring of energy losses as well as more precise control and adaptation. By including SGs in the IoT, a number of advantages will become directly available. For instance, the system can leverage on widely accepted security and privacy frameworks, on broad connectivity and seamless interoperability, on the possibility of developing cloud computing systems for enabling service virtualization and distribution, and on the availability of a rich set of widely accepted standards.

The visions of SGs and IoT have been recently combined into the Internet of Energy (IoE) [62]: Figure 3.2 shows a graphic example of the scenario. The

Figure 3.2: The Internet of Energy: a vision of the evolution of Smart Grids into the web.

IoE will allow a decentralized control of the network, thus relieving the network itself from the communication burden needed to harvest data and gather it into central servers; web and mobile applications can be easily realized as standard web services and exploited by energy consumers and small producers for real–time energy consumption monitoring and optimization.

Furthermore, the recent trends of IoT communications [63] are promoting the usage of low–cost, low–power devices, thus contributing to reducing the power demand for energy grid operations, as a further step towards lower power consumption, improved energy efficiency, and lower electro-magnetic pollution. Due to the inherent hardware limitations of these constrained devices (e.g., computation and storage), communication technologies need to be extremely efficient, and therefore require the use of highly optimized versions of Internet protocols [11].

There are two key factors for the acceptance of the SG vision by the community at large: demonstrating the benefits that producers, consumers and users can derive from an intelligent and efficient management of the energy distribution, and a thorough support for a wide set of accessible application for both everyday energy interactions and market scale operations. The most typical applications of a pervasive Smart Grid system include, but are not limited to:

- Asset management and fault tolerance
- Control and management of demand, outage statistics
- System support and reconfiguration
- Integration of distributed resources

Most Smart Grid scenarios involve metering of one or more physical quantities that characterize the way energy is produced, transported and consumed over any sort of premises. Note that "energy" here is not necessarily restricted to electricity, but may involve other sources as well, such as natural gas. Depending on the scenario, the quantities to be measured and the frequency of the measurements may vary considerably. For example, the energy consumption in a house may be monitored in many ways. One could generate periodic reports of the electric power drained by home appliances, in order to come up with policies such as "do not use this set of appliances together", "it is better to defer the usage of this appliance to low-cost hours", or "if only a quick room warming is needed, the electric heater is cheaper than gas heating". Applied to a broad slice of the population, such policies not only have the potential to bring savings to individual users, but may also help quench the overall energy consumption of an entire area or country. Beyond energy consumption evaluation, a SG residential system may also be involved in monitoring the quality of the power transmission over power lines, and hence infrequently but regularly generate a set of very high-rate samples of the 50/60 Hz AC waveforms, with the objective of detecting glitches or excess reactive power consumption by powered objects. Even though this would generate a very large amount of data [64], it is not necessary to transfer all of it to remote control centers: smart meters within the house may be in charge of detecting anomalous situations locally, and possibly only report the most relevant sets of samples for external analysis.

The extension of Smart Grid technologies to an urban scenario leads to "smart neighborhoods," where multiple smart places, such as houses, shops, multi-dwelling residential units and shopping centers are jointly monitored and managed, in order to ensure continuous provision of power and timely reaction to faults or to overloads. The typical scenario in this case is a wide-area monitoring application gathering consumption measurements from around the neighborhood and

deciding how to route power through the grid [65] so as to avoid that critical distribution lines are excessively loaded. Applied at both the local and the global level, this system will eventually help relieve the risks of power outages and the consequent economic impacts (estimated at around 150 Billion dollars for major power outages in the US [66]).

Factory scenarios are usually characterized by different objectives than home scenarios. Power usage monitoring and anomaly detection are employed for prompt maintenance of production lines, for the safety of operators and for open as well as closed-loop control of production processes [67]. Beside energy supply, Smart Grid systems, applied to an urban scenario, can be exploited to monitor other vital systems, such as sewage or water-supply in order to automate the generation of quality reports, alarm propagation, and the optimization of resource distribution [63]. Although some security concerns remain when routing critical power management data over the Internet, the realization of the Smart Grid vision has the potential to bring considerable advantages to people and to the economy, in terms of power savings and efficient energy management.

Small, cheap, resource-constrained devices, extensively investigated by a wealth of studies in the wireless sensor and actuator networks (WSANs) area are deemed to be the key enabling technology to make the SG implementable in practice. The maturity reached by the research on WSANs and by multiple generations of devices has made them capable to autonomously organize and reliably route data through large multihop topologies. WSANs are a widely recognized technology to efficiently realize pervasive computing applications: this capability is extremely useful to the practical realization of an efficient energy monitoring and control system. In fact, several nodes can be inexpensively deployed close to key power usage points, even in typical cost-constrained environments, such as residential ones. The main problem in using WSANs as the technology foundation of Smart Grids is to understand *i*) which protocols or data formats can efficiently transport the wide variety of data expected in a SG system; *ii*) which communication technology best serves as the user interface to a machine-to-machine system such as the SG; *iii*) which current standards are believed to be flexible enough to boost SGs as an interoperable multi-vendor technology.

The described scenarios and the issues above are the ultimate objectives for the advent of the Internet of Energy. To this extent, some technical goals are still

to be achieved, and will constitute the next challenges for researchers to implement (and for the market to adopt) the IoE. From a networking point of view, a standardized solution is needed for letting constrained devices, such as WSANs and other IoT hardware, be seamlessly used in the Web. For what concerns communication practices, common languages and scalable data representation are paramount for the IoE market. In any event, the final enabler of the IoE is the usability of the system by untrained users, a feature that must be carefully addressed.

3.2.3 e-Health [6]

The Internet of Things can also be a suitable communication framework aiding the migration to e-Health systems. Patients subject to remote assistance will be the core of the system, and will be constantly aided by next-generation, low–power, low–cost, Internet-enabled smart healthcare devices.

The devices each patient is equipped with, e.g., blood pressure monitors and others wearable sensors, will be directly connected to his healthcare records, that can be either stored into home appliances or using cloud-based storage systems. These devices will be smart enough to provide reminders, and assistance to the patient; moreover, as soon as they spot alert conditions, either the patient or the physician can be digitally advised of the current situation.

In particular, security attributes define and constrain what healthcare can be accessed by any given user: for example, a physician can access any information related to medical devices that his patients are wearing, but he will not be able to know the geographical position of the patient; however, during critical emergency, all the required information will be provided to the rescue team.

Beside patients, the IoT system will know hospitals, medical clinics and physicians, in order for patients to locate their nearest contact point for medical information and assistance. Again physicians information is subject to security policies, so that they can be contacted directly by authorized patients, searched for from a list or contacted in case of emergency

Finally, healthcare management software will know any medical device available to the patient or physician, and may communicate when needed directly with those objects to check patient conditions during emergency or for management

purposes; moreover, it can promptly notify users of any emergency situation, and, in this case, may also activate alternative methods of communications such as sending an e-mail or placing automatic calls using a software PBX, e.g., Asterisk.

3.3 Technical Requirements [4]

Notwithstanding the constraints (especially in terms of nodes computational power and storage capabilities) of the aforementioned scenarios, support to standard protocols adapted for operation in a WSN (such as 6LoWPAN/UDP) is highly recommended in order to achieve interoperability and integration with current Internet-aware devices. In light of these considerations, a minimal set of protocols encompassing all required functionalities is to be selected, in order to minimize complexity by maximizing code reuse. To keep network operations efficient, the architecture of the network hosting these functionalities should be scalable and easily extendable. To this end, we envision a resource-oriented paradigm, whereby heterogeneous services are provided both to sinks and to mobile nodes, which may be heterogeneous and not designed to receive a custom service in a specific network.

We distinguish among three types of nodes with correspondingly different feature sets, depending on node mobility, operating range, and level of specialization: *i*) Base Station Node (BSN), e.g., an IPv6 sink/router; *ii*) Mobile Node (MN) (e.g., wireless dongle to add WSN connectivity to a standard laptop); *iii*) Specialized Node (SN) (e.g., offering services like temperature readings or actuation). BSNs are usually static nodes, bearing no specialization and a network-wide operating range; to this end, they must be provided with bidirectional and simultaneous communication with one, many and possibly all nodes in the network, also exploiting data aggregation techniques as appropriate. Usually BSNs have direct connection to the Internet and can provide connectivity to the WSN; a dedicated channel for receiving events and subscribed data is also required. MNs, in a typical use case, are external nodes running no specific firmware for the WSN in use, but rather featuring compatibility with the network specifications and protocols. Given such compatibility MNs should be provided zero-knowledge access to any particular node in the network, possibly including BSNs; to this end, network

probing, direct access to resources, temporary network join and resource subscription are relevant features to be supported. Finally, SNs are nodes in charge of delivering one or more very specific services, which makes SNs become a core part of the network, and potentially the most limited devices. Even though they are specialized, they might be in charge of diverse activities: they need to serve requests by BSNs, MNs or even other SNs requiring cooperation or relaying.

This preliminary description allows us to identify a set of requirements that should be supported by the network communication paradigm. In light of the interoperability and integration of the network with Internet-based communications, we choose to employ the Representational State Transfer (REST) paradigm [68] well known in the Internet domain, whereby any resource is addressed by a unique identifier of standard format. The features to be supported are summarized as follows: *i*) direct simple resource request/response; *ii*) concise one-to-many resource request/response; *iii*) structured resource request/response; *iv*) resource subscription and event or delayed notification; *v*) zero-knowledge network probing.

While being powerful enough to address interoperability, REST makes the access to any resource as easy as a web server interrogation. REST also simplifies the development of the network communication paradigm, which can be built around a single protocol by properly leveraging our flexible Binary Web Service [7] implementation in a versatile resource-oriented node design.

3.4 SENSEI case-study [7]

Thanks to our active involvement in the European SENSEI project [53], we realized a prototype of IoT system feasible to address the technical requirements described above.

Our framework implementation is optimized, so as to allow easy provision and configuration of real-life resources. In the context of the European SENSEI project [53], the BWS module has been connected to two different Binary XML Services, Resource Access and Resource Publish, providing project-specific node interfaces.

The Resource Access Interface (RAI) provides access to specific resources identified by the URL or the ID specified by the BWS module. BWS methods

are mapped as follows: *i*) GET provides a reading of the current value of the resource; *ii*) PUT sets, if applicable, the resource value or inputs a new command; *iii*) POST is used to subscribe to the resource by setting an appropriate criterion to push notifications directly to the sender. The Resource Publish Interface (RPI) is used instead to provide a comprehensive description of node properties and Web services offered to a BSN.

RAI and RPI communication is based on an out-of-band agreement on an XML schema representing the information to be conveyed in the various operations, and then on an XML content exchange whenever required. EXI [69] has been selected as the standard format for Binary XML: however a full implementation of an EXI encoder/decoder is not advisable for a SN; therefore, a simplified implementation is to be preferred.

By investigating the EXI standard and the agreed XML schemas, the EXI coded schemas have been mapped to static and variable parts; furthermore, the process of encoding and decoding an EXI request body is done as follows: for every resource, headers, footers and separators are known; between such tags, resource-related values are read or written according to the simple algorithm required to encode/decode numeric values in the EXI coding.

Considering the well-known `telosb` sensor node architecture [34], we evaluated the ROM/RAM usage of the system described above, and summarized it in Table 3.1. We highlight that implementation of the Binary Web Service module makes efficient use of both ROM and RAM size, independently of the number of clients or servers required by the upper layers. The RAI component translates request and associates them to the appropriate resource; it also implements a highly specialized EXI encoder/decoder which proves to be effective in terms of ROM occupancy. Resource components (temperature, humidity, light, etc.) require a larger memory footprint mainly because a specialized driver component is required for every on-board chip. RAM occupancy, on the other hand is low with respect to the available space (approximately 10 kB), except for the UDP/6LoWPAN implementation which requires a large static RAM allocation, mainly due to a 1280-byte buffer required for re-assembling fragmented datagrams.

Table 3.1: TinyOS components ROM/RAM utilization

Component	ROM	RAM
TinyOS core	1398	4
802.15.4 and ActiveMessage	9418	328
UDP/6LoWPAN	5182	1936
BWS	2950	326
RAI/RPI	1374	156
Resources	9800	354

3.5 Lessons learned and Vision [4]

Thanks to the experience acquired up to the current stage of the SENSEI project, we have envisioned a next-generation system which leverages on the strengths of the architecture set up to date. A Web Service model for WSNs has proved to be valid for the current purposes and a careful implementation is strictly required to adapt this communication model to a limited sensor node. The versatility of our implementation, together with the flexibility of Web services allows to make further steps towards a fully integrated system built around the BWS component.

Our vision has required that every interaction, as those devised at the end of Section 3.3, are managed internally by the BWS. Each interaction will be mapped to use standard REST methods, paired with a proper XML definition of the data required in the process, to allow strong code reuse even for very different operations or services offered.

As shown in the previous section direct simple request/response interactions have been easily implemented, even though a specific interaction to gather many responses from different nodes through a single request (concise one-to-many communication) has been currently left as a future work. Another interesting feature required by next-generation systems is support for structured requests, required to gather multiple values from the same node; also, interpreting complex requests based on the current state of the resources (e.g., turn on the air conditioning system in rooms with temperature higher than a certain threshold and where the lights are turned on) is also a required function that can be provided. In any event, the previous interactions will be implemented thanks to the BWS

component flexibility which, together with the versatility of Web services, can support complex XML interactions without redesigning the paradigm and the components already in use.

In this vision EXI coding plays a central role, as binary XML coding should be easy to implement and should allow strong code reuse in order to facilitate the implementation of multifaceted Web services on sensor nodes. However, our understanding of EXI format has led to the conclusion that the procedure required for coding two different XML schemas with minimal differences could be completely different, so a minimal variation of the schema may require a very different implementation. In this light, we have started evaluating the EXI coding for sensor nodes, by building an XML schema pre-processor that will directly output the variable part of the C code required to encode/decode that schema; as a second step, we will supply the pre-processor with a set of optimizations aimed at minimizing the output code size.

The last step in building our next-generation network will be the standardization of the offered resources and services. This will be accomplished by assigning a URL to all resources in a standard, reusable and extendable fashion, and then by mapping the procedure to opt-in and configure guest mobile nodes to a web service URL. We are confident that such a system can be easily replicated in different scenarios through small changes specific to the different resources and services offered, but without requiring any modification to its core architecture.

Afterwards this experience, a new standardization effort has been started at the IETF called 6lowapp [70] with the goal of realizing application layer paradigms for constrained networks and devices. That working group has been recently named as Constrained RESTful Environments [71], which is currently leading to the standardization of the CoAP protocol [10].

4

IPv6 on Smart Objects

In this chapter we present an overview of our innovative network layer design, that introduces advanced memory management in TinyOS, to efficiently handle concurrent transmission, reception, and relaying of multiple IP packets. Our design allows even tightly constrained devices to hold longer queue lengths at the IP layer, thus permitting higher throughputs and lower loss probability.

Section 4.1 motivates the choice of IPv6/6LoWPAN and RPL as network layer protocols; Section 4.2 presents our innovative design of the network layer implementation, together with its experimental validation over real hardware boards, and its performance comparison against a state-of-the-art implementation, i.e., BLIP [72].

In Section 4.3, following the research lines identified in [73], we develop practical congestion control algorithms for constrained Internet of Things (IoT) exploiting 6LoWPAN technology. These networks are characterized by very constrained processing, memory and communication capabilities [4], a potentially large number of nodes, and infrequent communication patterns which very much differ from standard Internet flows. The main contributions contained in Section 4.3 are:

- We propose a number of practical and lightweight congestion control algorithms for constrained devices, devising CC policies based on distributed back pressure control, with the objectives of detecting and alleviating network congestion, providing reliability and ultimately controlling the injection of data traffic into the network.

- We present extensive simulation results by comparing the performance of

the proposed CC policies with that of ideal back pressure algorithms and showing that layer-3 BP congestion control is feasible on constrained IoT devices, and results in significant performance gains at the expense of a minimal added complexity.

- We present protocols and results for unidirectional and upstream data traffic as well as for bidirectional CoAP flows.

4.1 Overview [5]

Many platform integrators have been providing proprietary solutions for realizing the Internet of Things, and many standardization activities have been started for bridging the IoT to the Web. The solution proposed for the networking of smart objects, as well as its implementation as a full-fledged web-operable tool, must use emerging web standards, easily interoperable with widely adopted HTTP-based web services. This is mandatory for the system to work as a proper extension of the web. To address the requirements described in Section 3.2, we choose those standards being discussed within the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C).

The IETF is leading many efforts on networking constrained devices with low-energy requirements, whose results will be directly applicable to Smart Objects. The interest for IP-based solutions on devices with various hardware constraints, e.g., WSANs, motivated the creation of several working groups, aimed at steering the adaptation of typical Internet standards towards variants more appropriate to constrained wireless networks. Such activities include *i*) IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) for the network layer; *ii*) Routing Over Low power and Lossy networks (ROLL) for the routing of datagrams; *iii*) Constrained RESTful Environment (CoRE) for the application layer. Moreover, W3C's definition for the Efficient XML Interchange specification (EXI) is being completed. This effort was mainly motivated by the need for a compressed binary representation of the eXtensible Markup Language (XML) on devices unable to use XML in its original format due to memory limitations.

4.1.1 IPv6 over Low power WPANs (6LoWPAN)

6LoWPAN has been the first working group (WG) formed inside the IETF to investigate this topic [74], with main focus on providing Internet connectivity to constrained WPAN devices through IPv6.

The WG has been initially focused solely on designing a cross-layer solution for IEEE 802.15.4 networks. This initial effort resulted in the publication of RFC 4944, which defines the frame format, fragmentation method and generic header compression technique required to fit IPv6/UDP datagrams in the very limited IEEE 802.15.4 frame size. The 6LoWPAN format allows access to the IP world to a completely new generation of networked devices: cheap constrained hosts can get access to the Internet via the large addressing space of IPv6, which is fully supported under 6LoWPAN.

The most important technical feature introduced in 6LoWPAN is header compression. First of all the minimum size of the IPv6 header, without extensions, is 40 bytes, which would greatly threaten the adoption of IPv6 on constrained devices, because it amounts to about half the payload size of a regular packet. In order to heavily reduce IPv6/UDP header size while maintaining the functionalities and addressing space size, a cross-layer optimization approach has been used. Header compression is applicable to devices sharing the same network. In this case, some portions of the IP header are inferred from the MAC header, e.g., IPv6 link-local addresses are derived from the MAC addressing fields. An IPv6 40-byte header can be shrunk to a single-byte HC1 header; through similar considerations the UDP 8-byte header can also be reduced down to a 4-byte HC2 header.

The compression mechanism heavily relies on deriving upper-layer fields from MAC-level information; in order to overcome this limitation, a stateful header compression technique is being discussed in the WG, which can also avoid making explicit reference to data deducible from lower layers. This independence is obtained by exploiting information shared between the communication endpoints. The 6LoWPAN format has been showcased by several implementations, which have demonstrated its feasibility for severely constrained devices. ETSI, OMA, and IPSO, among others, strongly promote a wide adoption of this standard, and a new class of smart, IP-enabled objects is expected to populate the

market of the IoT.

4.1.2 Routing Over Low power and Lossy networks (ROLL)

IETF tasked the ROLL WG [75] to lay down the Routing Protocol for Low power and lossy networks (RPL) specification [76]. An explicit design guideline for RPL is the support of different routing objectives: in SGs, this means that alarm propagation can be routed to offer low delivery delay, whereas background monitoring traffic can be configured to cause limited energy consumption.

Different instances of RPL can co-exist in the same network, in order to optimize routing structures, called Directed Acyclic Graphs (DAG), according to different metrics. Every instance deploys one or more destination-oriented DAGs, (DODAGs for short), which are rooted at a specific node, e.g., the data center collecting measurements from smart appliances. All data is routed through the DODAG to its root. Roots form DODAGs by propagating DODAG Information Objects (DIOs) downstream via the Trickle probabilistic broadcasting algorithm [63], which helps suppress redundant transmissions of the same message. The reception of a DIO causes nodes to select their rank, which is a measure of distance from the root of the DODAG, and to choose one or more parent node among those neighbors characterized by a lower rank. The chosen parent will then be the preferred next hop when routing towards the root of the DODAG.

To allow the root to become aware of the tree and thus be able to perform downstream routing towards the nodes, each receiver of the DIO propagates a Destination Advertisement Object (DAO) towards the root (the message will reach the root through all parents of the nodes along the path). In case intermediate nodes are "storing nodes", they will keep track of the local topology of the tree, and tell the route to reach the DAO sender through themselves. Otherwise, the information required to reconstruct the downstream route will be stored in the DAO, so that the root can reach every tree node via source routing using the information in the DAO.

The behavior of RPL is particularly suited to SGs, where each node in the tree may be connected to a different appliance or sampling point along the power distribution cables: routing trees connect all nodes to a data center, and may be organized as needed. For example, third-party home emergency detection

systems may leverage on the available network by providing an additional root node to collect data via a low delay high priority tree, which RPL allows to exist on the same SG wireless network. Similarly, the data centers of different utility providers can share the same wireless network to route data. For example, the sensors attached to the power grid and the sensors attached to the gas pipes may collaborate to route each other's data whenever this provides advantages in terms of the routing metric being optimized: provided that all nodes are part of the same SG network, the adoption of RPL makes this cooperation seamless.

4.2 SiGLoWPAN [8]

In this section, we describe an innovative implementation of the IPv6 protocol stack, whose main features are: *i*) an advanced memory management approach, *ii*) link-layer independence, and *iii*) optimized RAM/ROM footprint. More specifically, we present the design principles at the basis of our implementation of IPv6/6LoWPAN, called SiGLoWPAN, and the results of a preliminary experimental campaign of such technologies in order to better understand their applicability to smart object systems.

4.2.1 Related Work

In the past decade, Wireless Sensor Network (WSN) research has been more focused on protocol optimization rather than on defining the protocol stack architecture. An effect of this trend has been noticed in the stack implementations, e.g., Levis et al. [46] noted a lack of consensus on the networking abstractions required in TinyOS, a popular open-source operating system for embedded devices. With the aim of defining a networking abstraction, TinyNET [2, 3] provided a networking framework able to support a wide set of protocol interactions and network services, including 6LoWPAN, but traditional layering was still not adopted.

As the IoT concept became more popular, the feasibility of more traditional networking stack architectures have been investigated [4, 11] with the aim of building smart objects using a set of protocols as similar as possible to the ones widely adopted in the Internet. At the same time a number of implementations

of the IPv6/6LoWPAN protocol suite have been developed by many institutions. Out of those, the most popular open-source implementations [49] are BLIP for TinyOS, and uIPv6 for Contiki.

Recently Chauvenet et al. [77] showed the applicability of 6LoWPAN techniques on Low-Rate Power Line Communication (LR-PLC) by evaluating a MAC similar to IEEE 802.15.4 and performing 6LoWPAN header compression over it. We believe that the IPv6/6LoWPAN protocol stack suits well the Smart Grid use-case as well as the general IoT concept. However, the IP stack should be carefully designed to be easily portable to different physical layers, flexibly support IP layer routing (e.g., RPL [76]), handle concurrent communications, and fulfill strict memory requirements posed by the SG scenario.

In this section a novel IPv6/6LoWPAN implementation, SiGLoWPAN, is presented and compared with BLIP [72], highlighting the benefits of its design in the SG context. In particular, its novelty is related to the memory management approach used in the stack, which enables a better support for multi-hop communications typical of meshed low-power lossy networks, such as those expected in real-world deployments of the SS.

4.2.2 Requirements and design goals

Since a large number of different transmission technologies are expected to be involved in the any Smart System, and on top of that, even single SG subsystems could require the cooperation of different wired and wireless technologies [78], an IP stack implementation suitable for the SG has to properly address the technical difficulties that may arise from such a complex networking infrastructure. This goal is even more challenging if we keep in mind that many SG devices are cheap hardware-constrained nodes which can handle only a very limited set of networking protocols.

With the aim of addressing such requirements, the design of SiGLoWPAN was driven by the following goals.

4.2.2.1 Effective layer-3 routing

Given the very wide range of medium access technologies that will be involved in the SG, the requirement of effective routing support on top of different link-layers

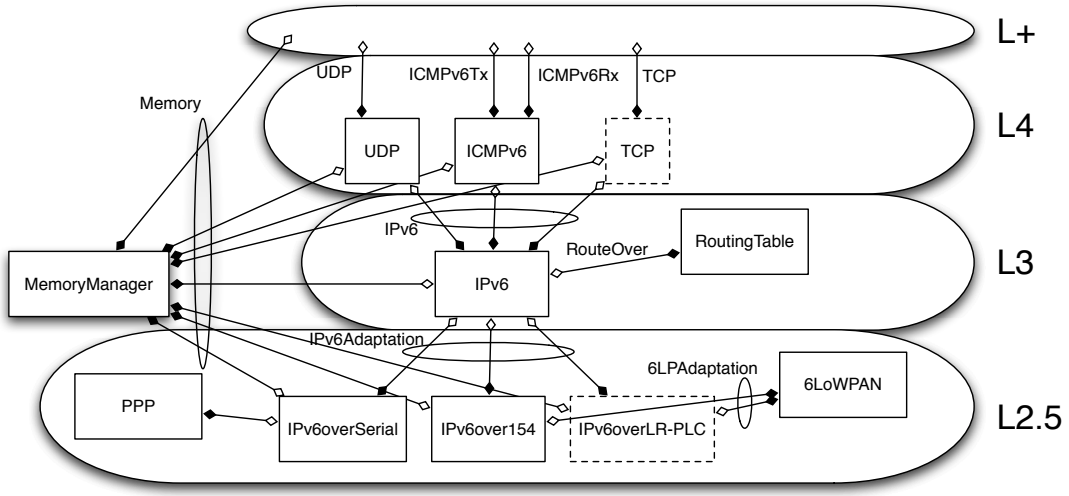

Figure 4.1: SiGLoWPAN architecture

is a key objective for SiGLoWPAN. Even if other implementations already support L3 routing, our architectural design and in particular the advanced memory management approach spanning from the lower layers up to the application allows SiGLoWPAN to efficiently handle relaying operations even on very constrained devices with few kilobytes of RAM and limited computing resources.

4.2.2.2 Link-layer independence

Motivated by the same consideration about the number of PHY layers upon which the SG will be built, the modules composing SiGLoWPAN have been carefully selected with the intent to achieve a cleanly layered implementation, possibly maintaining one-to-one correspondence between a protocol and the module where it is implemented. Thanks to this flexible but clean modularization, which confines each protocol within a single component, a high-level of flexibility and code reuse has been possible in our complex stack implementation, which supports 6LoWPAN independently from a specific link-layer and point-to-point protocol (PPP) communication over the serial link.

4.2.2.3 Lightweight implementation

The constrained nature of the majority of the devices involved in IoT scenarios, imposes hard limits on the number of instructions and the amount of memory that

a device can handle. Although the efficiency of the implementation initially had lower priority with respect to the other objectives previously discussed, a careful evaluation of the design choices made it possible to pursue this requirement as an additional goal. Thanks to the clean modularization, and by maximizing the level of code reuse, SiGLoWPAN implements the whole set of protocols using a small fraction of the ROM resources available on a realistic 16-bit RISC MCU (i.e., MSP430). On the same platform, the aforementioned memory management approach leads also to a very efficient RAM allocation. As a consequence, this efficient resource usage has become a key feature of our implementation, and allows the system to allocate more memory to the small IP queues, which are typical of constrained devices, thus directly improving the transmission and relaying performance network-wide.

4.2.3 Architectural overview

As discussed in the previous paragraphs, and shown in Figure 4.1, the SiGLoWPAN architecture is clearly layered and organized in self-contained modules corresponding to single protocols. As an exception, the memory management module spans across all the networking components up to the highest application layer of the node (L+). Besides the well-known layers 3 and 4, Figure 4.1 also shows the IP adaptation layer (i.e., layer 2.5) that performs the adaptations required to transport IP packets over specific link-layers.

From a high-level perspective, SiGLoWPAN gives the application layer access to the IPv6 stack by means of the TinyOS interfaces of the layer 4 protocols, such as `UDP`, `ICMPv6Rx` and `ICMPv6Tx`. Different instances of such interfaces are present, allowing multiple components to be linked to the same protocol; multiplexing is obtained using protocol parameters such as the local UDP port, e.g., an L+ application will setup UDP listening by linking to a specific instance of `UDP` interface identified by the local UDP port number.

The architecture proposed in Figure 4.1 is *memory-centric* because of the centrality of the `MemoryManager` component, which is shared from the lower layer 2.5, up to the application layer of the stack. This design choice is motivated by the consideration that every component not using the `MemoryManager` must allocate a static buffer. The problem is that such a static buffer must be big enough to

accommodate the maximum data which could be supported by this component, leading to memory usage inefficiencies inside the application components. Since IPv6 MTU is 1280 bytes, in real-life applications a single datagram may represent a considerable fraction of the available RAM on constrained devices (e.g., telosb has 10kiB of RAM). In addition, when a datagram is statically allocated at L+, lower layers have to either fully process it immediately or duplicate it in RAM for delayed processing, which always results in some kind of inefficiency. Moreover, for datagrams received for forwarding, tighter requirements apply on immediate processing or duplication, thus dynamic memory management becomes the only feasible choice to efficiently handle the transmission chain across the stack, especially when concurrent routing of multiple IP datagrams is required.

Link-layer independence is another important design goal of our implementation. In order to meet this requirement, the design of adaptation layers has been carried out in a highly modular fashion. IPv6 module handles a set of IPv6 adaptation layers using the `IPv6Adaptation` interface. Modules providing such interface have to offer the capability to transmit or receive IP datagrams over a specific medium, one at a time. The 6LoWPAN header compression and fragmentation approach has been implemented in a separate module, shared across the subset of adaptation layers supporting such compression (i.e., IEEE 802.15.4, LR-PLC, BT-LE [79], etc.). Unconstrained mediums that do not require such kind of adaptation may use other techniques, i.e., serial communication with a PC has been implemented using PPP.

Apart from the IP layer routing, SiGLoWPAN provides the capability to perform layer 2 routing (*mesh-under*) through the `MeshUnder` interface available at layer 2.5. Even if this approach can also be used in conjunction with layer 3 routing, this topic is out of the scope of this work.

4.2.4 Components

SiGLoWPAN stack is composed of both TinyOS modules and reusable standard C files, which are described in Table 4.1.

The set of TinyOS modules forming SiGLoWPAN are shown in Figure 4.1. The main components will be briefly described in the following.

Table 4.1: List of SiGLoWPAN header files

6LOWPAN.h	IPv6 packet format definitions
RFC4944.h	fragmentation header structure definition, mesh-under header structure definition
RFC6282.h	LOWPAN_IPHC header structure definition, LOWPAN_NHC structure definition
RFC4944.c	fragmentation handling functions, mesh-under header handling functions
RFC6282.c	LOWPAN_IPHC handling functions, LOWPAN_NHC handling functions

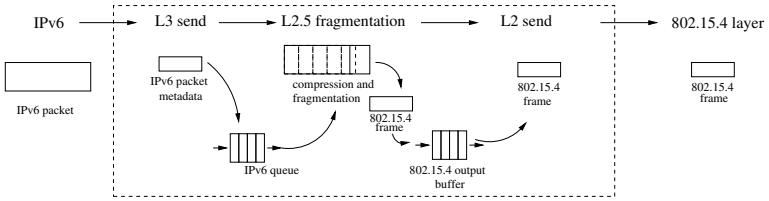

Figure 4.2: SiGLoWPAN sending procedure workflow

4.2.4.1 `MemoryManager`

In our *memory-centric* design, the `MemoryManager` component has paramount relevance. Though this module has been designed as a general purpose memory allocation component, it is currently involved only in handling the buffer space required for IPv6 datagrams. At compile time, a fraction of the available RAM of the node is statically allocated to this component, which will be assigned to the modules that need it by means of the *alloc* call. In order to enable complex interactions with buffer spaces, the memory manager makes the available RAM virtual by identifying each specific allocation using a virtual memory ID (`vmID`) rather than a physical memory address. The pointer to the physical address is obtained using the *id2p* call.

RAM virtualization enables transparent reallocation of the buffer space, which is especially useful in optimizing RAM requirements for single datagrams along the whole network stack. In fact, `vmID` is a global identifier for the buffer space and can be passed between layers removing the need for static memory allocations. This process is managed using the header reallocation feature of the buffer

through the *hrealloc* call. This function resizes the buffer by changing the physical address of the first byte and thus makes it possible to transparently add or remove headers to a buffer space.

When a packet is to be sent, at each layer of SiGLoWPAN the first part of the buffer is moved backwards and filled with the corresponding layer header, then only the `vmID` is signaled to the next layer. Network modules may also remove headers by calling the function *hrealloc* requesting a negative header reallocation on the buffer. The reallocation procedure is performed by the `MemoryManager`, by leveraging the fact that its memory pool is assigned starting from the bottom, the *hrealloc* procedure is simple and involves only memory pointer arithmetics without requiring any memory copying operation.

The current implementation of the `MemoryManager` component is still at an early stage of development and various optimizations are currently being evaluated, i.e., efficient proactive allocation techniques minimizing *hrealloc* complexity, or defragmentation policies to prevent fragmentation of the free memory space, thus allowing future usage of the `MemoryManager` component also for bigger long-term allocations.

4.2.4.2 L2.5

Layer 2.5 is the lowest level of the SiGLoWPAN stack, and performs the adaptation required to transmit IPv6 datagrams over a specific network interface. As the corresponding IP-over-X IETF specification, these modules can be either simple or complex depending on what are the operations required in order to adapt IPv6 packets for transmission on a specific L2 frame format. Modules at this layer receive a datagram to be transmitted as well as the next-hop target from the IPv6 component. As soon as the adaptation module is ready to receive a new datagram, it notifies IPv6 of this status by signaling a *sendDone* event.

The design of the `6LoWPAN` component required a more careful consideration in order to get a high level of code reuse across the different link-layer drivers. For this purpose we managed the interactions of `6LoWPAN` with actual layer 2.5 components designing a *6LPAdaptation* interface, which provides compression and fragmentation procedures for transmission, and their counterparts for reception. Each module using the *6LPAdaptation* interface offloads the operations required to build and process L2 frames to `6LoWPAN`.

Fig. 4.2 shows from a high level perspective the workflow performed during the send operation of an IPv6 datagram over IEEE 802.15.4 (6LoWPAN). As soon as the datagram to be sent is received by the `IPv6over154` component, the datagram and the data-link layer specific metadata required to compress the packet are passed to the `6LoWPAN` component using the *compress* command of the *6LPAdaptation* interface; the *compress* command builds a single L2 frame of the datagram at every call. As long as the `IPv6over154` has available slots in its internal output frame buffer, all the 6LoWPAN fragments are built subsequently; otherwise the remaining fragments will be built after each successful transmission.

The receive procedure is very similar, each unprocessed L2 frame is passed to the `6LoWPAN` component which will reconstruct the IPv6 datagram. As soon as the IPv6 datagram is fully reconstructed, it is passed to the corresponding adaptation layer module using the *6LPAdaptation* interface. This approach makes it possible to use a single queue to store the datagrams during the reconstruction phase. This queue is shared among the whole set of adaptation modules.

`IPv6over154` statically allocates a pool of L2 frames to hold the processed fragments waiting for transmission as well as the unprocessed fragments waiting for reconstruction. In our implementation the buffer space is shared for both the transmission and reception queues. Further investigation is needed to understand whether this buffer space may be shared also across multiple link-layer adaptation modules using different transmission technologies. This would further optimize RAM requirements and the overall efficiency.

4.2.4.3 L3

The layer 3 of our IPv6 stack is composed of the `IPv6` module and its companion modules, i.e., `IPv6Address` and `IPv6RoutingTable`.

The `IPv6` component is characterized by the following features: *i*) it provides *send* and *receive* primitives to the upper layers, *ii*) it keeps track of the available link-layer interfaces and their operational state, *iii*) it keeps track of the local IPv6 addresses along with `IPv6Address`, *iv*) it performs routing operations and decisions on each IPv6 datagram together with `IPv6RoutingTable`, and *v*) it provides IPv6 pseudo-header support to enable upper-layer protocols to perform checksum calculations.

For each datagram `IPv6` *puts in* or *pops out* the IPv6 header from the cor-

responding buffer space, leveraging the *hrealloc* function, and thus without any additional memory allocation.

Each datagram received by the IPv6 layer, either from the network or from the upper layers, passes through the routing engine which either *i*) delivers it locally if its destination corresponds to the local node, *ii*) flags it to be routed towards a specific link-local next-hop of an active adaptation interface, or *iii*) silently discards it.

The local delivery procedure is performed by querying the `IPv6Address` component to understand whether the target IPv6 is the local node. This component maintains a list of all the local IPv6 addresses, keeping track of the name and type of the interface associated to each address. The classification type of the interface depends on the address characteristics, i.e., unicast or multicast, global, site-local or link-local. The `IPv6Address` component is also used to assign a source IPv6 address to each locally outgoing datagram.

The routing procedure is assisted by the `IPv6RoutingTable` component, that is queried for the next-hop node, once the IPv6 destination of the datagram is given. This component maintains a table containing the next-hop host for every particular IPv6 prefix known by the local node. In order to simplify the prefix matching process, which could take a considerable amount of time, entries are kept sorted using the destination prefix length field, from the longest to the shortest. Maintenance of the routing table entries is out of the scope of the `IPv6` component. An appropriate routing protocol, e.g., RPL [76], is required to dynamically populate the table.

4.2.4.4 L4

Layer 4 is composed of the modules typically used by the application layer. In our implementation, we currently provide only `UDP` and `ICMPv6`, but additional protocols can be easily added to our stack by implementing a specific protocol on top of the `IPv6` module.

The `UDP` module handles UDP datagrams, i.e., IPv6 datagrams containing the UDP next-header received by the node delivered by `IPv6`. This delivery process is implemented leveraging static linking of TinyOS by means of parameterized interfaces. The same applies to `ICMPv6`.

Thanks to the RAM manager, when an L4 packet is sent, each module real-

Table 4.2: SiGLoWPAN ROM and RAM

Component	ROM	RAM
TinyOS & CC2420 transceiver	10714	383
`MemoryManager`	1002	6996[1]
L2.5: `6LoWPAN` & `IPv6over154`	7176	1904
L3: all	2572	956
L4: `UDP`	130	-
Total	21594	10239
Free space	*27558*	*6994*

locates the buffer and adds the proper header on top of it. Analogously, when a packet is received, the module extracts the header information and deallocates the buffer memory pertaining to the header, without any additional RAM requirement inside the L4 component.

4.2.5 Evaluation

An experimental evaluation campaign has been performed using our SiGLoWPAN implementation to show *i*) its RAM and ROM memory footprint, and *ii*) the achievable throughput at layer 4 compared to another 6LoWPAN implementation, i.e., BLIP.

Table 4.2 shows the overall ROM and RAM allocations across the different components of the stack. In the proposed experiment the `MemoryManager` offers 6170 bytes of RAM to the components, `IPv6RoutingTable` holds up to 20 entries, `IPv6` queue is 10 datagrams long, `IPv6over154` can queue up to 12 IEEE 802.15.4 frames, the `6LoWPAN` reconstruction queue is 10 IPv6 packets long and the application at layer L+ can send packets of arbitrary size up to 1240 bytes of payload.

It can be noted that more than two-thirds of the RAM is allocated to the Memory component and the remaining third is mainly split between L2.5 and L3 components. The memory assigned to the 6LoWPAN components is mainly used to allocate the L2 output frame buffer, but also to hold metadata required

[1]Unused RAM is assigned to the `MemoryManager`. This value includes 6994 bytes of free RAM available and dynamically allocable by L+ components.

for fragmented IPv6 datagrams reconstruction; the RAM allocated to layer 3 is mainly used to store the routing table but also for the metadata required by IP layer queuing. This result highlights the memory efficiency of our implementation that makes it possible to assign the biggest part of the node resources to network buffers and IP routing. As a future optimization, we target building the L2 output frame buffer and the routing table out of a dynamic buffer maintained by the `MemoryManager` component.

Table 4.2 shows the amount of ROM required for program code of the various SiGLoWPAN subsystems; the grand total of ROM occupied by SiGLoWPAN is 11kiB, required mainly by the 6LoWPAN header compression and fragmentation handling functions. This result motivates the introduction of our reusable 6LoWPAN component shared across the different network interfaces using this compression. ROM occupancy is about 2kiB less than BLIP, even though a detailed comparison is not possible because there may exist minor differences in the implemented features that have an impact on the program size; for example our protocol stack includes a more complex `MemoryManager` component supporting hrealloc, whereas BLIP has a simpler and hence smaller one.

To test the throughput of the whole protocol stack, an application implementing the Iperf [80] protocol has been developed over both SiGLoWPAN and BLIP measuring the goodput, packet loss and jitter of a UDP unicast constant bitrate (CBR) session.

Figure 4.3 shows a comparison between SiGLoWPAN and BLIP. In this experiment, we measured the goodput at the receiver between two telosb [34] nodes on the same IEEE 802.15.4 PAN within range of each other; to this end the client node starts a UDP CBR session towards the server which measures the average speed at which the data at the L+ layer is received. To perform a fair comparison between BLIP and SiGLoWPAN, all the comparable network buffers in the two implementations have been set to the same length. In our experiments the bottleneck was at the client side for both BLIP and SiGLoWPAN, which were unable to send the IPv6 datagrams faster than the speed shown, whereas at the receiver side the packet loss was negligible. Very small datagrams experience slow throughput due to the high fixed time required to successfully emit an L2 frame, the overhead added by the 6LoWPAN header, and its complex compression process. The sawtooth-like pattern in the graph is due to the reduced efficiency

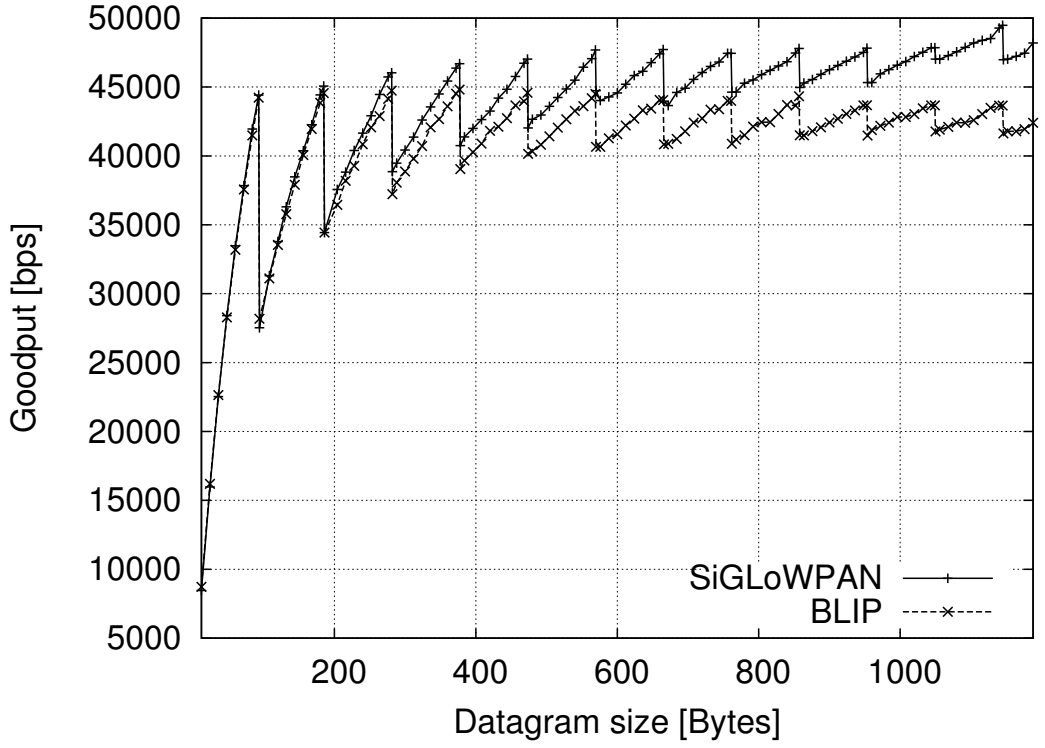

Figure 4.3: Receive-side goodput comparison between BLIP and SiGLoWPAN

caused by the introduction of an additional L2 frame required to transmit an IPv6 datagram when its last fragment grows exceeds the L2 frame MTU.

SiGLoWPAN outperforms BLIP for larger datagrams up to 13%, mainly due to the fact that BLIP fragments an IPv6 datagram at the 6LoWPAN layer in a single step, thus requiring that the L2 queue has enough space for all the fragments at once. SiGLoWPAN can build the datagram in successive steps, thanks to the advanced memory system supporting the *hrealloc* function that does not require static allocation at the L+ layers and makes it possible to perform the *free* operation at the lowest layers of the stack.

4.3 Back Pressure Congestion Control [9]

In the last few years, we have witnessed considerable advances in terms of protocol design for wireless sensor networking. These have led to a solid understanding of the problems related to channel access, routing and data gathering, delivering efficient protocol stacks and ultimately spurring the standardization of protocols for data collection and addressing.

The work in this chapter considers network protocols recently standardized by IETF, namely, CoAP [10] and 6LoWPAN [81], whose combined use permits Web-based bi-directional communications between sensor devices and Internet servers. 6LoWPAN provides header compression and specifies communication profiles that allow the implementation of IPv6 addressing. CoAP is a stateless protocol that is aimed at replacing HTTP for lightweight and resource constrained devices. As such, it implements a reduced set of functionalities with respect to HTTP. While CoAP and 6LoWPAN provide the basis for Web-oriented protocol stacks for embedded devices and natively support UDP traffic, they do not fully address the congestion problem, and only provide some conservative recommendations, as we discuss below in Section 4.3.3.

The Internet protocol suite, i.e., TCP/IP, has been designed adopting the "end-to-end argument" [82], which has been proven to be very effective in networks of smart terminals operating bulk data transfers. However, TCP congestion control (CC) [83] has been designed with an implicit assumption: data transfers causing congestion are usually long enough to be efficiently controlled through end-to-end CC algorithms. By their own nature, slow start and congestion avoid-

ance are techniques that converge after some time and after a potentially large amount of data has been transferred. However, when the amount of data required to create congestion on the network is very small, these techniques do not provide an efficient solution to the CC problem. In addition to this, TCP is known to be severely impacted by the long delays that are typical of constrained networks.

Our present work quantifies the benefits of implementing congestion control at layer-3 by exploiting practical and lightweight algorithms based on the concept of back pressure routing by Tassiulas and Ephremides [84]. Since its conception, Back Pressure (BP) policies have been extensively explored, leading to distributed theoretical algorithms that achieve optimal throughput performance in distributed networks. Practical applications of these schemes have also been studied in several papers such as [85], which applies a similar policy to the queues of wireless sensor nodes to realize an efficient data collection protocol. However, that solution makes strong use of channel snooping and poses limitations on the implementation of radio duty-cycling (RDC). In [86] the authors propose CODA, a distributed algorithm that uses explicit messages to detect congestion and therefore can also work in the presence of RDC. An evaluation of CODA in 6LoWPAN networks can be found in [87], where the authors measure the loss probability and the number of delivered packets. While these studies on CODA are of interest for the application of congestion control principles in energy efficient networks, some practical issues still remain open, namely: *i*) the explicit BP messages are not provided in standard existing protocols, and therefore cannot be used in standard networking stacks, and *ii*) there is no discussion on some important issues such as the effect of the required number of hop-by-hop retransmissions and of bidirectional CoAP traffic support.

Our main objective in this chapter is to systematically compare through detailed simulations different lightweight BP approaches, including existing as well as original algorithms, in order to assess their suitability for the implementation into IoT devices and their benefits in terms of performance gains.

The remainder of this chapter is organized as follows. In Section 4.3.1 we describe the system model and present our practical BP-based congestion control algorithms for constrained devices. First, in Section 4.3.2 we evaluate the performance of these algorithms focusing on unidirectional and upstream data collection. Thus, in Section 4.3.3 we consider bidirectional communication sce-

narios as those arising from CoAP-based Web-services.

4.3.1 Back Pressure Congestion Control on 6LoWPAN

In the following, we present some CC designs that are explicitly tailored to constrained networks featuring infrequent communication patterns. Specifically, we propose to perform congestion control actions at the network layer, as this allows the implementation of BP algorithms that work on aggregates of datagrams, i.e., on IP queues. Note that working on aggregates is desirable due to the nature of the traffic found in 6LoWPAN networks, which usually reaches considerable volumes only when the output of multiple nodes is combined. Moreover, this results in a lower complexity in terms of software structure, memory utilization and communication requirements for the control of network queues.

4.3.1.1 Node Model

Each node has been modeled according to the Internet Host model [88], which classifies protocols into Link, Network, Transport and Application layers.

- **Link:** 6LoWPAN has been specifically designed for the IEEE 802.15.4 PHY/MAC [89]. Thus, in our model each node is equipped with an IEEE 802.15.4 radio transceiver operating at 2.4 GHz with a nominal available transmission rate of 250 kb/s. Layer-2 operates according to the IEEE 802.15.4 standard and the total number of transmissions per packet is limited to a maximum of 7.

- **Network:** IPv6 and 6LoWPAN belong to this category; our node has been equipped with a standard layer-3 device (L3D) operating as follows. For each IP datagram, received either from the applications residing in the upper layers or from the radio, L3D first understands whether this datagram has to be delivered to the local host.

  As a second step, L3D looks in the Internet routing table, extracts the next-hop toward which the datagram has to be sent, and places the received datagram into the layer-3 queue for outbound traffic. This queue is managed according to a First-In First-Out Drop Tail (FIFO-DT) discipline. Note

that we account for a single IP queue at layer-3, this limitation is realistic and typical of constrained devices.

Our L3Ds implement hop-by-hop layer-3 retransmissions and different BP control algorithms as specified below, in Sections 4.3.1.2 and 4.3.3.1.

- **Transport:** The UDP transport protocol is adopted. UDP only performs a checksum check for every received datagram, without any further processing or buffering operations.

- **Application:** We have considered two usage scenarios:

  1. **Unidirectional flows (Section 4.3.2):** for the study of unidirectional upstream data traffic, we have adopted the Iperf [80] protocol. It permits to evaluate at the receiver the number of lost packets, the number of out-of-order deliveries, the multi-hop delay and the per-packet jitter. Data sources emit UDP traffic at a constant bit-rate (CBR), except for the cases where the local layer-3 queue is full. In these cases, the emission of the datagram is delayed until a layer-3 queue slot becomes available.
  2. **Bidirectional flows (Section 4.3.3):** to evaluate the effectiveness of the proposed congestion control algorithms for bidirectional traffic, we have used the Constrained Application Protocol (CoAP) [10] to transport Iperf messages. CoAP implements a lightweight bidirectional exchange targeted to client-server architectures. In this case clients are placed outside the constrained IoT domain and emit CoAP requests at a constant bit rate. These requests are sent to a border gateway and from here to the IoT nodes. Upon receiving these requests, IoT nodes reply with CoAP responses that flow in the opposite direction.

4.3.1.2 Layer-3 Device Types

We advocate the implementation of congestion control through the use of practical back pressure techniques, which are embedded into the layer-3 device of each sensor node. Next, these augmented L3Ds are presented in detail, whereas their performance evaluation is carried out in Sections 4.3.2 and 4.3.3, where we

respectively look at unidirectional and bidirectional flows.

Static With this term we refer to the L3D described in Section 4.3.1.1, which does not account for any congestion control mechanism. This is a baseline scheme considered here to gauge the advantages offered by the following BP schemes.

IdealBP refrains from transmitting as long as the queue length at the next-hop is higher than that of the local queue. This behavior mimics the ideal BP policies devised by Tassiulas and Ephremides [84]. Note that in actual implementations nodes can only know the queue length at the next-hop through the exchange of proper control signals. For *IdealBP*, in our simulations this information is made available to any node through a genie. Although *IdealBP* is impractical, we have considered it here to validate the BP approach and also see how much its performance deviates from that of the practical algorithms that we propose next.

According to *IdealBP*'s BP policy, the datagram at the current node is transmitted to the next hop whenever their queue differential is positive and the remote queue length at layer-3 is smaller than a pre-determined threshold $Q_{\text{thr}} > 0$. This threshold is required because it could happen that multiple devices concurrently send[1] their datagrams (one per device) to the same next hop. In this case, the queue at the next hops could overflow even though the preceding queue differential was positive.

Griping uses an explicit BP signal on congestion and is similar to the CODA BP policy [86], that has also been evaluated in [87]. Differently from [86] and [87], in *Griping* subsequent BP control messages must be transmitted at least K seconds apart, where K is a tunable parameter. In fact, we noticed that close transmissions of BP control messages toward the same source lead to inefficicies in terms of transmission overhead.

Whenever a *Griping* L3D receiver gets a new datagram and its layer-3 queue length is larger than a threshold $Q_{\text{thr}} > 0$, it emits a unicast BP control message toward the source of that datagram. At any time, each *Griping* transmitter sends

[1]Note that the concurrent transmissions occur at layer-3; lower layers will multiplex these transmissions so as to avoid collisions, by possibly retransmitting collided packets.

its own datagrams at a rate that is updated according to an Additive Increase and Multiplicative Decrease (AIMD) approach. Specifically, the rate is halved upon receiving a BP control message and is otherwise increased by one datagram every T seconds. Further, as stated above, subsequent BP control messages must be transmitted at least K seconds apart. In our simulations, the parameters K and T have been tuned and subsequently set to 100 ms and 750 ms, respectively.

Due to its simplicity, *Griping* is amenable to the implementation on constrained nodes. Moreover, we note that this technique does not require any interaction with the PHY and MAC layers and therefore does not rely on their specific implementation. This makes it possible to implement *Griping* with radio duty cycling, which is a critical feature for wireless sensor networks.

Layer-3 losses in *Griping* occur in two cases.

C1) Receiver side: a packet is correctly received at layer-2 and is passed to layer-3, where the network queue is full. The packet is thus discarded and a layer-3 queue overflow occurs.

C2) Sender side: a packet is discarded when none of the allowed retransmissions at layer-3 has led to its successful reception.

Deaf is an alternative approach that aims at removing the complexity associated with the transmission of BP control messages. Specifically, a *Deaf* receiver stops sending layer-2 acknowledgements whenever the layer-3 queue length is larger than a threshold $Q_{\text{thr}} > 0$. The stopped acknowledgment flow is perceived by the *Deaf* transmitter as an *implicit* BP notification.

A *Deaf* transmitter handles layer-3 retransmissions as follows: after a new transmission, a back off timer is initialized to W seconds, where W is drawn from a random variable uniformly distributed in $[0, c_W 2^{e_w}]$. After each retransmission of the same datagram the transmission counter, n_{tx}, is increased by one and the length of the latter interval is adapted by picking a new value of e_w as follows $e_w = \min(n_{\text{tx}}, n_{\text{tx}}^{\max})$, with $n_{\text{tx}}^{\max} = 4$. n_{tx} is initialized to zero for the first transmission and the constant c_W has been set to 0.1. We say that a packet failure event occurs whenever the maximum number of retransmissions is reached for a given layer-3 packet, which is still unsuccessfully delivered.

Note that this technique does require some cross-layer interaction between layer-3 and layer-2. In fact, a given layer-2 frame is not acknowledged by *Deaf* whenever the overlying layer-3 communicates to the lower layer a failure for that packet. However, this does not require to process further PHY- or MAC-layer metrics. For this reason, we argue that *Deaf* does not interfere with radio duty-cycling as others cross-layer approaches usually do.

Note that congestion control in this case is enforced by spacing apart the retransmissions of the same packet at layer-3. This amounts to decreasing the actual layer-3 transmission rate through implicit notifications from the *Deaf* receiver.

Deaf never acknowledges layer-2 packets whenever the layer-3 queue is full. Thus, event C1 above never occurs and packets can only be discarded due to event C2.

Fuse *Fuse* behaves as *Griping* until its queue length is smaller than the maximum queue size $Q_{\max}$. When the layer-3 queue is full, *Fuse* combines the BP actions of *Griping* and *Deaf*. That is, in this case *Fuse* stops sending layer-2 acknowledgments (as *Deaf* does) but also continues to send explicit congestion notification messages (as *Griping* does). As we shall see below, this combined action effectively reaps the benefits of both *Deaf* and *Griping* BP policies.

4.3.2 Unidirectional Upstream Data Traffic

In this section we analyze the performance of the back pressure algorithms of Section 4.3.1.2 when applied to 6LoWPAN networks adopting RPL [76] and transmitting data packets over unidirectional and upstream flows, i.e., from some source sensor nodes to the border router which interconnects the constrained sensor network to the unconstrained Internet, see the first case study of Fig. 4.4.

4.3.2.1 System Parameters

The following parameters have been chosen to evaluate their impact on the performance.

Offered Traffic Load (λ_{tx}) defines the rate at which each source emits UDP

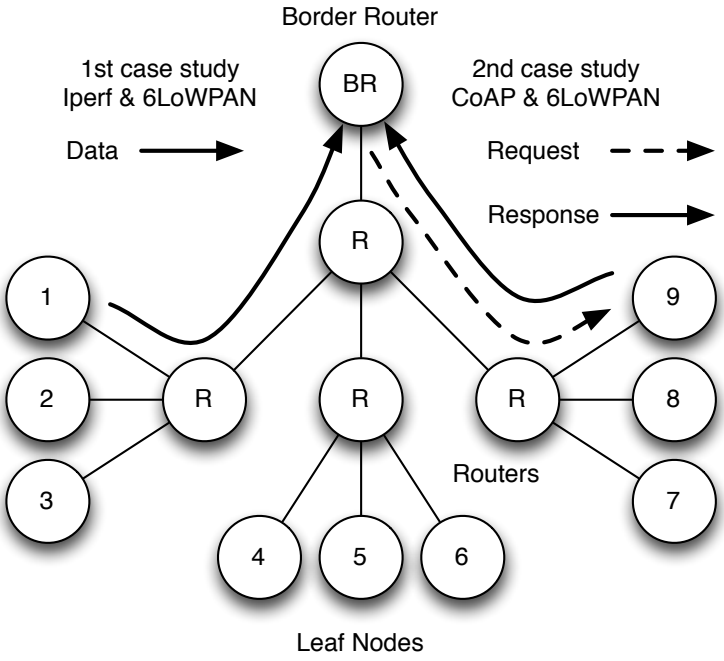

Figure 4.4: Network topology

datagrams at layer-3 and is measured in packets per second.

Number of retransmissions (N_{retx}) controls the maximum number of retransmissions. Specifically, when $N_{\text{retx}} = 0$ retransmissions are disabled at both layer-2 and layer-3. When $N_{\text{retx}} = 1$ retransmissions are disabled at layer-3, whereas a maximum of 7 transmissions per packet is possible at layer-2. When $N_{\text{retx}} > 1$, the maximum number of allowed layer-2 transmissions is set to 7, and the maximum number of layer-3 retransmissions is set to $N_{\text{retx}} - 1$.

Maximum queue length (Q_{max}) defines the maximum available memory for the layer-3 queue, which is the same for every node and is expressed in terms of number of layer-3 packets.

Queue threshold (Q_{thr}) is a threshold on the queue length at the L3D receiver that is used to assess when a control action is required by the BP algorithms, as detailed in Section 4.3.1.2.

4.3.2.2 Performance Metrics

To compare the performance of the various L3Ds, the following metrics have been considered.

Reception rate (λ_{rx}) defines the average rate at which layer-3 packets are correctly received by the destination, and is measured in packets per second.

Multihop delay (D) refers to the time taken by a packet to be correctly received by the border router (BR, see Fig. 4.4) from its transmission instant at the source (one of the 9 leaf nodes of the routing tree of Fig. 4.4). D is the average delay, which is obtained averaging the packet delay over all packets that are correctly received by the border router.

Loss probability (P_{loss}) represents the probability that an emitted datagram is lost either due to buffer overrun or because the maximum number of retransmissions has been reached. P_{loss} is computed as a ratio of the total number of datagrams lost into the network to the total number of datagrams emitted by all sources.

Rejection rate (R) defines the average rate at which packets from the application are rejected by the network layer due to a full layer-3 queue. In this case, application layer packets are not lost at layer-3 but their insertion into the layer-3 queue is denied and an error message is propagated toward the application layer. Upon receiving this error message, the application slows down its transmission rate, temporarily stopping its transmission flow and resuming it whenever new buffer space becomes available at layer-3.

Transmission Overhead (N_{tx}) represents the average total number of layer-2 packets that are transmitted in the network for the successful end-to-end (from a leaf node to the border router) delivery of a single layer-3 datagram. This metric accounts for the number of layer-2 packets that are sent to carry layer-3 data messages as well as layer-3 BP control messages, such as those sent by *Griping* and *Fuse* for the explicit signaling of a congestion event.

4.3.2.3 Results for Upstream Unidirectional Traffic

Simulation setup: Simulations have been run in ns-3 [90], using IEEE 802.15.4 at the PHY and MAC layers. At layer-2 packets have a fixed size of 127 bytes, including 12 bytes for the layer-2 headers. Layer-3 datagrams have a fixed size of 115 bytes, which means that a layer-3 datagram fits into a layer-2 packet and fragmentation is not needed. A tree topology has been built, see Fig. 4.4, containing 9 leaf nodes, 4 routers (R) and a border router (BR).

This topology is representative of a typical routing scenario for 6LoWPAN networks adopting RPL [76], a recently standardized routing protocol for low-power lossy networks.

In the following results, errors due to wireless transmissions have not been considered as neglecting channel impairments makes it possible to isolate and characterize the effects that are solely due to the considered congestion control algorithms, which is the main purpose of our study in this paper.

The simulation duration is set to 200 seconds, and $Q_{\max}$ is set to 31 packets to resemble typical limitations of IPv6/6LoWPAN stacks on constrained hardware platforms (see [8]). Each of the points in the following graphs is obtained averaging 21 independent simulation runs.

Impact of varying the transmission rate λ_{tx}: In Fig. 4.5 we show λ_{rx} and D for each L3D device as a function of $\lambda_{\text{tx}} \in [1, 30]$ pkt/s. The remaining system parameters have been set to: $N_{\text{retx}} = 15$, $Q_{\max} = 31$ packets, and $Q_{\text{thr}} = 15$ packets.

Considering the *Static* device, as long as the input rate λ_{tx} remains smaller than a certain *saturation threshold* λ_{sat} (of about 9 pkt/s in Fig. 4.5), we have that the reception rate $\lambda_{\text{rx}}^{\text{Static}}$ is approximately equal to $\lambda_{\text{tx}}^{\text{Static}}$ and D^{Static} is stable and small. This means that the network can effectively serve the injected data traffic. Here, the packet delay is typically dominated by the transmission and propagation delays over the involved multiple-hop paths from the sources to the BR, whereas the queueing delay is negligible. As $\lambda_{\text{tx}}^{\text{Static}}$ grows larger than λ_{sat}, $\lambda_{\text{rx}}^{\text{Static}}$ saturates reaching the so called *saturation throughput*. At this point, D^{Static} grows abruptly and this is due to the queueing component of the delay, that considerably increases as λ_{tx} becomes higher than the actual layer-2 service rate.

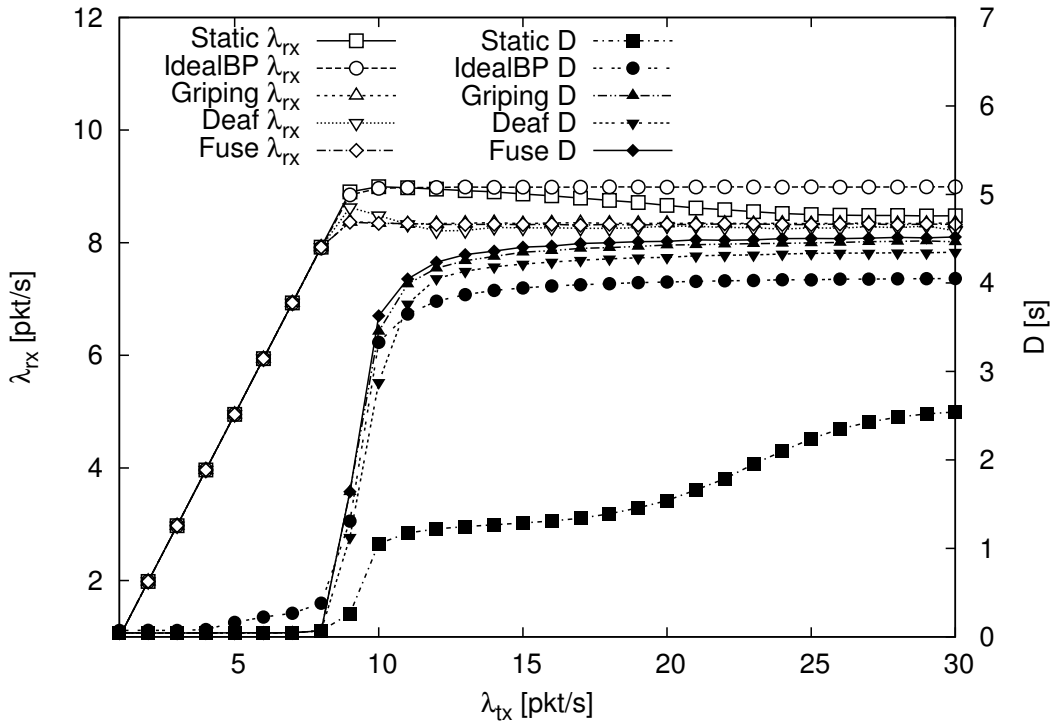

Figure 4.5: Reception rate λ_{rx} and multihop delay D *vs.* the offered traffic load λ_{tx}.

Similar performance tradeoffs are observed for all L3Ds.

While there are no substantial differences between *Static* and the other L3Ds in terms of λ_{rx}, we note that all the other devices obtain an average delay D increased by a factor of about 4 during congestion if compared with *Static*. This is due to the fact that these devices put off the transmission of new layer-3 packets when the network is congested, whereas *Static* keeps transmitting at a fixed rate, irrespective of the congestion status of the network. Also, *Griping* and *Fuse* account for the longest delay, and this is due to their explicit transmission of back pressure messages. From this first figure we observe that back pressure tends to increase the delay but is able to retain most of the throughput performance of the greedy *Static* transmission policy.

The rejection rate R has been plotted in Fig. 4.6 for the same simulation parameters. For BP devices, flow congestion actions are taken as soon as λ_{tx} becomes equal to λ_{sat} and are enforced as long as $\lambda_{\mathrm{tx}} \geq \lambda_{\mathrm{sat}}$. These actions

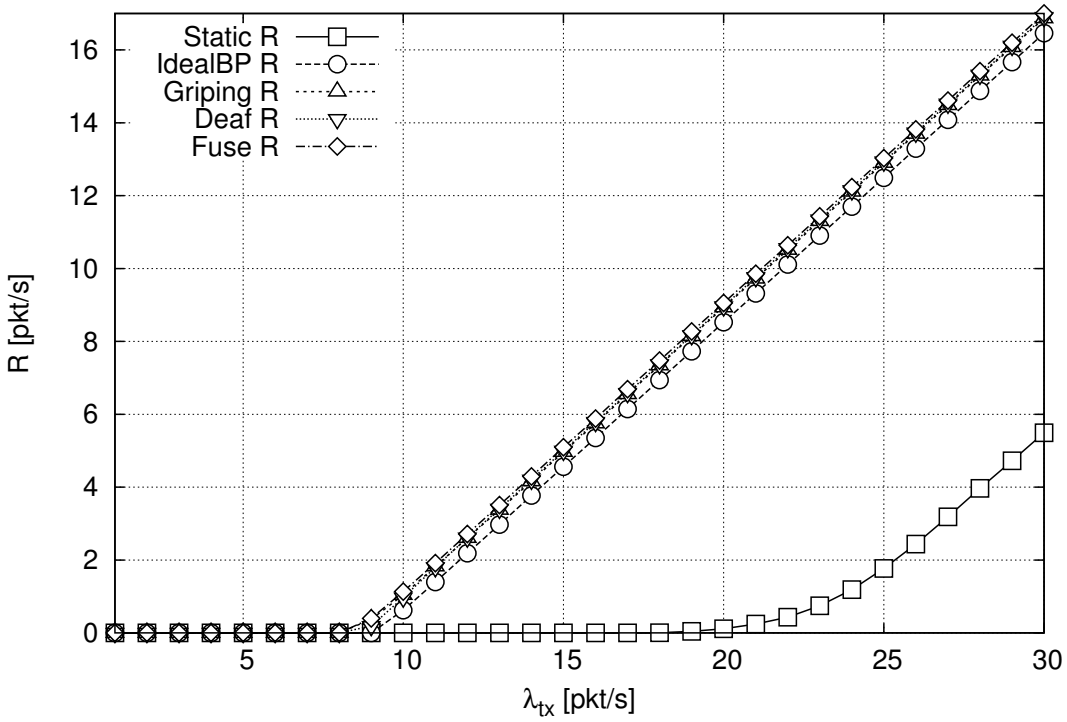

Figure 4.6: Rejection rate R *vs.* the offered traffic load λ_{tx}.

correspond to increasing the layer-3 rejection rate R. We note that *IdealBP* has the lowest rejection rate and the highest reception rate among all BP schemes, and thus, as expected, it is the best performing algorithm, i.e., the one able to fully exploit the benefits of back pressure.

R of *Deaf*, *Griping* and *Fuse* is very similar and close to that of *IdealBP*. Moreover, their back pressure policy becomes effective when $\lambda_{\text{tx}} \geq \lambda_{\text{sat}}$, which is testified by the prompt increase in R when the network operates beyond the saturation point.

Static keeps sending packets at the maximum possible rate, irrespective of the queue status at the relays. This moves to the right the value of λ for which layer-3 queues are filled up and packets start to be rejected (the increase of R becomes apparent for $\lambda_{\text{tx}} \geq 20$ pkt/s in Fig. 4.6). However, as we shall see below the drawback of this aggressive transmission behavior is that layer-3 queues are subject to higher loss rates.

As we show shortly, *Griping*, *Deaf* and *Fuse* have a substantially smaller

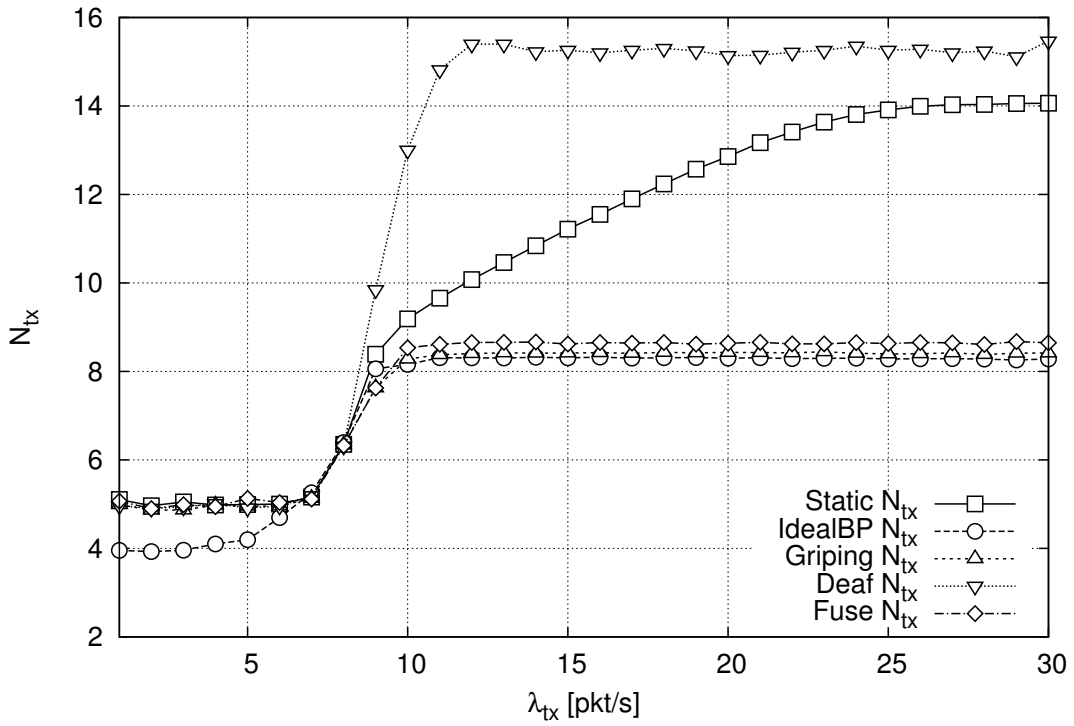

Figure 4.7: Transmission overhead N_{tx} *vs.* the offered traffic load λ_{tx}.

P_{loss} than *Static* as they reject only the data traffic that the network cannot sustain, mimicking *IdealBP*'s behavior. Note that layer-3 rejection does not imply discarding packets but rather slowing down the packet generation rate at the application.[1]

In Fig. 4.7 we show the transmission overhead N_{tx} as a function of the offered traffic load λ_{tx}. As expected, *IdealBP* has the best performance among all schemes as it applies BP control by leveraging the exact and instantaneous knowledge of all network queues, which is provided in the simulations through a genie. As λ_{tx} increases beyond λ_{sat} all the remaining schemes show a degraded performance in terms of N_{tx}. *Deaf* is the scheme that leads to the highest transmission overhead and this is inherent in its design, as this scheme tends to hit the maximum number of retransmission attempts while handling congestion control.

[1]Rejecting traffic that cannot be successfully handled by the network results in improved performance for all users. This can be supported with minimal impact by those applications featuring elastic data traffic. Otherwise, the application will see some degraded performance.

Static is the second-worst as in this case congestion is emphasized through the careless injection of data traffic. *Griping* and both perform very close to *IdealBP* as the corresponding BP policies explicitly send congestion notifications to the senders and this has the effect of timely slowing down the volume of data that is injected into the network, alleviating the congestion.

Impact of varying N_{retx}: Fig. 4.8 shows the loss probability P_{loss} as a function of N_{retx}. The remaining system parameters have been set to: $\lambda_{\text{tx}} = 20$ pkt/s, $Q_{\max} = 31$ packets, and $Q_{\text{thr}} = 15$ packets. Note that a transmission rate $\lambda_{\text{tx}} > \lambda_{\text{sat}}$ has been chosen so as to measure the ability of the different L3Ds to handle network congestion.

From Fig. 4.8 we observe the expected result that P_{loss} generally decreases as N_{retx} grows. This decrease is faster for *Griping*, *Fuse* and *IdealBP* as these algorithms use explicit signaling to detect congestion. The initial P_{loss} decrease is slower for *Deaf* which therefore shows worse P_{loss} performance for small values of N_{retx}, say, $N_{\text{retx}} \leq 7$. As expected, *Static* has the worst reliability performance as retransmissions are disabled for this scheme.

Also, *Griping* has a floor at $P_{\text{loss}} \simeq 0.02$, which is due to the inherent delay incurred in the explicit BP notification from the relay nodes. In fact, between the instant when a BP message is issued by a relay node and the instant when the controller at the corresponding source node enforces some back pressure action, the transmission rate remains equal to the one that has caused the congestion and, in turn, layer-3 losses are possible at the receiver node due to the overflow of its buffer. Thus, a *vulnerable period* exists between the instant when congestion is detected at the relays (that is, when their queue size increases beyond Q_{thr}) and the instant when the layer-3 flow is effectively slowed down at the sources. During this vulnerable period, losses due to buffer overflows are likely to occur. For *Deaf*, losses are still present due to the exhaustion of the overall number of retransmissions per packet per hop (both layer-2 and layer-3) and for this reason its P_{loss} monotonically decreases with an increasing N_{retx}.

Fuse has the best P_{loss} performance and the reason for this is the combined effect of *Griping* and *Deaf*. In particular, the explicit signaling of *Griping* allows for a prompter reaction to congestion events, which substantially decreases the probability of *Fuse* reaching the maximum number of retransmissions. More-

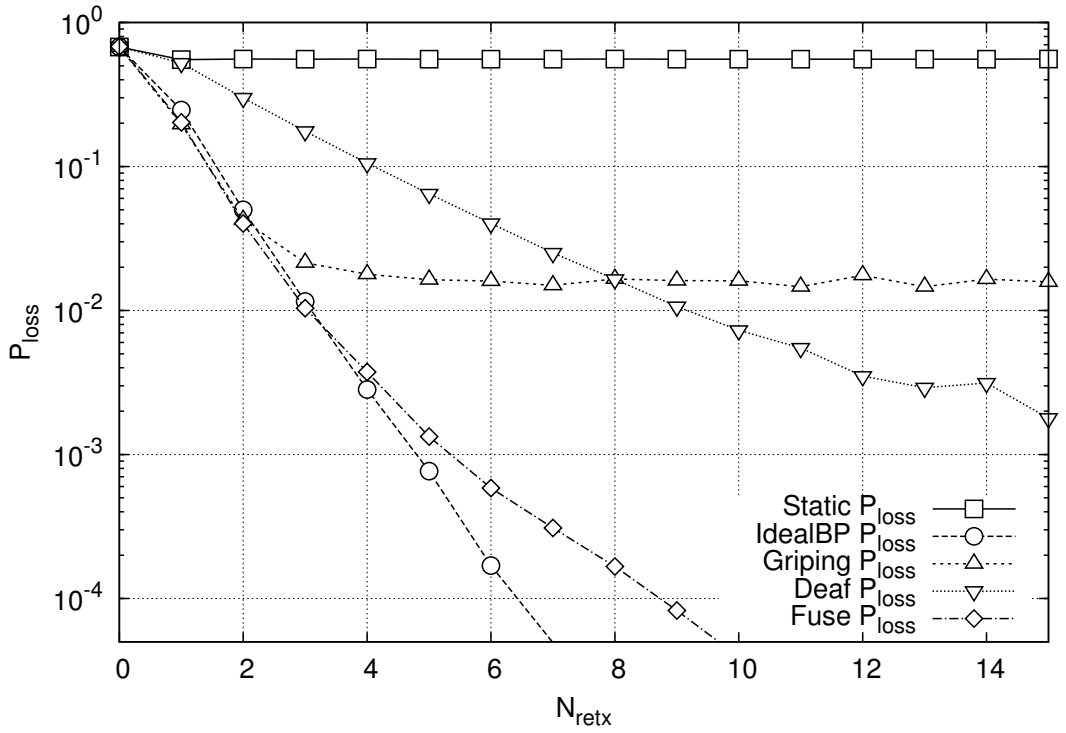

Figure 4.8: Loss probability P_{loss} *vs.* the number of retransmissions N_{retx}.

over, the vulnerable period issue is solved as, whenever the receiver's queue is filled up, *Deaf*'s BP control is invoked and packets that overflow from this queue are subsequently retransmitted by the corresponding sender (due to the stopped acknowledgement flow).

For what concerns previously shown performance metrics, all of them stabilize for small values of N_{retx} to the values shown in Figs. 4.5, 4.6, and 4.7.

Furthermore, when $N_{\text{retx}} = 0$, network congestion goes undetected and back pressure algorithms are never activated. In fact, in this case packets are transmitted but never retained in local queues, which are therefore filled up at a much slower pace. Thus, increasing N_{retx} allows the fill-up of layer-3 queues and, in turn, the detection of congestion events: in fact, the rejection rate R is positive for $N_{\text{retx}} > 0$. $N_{\text{retx}} = 0$ leads to poor performance on all metrics for all BP schemes.

Even $N_{\text{retx}} = 1$ leads to substantial throughput improvements in terms of λ_{rx}. Setting $N_{\text{retx}} = 2$, which means up to 7 layer-2 and just 1 layer-3 retransmis-

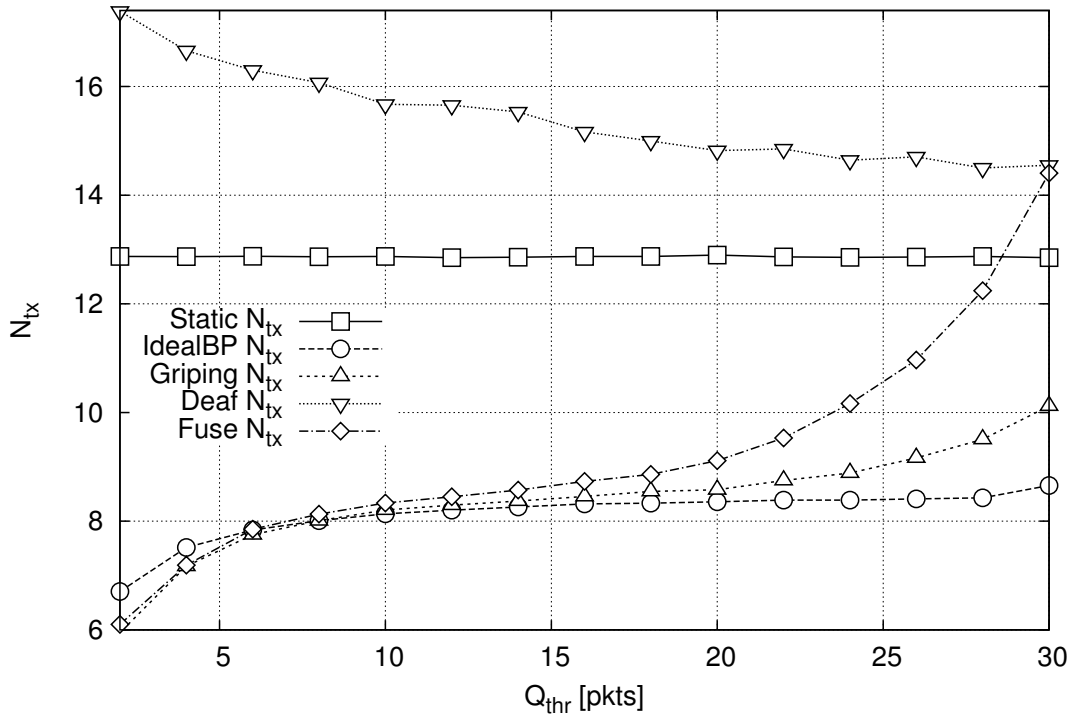

Figure 4.9: Transmission overhead N_{tx} *vs.* the queue threshold Q_{thr}.

sion, grants a throughput that is very close to the maximum achievable for the given network setup. The throughput increase is always accompanied by a corresponding increase in the delay performance, which also stabilizes for small values of N_{retx}. Counterintuitively, N_{tx} remains stable for all BP devices when $N_{\text{retx}} > 1$.

Impact of varying Q_{thr}: Fig. 4.9 shows the impact of Q_{thr} for $N_{\text{retx}} = 15$, $\lambda_{\text{tx}} = 20$ pkt/s, $Q_{\max} = 31$ packets. Here, we only show the plot for N_{tx}, the other metrics are just discussed as their behavior is similar to what observed above. *Static* is represented as a horizontal line in the plot, since its behavior does not depend upon Q_{thr}.

As expected, λ_{rx} grows for increasing Q_{thr}, as a larger threshold lowers the probability of having buffer under-runs, thus leading to higher throughput efficiencies. For the average delay D, an increasing Q_{thr} puts off the enforcement of back pressure control actions. Correspondingly, the number of packets stored in layer-3 queues and their average delay both increase. On the other hand, very

low values of Q_{thr} lead to long delays too as in this case back pressure control is almost always active, i.e., transmission rates are often slowed down and this implies longer L3D service times.

As expected, R decreases monotonically with Q_{thr} for all BP schemes as the rate of back pressure control actions is lowered for increasing values of Q_{thr}.

The behavior of P_{loss} differs among the considered layer-3 devices. These results are just commented but not plotted for the sake of space. *IdealBP* shows no losses at all as its control action is deterministic and immediate. Lowering Q_{thr} for *Griping* implies an earlier enforcement of back pressure policies, which leads to a larger buffer space to compensate for incoming packets during its vulnerable period. Thus, an increasing Q_{thr} implies larger buffer overflow probabilities (i.e., larger P_{loss}). Conversely, for *Deaf*, P_{loss} decreases with increasing Q_{thr}, ranging between 0.2% and 0.01%. This is because using the *Deaf* device packet losses only occur whenever a source reaches the maximum number of allowed retransmissions for a datagram (see C2 above). This event for *Deaf* is more likely to occur when Q_{thr} is small, because in this case BP congestion control is activated more often, which means that layer-2 packets are acknowledged less frequently and this leads to more layer-2 failures. *Fuse* has the best performance among all L3Ds with a P_{loss} that is smaller than 10^{-4} for $Q_{\text{thr}} \leq 26$, whereas its P_{loss} converges to that of *Deaf* as Q_{thr} approaches $Q_{\max}$. This good performance is due to the combined effect of *Griping* and *Deaf* BP control.

Finally, from Fig. 4.9 we observe that *IdealBP* has the best overhead performance, whereas *Deaf* has the worst. *Griping* performs very close to *IdealBP* for all values of Q_{thr}. *Fuse* performs very close to both *IdealBP* and *Griping* for Q_{thr} smaller than, say, $Q_{\max}/2$, whereas as Q_{thr} increases toward $Q_{\max}$ the transmission overhead of *Fuse* converges to that of *Deaf*. This is representative of the fact that the overhead in the latter case is dominated by the retransmissions due to the stopped acknowledgment flow.

4.3.3 Back Pressure Congestion Control for CoAP

Current trends in IoT networking involve the use of web services on constrained IoT devices. These entail the bi-directional exchange of messages according to a request/response paradigm, see Fig. 4.4, as per the REST architectural style [33].

The Constrained Application Protocol (CoAP) [10] defines a simple, efficient, and flexible protocol to allow REST architectures to scale down to smart objects, by preserving interoperability with HTTP [12].

In this section, we apply back pressure congestion control to bidirectional CoAP traffic. In this case, typical communication patterns amount to the transmission of CoAP control messages from the outside Internet network to the constrained IoT nodes and their subsequent CoAP responses. CoAP makes it possible for IoT resources to be accessible as web services, and in particular makes them available on the Internet as HTTP web services (through CoAP to/from HTTP mapping, see, e.g., [12]).

Referring to the second case study of Fig. 4.4, CoAP requests are sent over the constrained network as IPv6 datagrams flowing from the 6LoWPAN border router down to the leaf nodes. Upon receiving these requests, leaf nodes reply with IPv6 datagrams carrying the corresponding CoAP responses; the latter datagrams flow from the leaf nodes to the border router.

Note that the congestion problem is only marginally handled by the CoAP specification, which recommends a fixed congestion window of 1 packet at the CoAP senders. However, this static window may result in underutilized transmission resources when the network has some residual transport capacity and is as well inefficient when even this small window value suffices to create congestion. Differently, our approach is to prevent network queues from overrunning and as well to avoid the injection by the border router of an excessive number of requests into the constrained network. The latter objective is accomplished at the border router through the rejection of requests coming from the external Internet network using a "503 Service Unavailable" error response, which signals to the requesting client that the wanted resource is temporarily unavailable. This combined control is the purpose of our study in the following.

4.3.3.1 L3 Devices for Bidirectional Back Pressure

Differently from traditional BP, when bidirectional traffic is taken into account some additional mechanisms need to be added to the congestion control policies. In fact, BP should not slow down response traffic, because any dropped CoAP response would mean a network loss from the client's perspective.

Taking into account the fact that every CoAP request solicits a CoAP re-

sponse flowing along the reverse path, the length of the network queues is still a valid measure of network congestion. However, this measure alone is not entirely representative of the number of outstanding CoAP requests that are still waiting for a corresponding CoAP response message. Note that these responses may still cause buffer overruns as they are transmitted over the constrained network. Ideally, one would need to track the number of outstanding CoAP requests, so as to gauge the expected future load due to CoAP responses and shape the data traffic accordingly. However, such a task is generally too complex for resource constrained IoT devices. Aiming at a lightweight design, in the following we modify the L3Ds of Section 4.3.1.2 with the objective of pushing back CoAP requests only based on the queue length metric alone. While suboptimal, this solution entails little changes on current CoAP stacks and incurs low communication overhead. The goal of this section is to check whether, in spite of its simplicity, our queue-length-based control can provide satisfactory performance and also check which are the most important parameters that have to be tuned for its successful utilization.

Henceforth, the L3D devices of Section 4.3.1.2 have been modified as follows:

- **Static** this device does not apply any congestion control algorithm and is unchanged with respect to that of Section 4.3.1.2.
- **IdealBP** only applies its queue-length-based differential BP to the CoAP request traffic flowing from the border router to the leaf IoT nodes.
- **Griping** emits its explicit back pressure messages only upon the reception of CoAP requests.
- **Deaf** implements the backoff policy of Section 4.3.1.2 and refrains from transmitting layer-2 acknowledgements when the corresponding network layer datagrams are CoAP requests.
- **Fuse** extends the *Fuse* BP policy of Section 4.3.1.2 adding a further threshold Q_{thr2} such that $Q_{\text{thr}} < Q_{\text{thr2}} < Q_{\text{max}}$. Hence, the *Deaf* BP policy is activated when the queue length grows beyond the new threshold Q_{thr2}, whereas the behavior of the *Griping* BP policy remains unchanged. This second threshold allows the activation of BP congestion before the layer-3

queue is filled with packets and this leaves some room to accommodate the CoAP reverse traffic.[1] As for *Griping* and *Deaf*, BP is only applied to CoAP requests.

For the border router, whenever its layer-3 queue becomes full, it rejects any further incoming CoAP request by issuing a "503 Service Unavailable" error message.

4.3.3.2 System Parameters

Offered request load (λ_{tx}) defines the rate at which each CoAP client, placed in the external Internet network, sends CoAP requests toward a specific CoAP server placed within the constrained IoT network. Note that a server corresponds to an IoT leaf node in our simulation scenario, see Fig. 4.4.

The definition of the remaining system parameters Q_{thr}, Q_{max}, N_{retx} remains the same as that of Section 4.3.2.1, the new threshold Q_{thr2} has been set to $Q_{\text{thr}}+5$ packets.

4.3.3.3 Performance Metrics

To compare the performance of the proposed L3Ds, the following performance metrics have been considered.

Received response rate (λ_{rx}) defines the average per server (running on a leaf node) rate of CoAP responses that are correctly received by the border router and is measured in correctly received CoAP responses per second per server.

Round trip-time (D) defines the average lapse of time (seconds) spent at the border router waiting for a CoAP response to an accepted CoAP request.

Loss probability (P_{loss}) defines the percentage of CoAP responses that are not received by the border router, although the corresponding CoAP requests have

[1] In other terms, the difference $Q_{\text{max}} - Q_{\text{thr2}}$ is our best *a priori* estimate of the impact of the CoAP responses that will follow the CoAP requests that are currently admitted in the network.

been accepted into the constrained network.

Rate of rejects (R) defines the average per client rate of CoAP requests that are not accepted into the network by the border router (issuing an HTTP 503 status code, as per our discussion above) and is measured in terms of rejected CoAP requests per second per client.

Transmission Overhead (N_{tx}) represents the average number of layer-2 packets that are transmitted in the network for the successful end-to-end bidirectional exchange (from the border router to a leaf node and back to the border router) of a single CoAP request and response pair. This metrics accounts for the layer-2 packets that are sent to carry CoAP requests and responses as well as layer-3 BP control messages, such as those sent by *Griping* and *Fuse* for the explicit signaling of a congestion event.

4.3.3.4 Results for Bidirectional CoAP Traffic

Simulation setup: simulations have been run over the topology of Fig. 4.4, where a CoAP server has been deployed on each of the 9 leaf nodes; the border router hosts a CoAP proxy, which accepts CoAP requests from 9 CoAP clients placed in the external Internet network. Each CoAP client emits Non-Confirmable (NON)[1] requests at a constant rate λ_{tx} toward a CoAP server running on a leaf node. CoAP requests and responses have a fixed layer-3 size of 12 bytes and 115 bytes, respectively, including 6LoWPAN/UDP headers. The duration for each simulation run is 500 seconds, and the queue size of all nodes is 31 packets. The simulation points on the following graphs have been obtained averaging over 21 independent simulation runs.

Impact of varying λ_{tx}: as a first result, Fig. 4.10 shows λ_{rx} and D as a function of $\lambda_{\text{tx}} \in \{1, \ldots, 15\}$ req/client/s. The remaining simulation parameters are $Q_{\max} = 31$ packets, $Q_{\text{thr}} = 20$ packets, $Q_{\text{thr2}} = 25$ packets and $N_{\text{retx}} = 15$.

For Fig. 4.10, we note that the general behavior of all metrics is similar to

[1]NON requests do not have application-layer retransmissions; we chose to use this kind of requests, since our objective is the layer-3 evaluation of the congestion control performance.

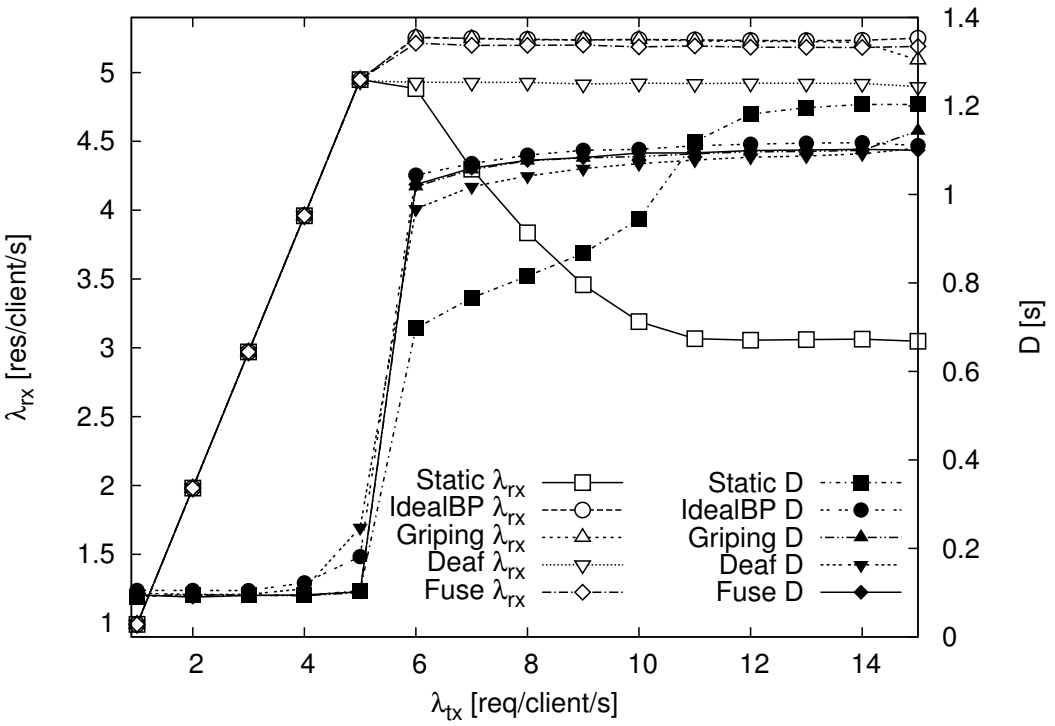

Figure 4.10: Received response rate λ_{rx}, and round-trip time D *vs.* the offered request load λ_{tx}.

that observed for unidirectional traffic, see Fig. 4.5. The main difference is that in this case λ_{sat} is nearly halved due to the presence of CoAP bidirectional exchanges, whereby two packets (CoAP request and response) must be handled by the network for each accepted CoAP request. In fact, although CoAP requests and responses differ in size, their cost in terms of overall time spent, including retransmissions, is nearly the same and this is due to the dominating effect of MAC layer tasks such as the time required to gain access to the channel, back off times, etc., which do not depend on the data frame size.

Also, we note that $\lambda_{\text{rx}}^{\text{Static}}$ equals $\lambda_{\text{tx}}^{\text{Static}}$ up to about $\lambda_{\text{sat}} = 5$ req/client/s, beyond which the response rate starts decreasing to a floor of about 3 req/client/s. This behavior is due to the so called *congestion collapse event*, similar to that observed in the early days of the Internet (see [83, 91]). The congestion collapse is caused by the border router accepting more requests than those that can be served by the network, which is given by $\lambda_{\text{sat}}^{\text{Static}}$. For the delay, we note that D

grows until λ_{rx} stabilizes.

Notably, *IdealBP*, *Griping*, *Deaf* and *Fuse* are not subject to the congestion collapse of *Static* but their throughput performance stabilizes as soon as λ_{tx} increases beyond λ_{sat}. Moreover, their delay remains stable even with λ_{tx} larger than λ_{sat}. This occurs because the border router acts as a proxy by rejecting traffic as soon as its outbound queue toward the constrained network becomes full.

For the rejection rate R, similarly to what observed for Fig. 4.6 (unidirectional flows), *Static* starts rejecting packets when $\lambda_{\text{rx}}^{\text{Static}}$ is approximately 10 req/client/s, which is about twice λ_{sat}. The remaining L3Ds react to an increasing λ_{tx} by rejecting packets as soon as the offered traffic increases beyond λ_{sat}, with λ_{sat} halved with respect to that of Fig. 4.6. As for the unidirectional traffic scenario, *IdealBP* shows no losses, $P_{\text{loss}}^{\text{Griping}}$ converges to about 0.5%, $P_{\text{loss}}^{\text{Deaf}}$ and $P_{\text{loss}}^{\text{Fuse}}$ both stabilize around 10^{-5}.

Overall, it is worth noting that *Fuse* obtains nearly the same throughput as *Griping* but has the same P_{loss} performance as *Deaf*. This is due to the combined effect of *Griping*'s explicit signaling, which effectively limits the send rate within the network, and the fact that all requests are deterministically rejected by the border router when its queue length increases beyond Q_{thr2}, which helps preventing congestion events.

In Fig. 4.11 we look at the transmission overhead N_{tx}. As for the unidirectional traffic scenario, *IdealBP* shows a smaller transmission overhead than the other schemes for $\lambda_{\text{tx}} \leq \lambda_{\text{sat}}$. *Static* presents the highest N_{tx}, which increases for increasing λ_{tx} until it hits a maximum and this occurs at around $2\lambda_{\text{sat}}$, when the corresponding R starts increasing. The remaining back pressure schemes effectively limit the maximum overhead and in particular we note that *Deaf* performs quite well here, in contrast to its unsatisfactory overhead performance for unidirectional traffic. The reason for this is that in this case the border router is the only source of data traffic and sends its packets directly over the bottleneck link of the network. In this case, *Deaf*'s exponential backoff mechanism effectively keeps the overhead at a small value. This is in contrast to what happens for the unidirectional scenario where: *i*) there are multiple sources competing for the channel (multiple leaf nodes), *ii*) the considered tree topology is such that these multiple sources all insist onto the same routers and the data traffic is ultimately

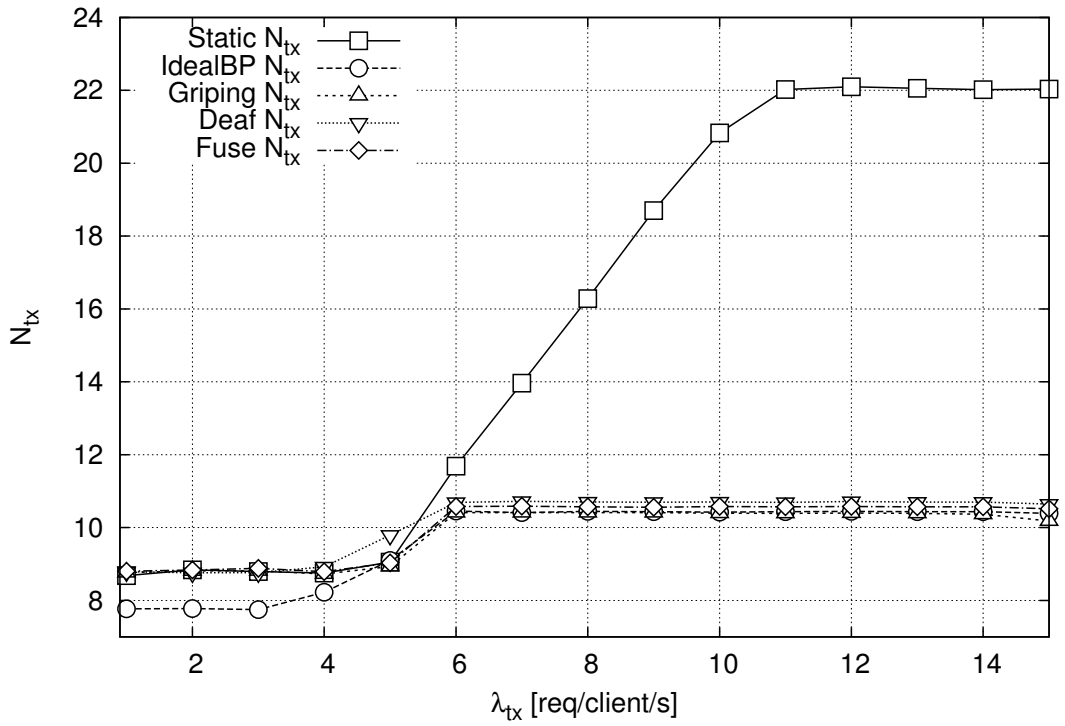

Figure 4.11: Transmission overhead N_{tx} *vs.* the offered request load λ_{tx}.

conveyed to a single border router (from many nodes to one), leading to an increasing congestion status as the data gets closer to the border router. Thus, in the unidirectional upstream case these facts result in a much more congested network and *Deaf*'s exponential backoff alone is ineffective.

The good performance of *Deaf* for bidirectional CoAP flows makes it suitable to add BP functionalities to current CoAP/6LoWPAN protocol stacks, without requiring the definition of further BP messages. In fact, this scheme in spite of its simplicity effectively avoids the network collapse and also leads to a reasonably small traffic overhead.

Impact of varying N_{retx}: Fig. 4.12 shows P_{loss} by varying N_{retx} in $\{0, 1, \ldots, 15\}$. The remaining simulation parameters are $\lambda_{\text{tx}} = 20$ req/client/s (the system is congested), $Q_{\max} = 31$ packets, $Q_{\text{thr}} = 20$ packets and $Q_{\text{thr2}} = 25$ packets.

As observed in Section 4.3.1, BP requires an adequate number of hop-by-hop retransmissions to work. In fact, *IdealBP* obtains no substantial advantage

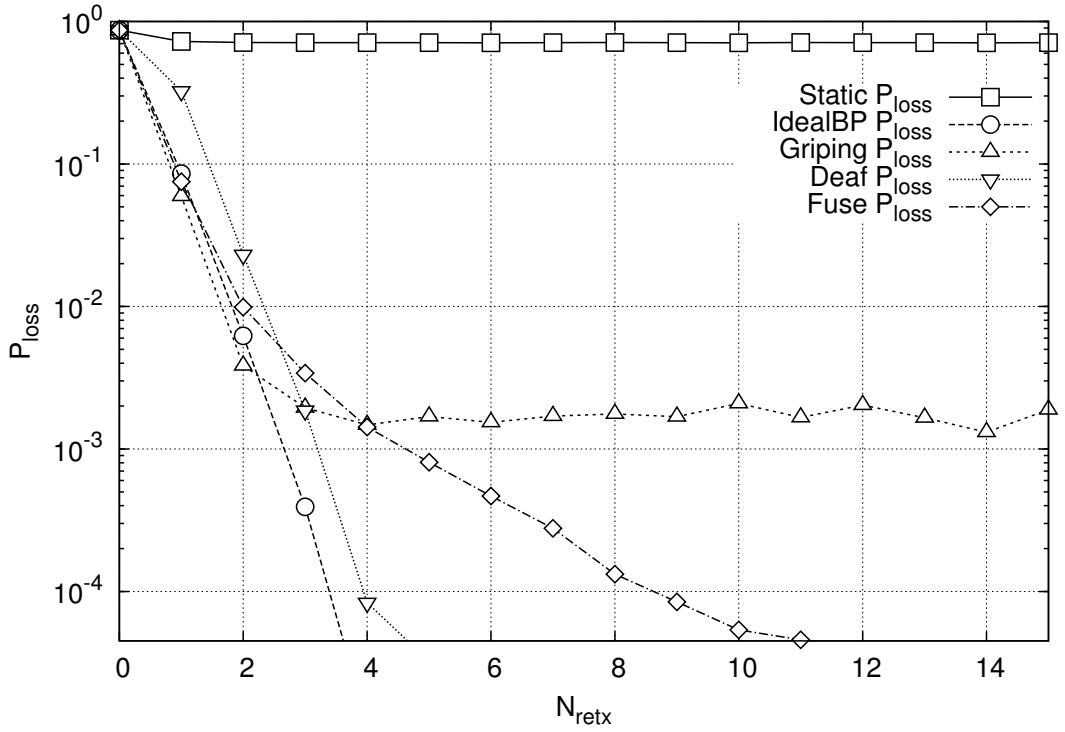

Figure 4.12: Loss probability P_{loss} *vs.* the number of retransmissions N_{retx}.

over *Static* when no retransmissions are allowed, whereas a very small number of retransmissions ($N_{\text{retx}} \leq 3$ for the considered setup) is sufficient for it to effectively relieve network congestion (see the sudden drop of $P_{\text{loss}}^{\text{IdealBP}}$ as N_{retx} grows).

Griping and *Deaf* require as well an adequate number of retransmissions to effectively work. $P_{\text{loss}}^{\text{Griping}}$ drops quickly and then stabilizes to a floor of about 0.02%. P_{loss} monotonically decreases for increasing N_{retx} for both *Deaf* and *Fuse*, although the latter requires a higher number of retransmissions due to the delayed BP control implied by the new threshold $Q_{\text{thr2}} > Q_{\text{thr}}$. As in the unidirectional scenario, a high number of retransmissions does not negatively impact N_{tx}, which remains stable for all devices with $N_{\text{retx}} > 3$.

Impact of varying $Q_{\max}$: memory requirements have a strong relevance for constrained devices. In particular, the available memory and its management limit the queue length in actual implementations, e.g., see [8].

Figs. 4.13 and 4.14 show λ_{rx} and P_{loss} by varying $Q_{\max}$ in $\{3, 6, \ldots, 60\}$. The

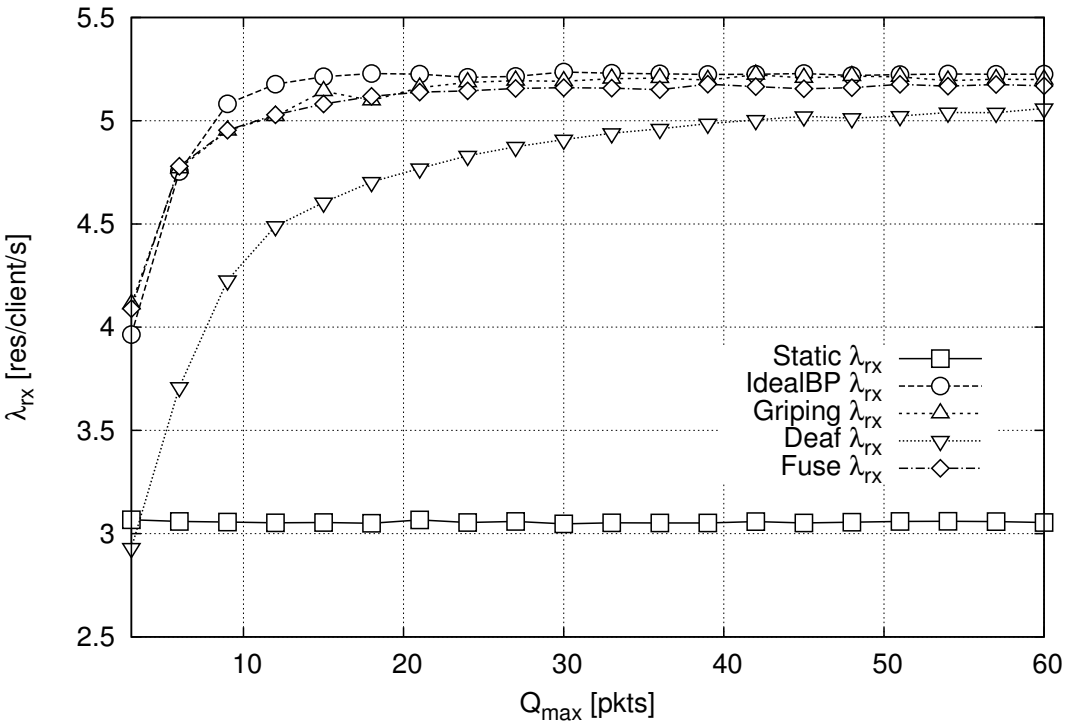

Figure 4.13: Received response rate λ_{rx} *vs.* the maximum queue length $Q_{\max}$.

remaining simulation parameters are: $Q_{\text{thr}} = \lceil (2/3)Q_{\max} \rceil$ packets, $Q_{\text{thr2}} = Q_{\text{thr}} +$ 5 packets, $N_{\text{retx}} = 15$ and $\lambda_{\text{tx}} = 20$ req/client/s.

The throughput λ_{rx} of *Static* remains stable around 3 res/client/s and is only marginally affected by $Q_{\max}$. For *IdealBP*, *Griping* and *Fuse*, λ_{rx} converges to about 5.2 res/client/s for $Q_{\max} \geq 15$ packets. Thus, besides improving reliability, BP control also makes it possible to roughly double the throughput performance. We also observe that *Deaf* has a throughput performance that is roughly from 5 to 10% worse than that of the other BP schemes. The reason of this is inherent in how *Deaf* reacts to congestion events. In fact, *Deaf* detects congestion by stopping the transmission of the layer-2 acknowledgments associated with layer-3 CoAP requests. This has the twofold effect of slowing down the send rate of CoAP requests, while occupying the channel with their retransmissions. However, these retransmissions prevent the senders from exploiting the channel for other useful traffic such as the transmission of CoAP responses, whose correct delivery would contribute to a higher throughput performance. The performance gap

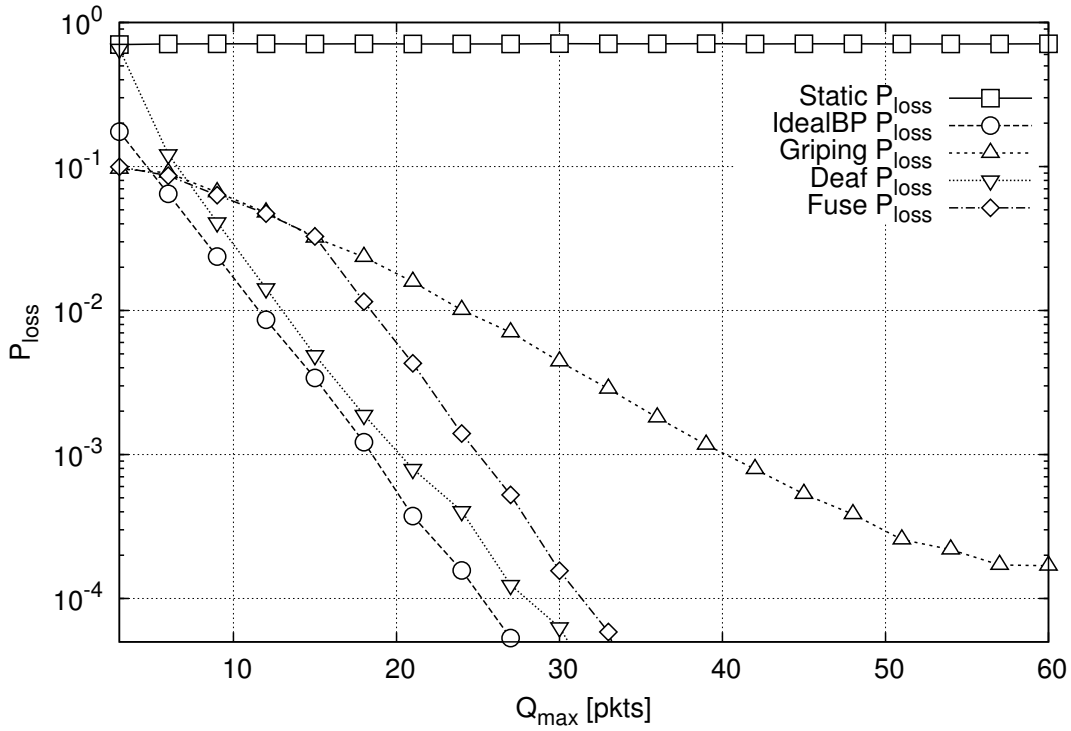

Figure 4.14: Loss probability P_{loss} *vs.* the maximum queue length $Q_{\max}$.

between *Fuse* and the other BP schemes decreases for increasing $Q_{\max}$, as Q_{thr} also increases, leading to a less frequent activation of BP control policies and of the just discussed inefficiencies in terms of channel utilization (waste of channel resources).

For P_{loss} from Fig. 4.14 we see that *Static* is unaffected by $Q_{\max}$, whereas the reliability performance improves for all other L3Ds for increasing $Q_{\max}$. This occurs because a larger $Q_{\max}$ implies that network queues have more room to absorb traffic bursts and this makes buffer overruns less likely to occur. We also observe that *Deaf* has a slightly smaller P_{loss} than *Fuse* as its smaller BP threshold $Q_{\text{thr}} < Q_{\text{thr2}}$ implies a prompter reaction to congestion events.

5

Application Protocols and Formats

Starting from the eighties, worldwide communications popularity has always been increasing. The Internet paradigm has become a common denominator for networking applications. Nowadays, many information and communications services rely on IP technology: web-shopping, online-databases, social networks being three notable examples.

Smart Grids (SG) [92, 93] and the Internet of Things (IoT) [94] are nowadays popular research topics in the ICT community. Even though born from different needs, they have quite a few aspects in common and are characterized by similar challenges. Key objectives for both scenarios are: **seamless integration with IP**, system **scalability** and **interoperability**.

Both scenarios comprise millions of heterogeneous embedded devices featuring different technologies, each developed to satisfy particular needs. Just to name a few, wireless sensor and actuator networks (WS&ANs) [63] adopt low–power radios and simple CPUs, Radio Frequency Identifiers (RFIDs) [95] and Near Field Communication (NFC) [96] rely on little computational power and very short range radios, whereas wired embedded devices are equipped with Power Line Communication (PLC) [97] and ARM CPUs. The seamless and scalable interworking of such diverse technologies is crucial to the success of SG and IoT. Also, communication protocols should scale to a large number of devices; this implies a clever design of the architecture together with efficient implementations

of communication paradigms.

1) **Seamless integration with IP:** this objective involves the provision of a seamless link between the Internet world and the machine–communications world.

2) **Scalability and interoperability:** both scenarios comprise millions of embedded devices characterized by numerous different hardware and communication technologies, to satisfy particular needs.

The Internet world is now in its mature age: the Internet Protocol (IP) is the most used network protocol along with the Hyper–Text Transmission Protocol (HTTP). Interconnecting IP and embedded systems/machines has always been a hot research topic, however, only recently did standardization bodies start to play a decisive role in this respect. Currently, many initiatives are competing to be the first to provide a feasible standard for SG/IoT.

First of all, we describe the Binary Web Service (BWS) protocol, developed within the SENSEI project [53], which introduced the concept of an efficient, binary and simple realization of web services for smart Internet-enabled objects; its pioneering development has heavily contributed to the subsequent formation of the CoRE charter in the IETF [71].

Last but not least, we illustrate some of the strengths of the Internet Engineering Task Force (IETF) approach. To this end, we detail the realization of simple, but powerful Web Services for IoT applications that use the Constrained Application Protocol (CoAP) [98], which is being defined in the Constrained RESTful Environment (CoRE) charter [71], and the eXtensible Markup Language (XML), which is combined with the Efficient XML Interchange (EXI) format [69].

5.1 Related Work

Recently standardization bodies started to play a decisive role in interconnecting constrained devices with the Internet. Currently, many initiatives are competing to be the first to provide a feasible standard for SG/IoT.

Recent research efforts explore the performance and the practical feasibility of a REST-based approach [4] on top of a 6LoWPAN stack [52] in WSNs;

Web services have to be suitable in complex installations and easily deployable while retaining the flexibility typical of IP-based protocols. Other recent papers [57, 58, 59] have already highlighted the benefits of lightweight Web-based protocols accessing sensors resource data through Uniform Resource Identifiers (URI) and request methods (GET, PUT, POST, DELETE) [4, 56, 63]; these concepts are the basis of Web Services and their introduction in WSNs environments is straightforward. However, Web Service-enabled WSNs still need a complete protocol stack definition for their direct integration in the Internet, proving that a Web-based system can smoothly bridge information, objects and new services through WSNs.

XML has been acknowledged as the *de facto* standard for data representation and exchange but its great flexibility comes at the price of being very redundant; to alleviate this, many solutions (see [99] for a thorough review of XML compression techniques) are available: blind compressors, such as gzip, bzip2, DTDPPM [100] treat XML as plain text files; a second group of compressors (e.g., enhanced XMill [101], XMLPPM [102]) takes the XML document structure into account to achieve higher compression ratios; which can be given using a separate source or can be obtained from the document itself. An exhaustive survey on XML compression techniques can be found in [99]. To the best of our knowledge, XBC and EXI have been the first working groups focusing on optimizing XML for constrained devices and the W3C [103] selected EXI as its standard.

5.2 Constrained Web Services

The Internet of Things (IoT) is constantly focusing higher efforts to build the technological foundation required to bring the Internet concept to anything, anytime and anywhere. Next-generation Web services are envisioned to be seamlessly connected to everyday objects, and to provide smarter interactions possible only thanks to the tight connection to the physical world. Wireless Sensor & Actuator Networks (WS&ANs) have been recognized as the technology required to bridge the Internet to the "things", tiny cheap objects that through specialization and expansion can perform a wide range of tasks, supporting battery-operation and wireless connectivity which are the key foundations to pervasive deployment.

Nevertheless practical realization of this vision is hard to achieve reusing es-

tablished standards and technology. Problem statement is two-fold: *i*) WS&AN nodes have severe constraints in terms of memory, processing speed, energy, transmission range and bandwidth, which set tight requirements in terms of architectural design and protocol selection, *ii*) the Web model is the core of the Internet and for this reason we want to transpose the current REST paradigm down to the IoT node.

In this section, we present the evolution of the definition of a REST communication protocol doable on constrained IoT devices, by initially introducing the original design of BinaryWS. As the results obtained with BWS were encouraging, this protocol has been carried to the IETF, where it is now at an advanced state of standardization and is proceeding towards becoming an Internet standard for application layer communications in the Internet of Things ecosystem [10].

5.2.1 Web-enabled Smart Objects

WS&AN has been a hot research topic for nearly 10 years now and can be considered mature. This is testified, e.g., by the huge number of ZigBee devices being shipped, which has been doubling every year, hitting 20 millions in 2009 [104]. The main characteristics of the sensor devices in a WS&AN are: **low–power radio**, providing wireless communication capabilities, but very little bandwidth, **low–power CPU**, enabling substantial energy savings in the face of little computational capabilities, and **small footprint**, allowing easy installation, but posing design constraints.

The combination of these characteristics has led to very economic devices, easy to install and able to work unattended for long periods of time before needing a battery replacement. These features make WS&AN one of the main actors in the IoT world. In fact, a sensor node can transform any appliance, any switch, any controllable machinery into a fully-connected object. In Figure 5.1 we show how a home environment can be instrumented with sensor nodes for smart metering and control of appliances. These features enabled the development of very economic devices, easy to install and showing long lifetime on batteries, thus making WS&AN one of the main actors in the IoT world. In fact, a sensor node can transform any appliance, any switch, any controllable machinery into a fully-connected object.

Figure 5.1: Internet of Things in domestic environments: appliances and utilities can be monitored and controlled in near-realtime using smart embedded systems with IP connectivity.

While this figure shows a domestic environment, where every electronic device is part of the home network, this concept can be extended to a larger number of scenarios, such as hospitals, offices, shopping malls, factories and even cities; thus realizing any sort of *smart*–environment. Adding Internet connectivity to these smart-environments, will make them interoperable and interconnected and a whole set of new services can be thought of. Finally, if every interconnected device can communicate using the same language(s), this will become the Internet of Things, where every single network element can be treated as a tiny Web-Server, characterized by a unique identifier and (potentially) providing information and services.

In order for the IoT to become a reality there are still many requirements to be satisfied: as already said, there is the need for common languages, but also of a common framework for the description of the information. In addition, these requirements, that are just the first two of a longer list, must be satisfied for the aforementioned constrained devices. In order to integrate these sensor devices with the Internet as seamlessly as possible, the most natural choice would be using the most popular protocols in the Web: HTTP-IP and XML. Moreover, it is possible to mutuate the concept of resources from the Internet: Web resources are information and service providers such as websites, but can be used to describe smart–devices too, because they provide the same functionalities.

Ideally, in the IoT each device will be represented as a resource providing its

own description in terms of hardware capabilities and software interfaces as well as a description of the services that it provides. For instance, the refrigerator will provide information about its description, such as its main physical function, its retailer, its operating status, but will also provide access to smart–services such as "best before"–notifiers when products in it are passing that date or an "out-of-stock"–notifier, informing the user about which products are needed. The user will thus be able to interact with the refrigerator in the same way as with any website, by just connecting to the appropriate (IP) address and modifying parameters or activating services.

5.2.2 Binary Web Services [7]

Motivation The structure of the a typical web service stack can be described as follows. TCP is initially used to set up a connection between client and server end-points. Then HTTP request messages are sent from the client to the server. Requests can make use of any HTTP method, typically GET, POST, PUT or DELETE. Finally, the client specifies the resource on the server to request in the form of a URL. The server responds with a response and a Code. HTTP is able to carry a body in either direction using any MIME type and encoding. The body of the HTTP web service message is typically XML, in a format known by the client and server. In order to perform sequences of Remote Procedure Calls (RPCs), the SOAP protocol may be used in the XML body, adding a layer of complexity. After the sequence of requests is complete the TCP connection is closed. The major problems limiting the use of HTTP in 6LoWPAN sensor networks include: *High Overhead* caused by plain text encoding and verbosity of the HTTP header format, *TCP Binding*, which seriously penalizes performance in ad-hoc wireless networks especially when connections are short-lived and *Complexity* because real HTTP servers, client and proxies use a large number of optional headers which can be very complex and often deliver unnecessary side informations.

Overview The Binary Web Service (BWS) protocol [7] is a binary, scaled-down realization of REST, but compatible with the verbose HTTP protocol [105]. BWS is based on UDP, enabling the REST interaction model on severely limited devices such as wireless sensors. To access a resource through BWS, a request message

is issued. In the 2-3 bytes long header of the message are specified the target resource (identified by a URL), the access method (GET, PUT, POST, etc.) and the format of the payload (Content-Type); the payload contains any data required to fully describe the request. A response describing the result of the request is sent by the receiving resource to the requesting entity, which is identified by its source (IP, port) pair; the header of the response contains the HTTP status code summarizing the result and the Content-Type (if any). All the aforementioned fields are encoded in binary form in a simple, short header; full support for URL strings is preserved but, alternatively, the target resource can be specified using a binary code.

BWS delegates payload data compression, and advocates the use of Efficient XML Interchange (EXI) from W3C [69] for Binary XML encoding. This choice is mainly due to the chance of operating the encoder in a schema-informed byte-aligned mode [69] which makes coding simpler and reduces the output size. However, we note that building the grammar and thus the specific implementation from a schema is quite complex; moreover building any generic EXI encoder is indeed difficult, but simple implementations can be derived only by analyzing specific schemas and by deriving the subset of features required by those schemas.

5.2.2.1 Implementation description

In our implementation, the communication with any resource is uniformed using common, flexible components (e.g., Binary XML Services), through a single-instance and resource-shared BWS implementation on top of the UDP/6LoWPAN stack. The same BWS module is designed to provide both server and client capabilities. It supports opening multiple servers, each able to serve parallel incoming requests from clients; as a client, it can handle concurrent communication with multiple different servers; therefore our implementation allows flexible services to be based on a single, reusable component. Both the client and the server entities provide support for URL requests or compressed URL requests, i.e., numerical identifiers (ID).

The BWS component provides server functionalities through the `BWSServer` interface, triggering an event for each incoming request to the registered resource.

```
interface BWSServer{
```

```
event error_t request(
  uint16_t rid,
  uint8_t id,
  uint8_t method,
  uint8_t content_type,
  uint8_t *content,
  uint16_t length );

event error_t requestURL(
  uint16_t rid,
  uint8_t *url,
  uint8_t method,
  uint8_t content_type,
  uint8_t *content,
  uint16_t length );

command void response(
  uint16_t rid,
  error_t status,
  uint8_t content_type,
  uint8_t *content,
  uint8_t length);   }
```

The interpretation of the method and content is left to the appropriate resource depending on the requested web service, which is identified by the server port and the URL or ID. Every request is identified by a 16 bits locally unique field (`rid`), in order to support triggering multiple requests to the same component. A web service can still be configured to handle one request at a time by refusing further inquiries using an appropriate status response; more advanced components can track multiple requests and independently respond to each. The BWS module keeps track of every active request, by mapping every `rid` to the requesting node IPv6 address and UDP port. Responses can be sent asynchronously issuing a `response` command to the BWS server and providing the `rid` matching the request being served.

Remote BWS servers can be accessed through a complemental interface (`BWSClient`), which provides methods to access BWS features. On a client, every request is mapped to a command which requires a service to the BWS component; the corresponding response is an event triggered by the BWS module. To support

clients sending concurrent requests, a 16-bit `rid` is associated to each request and returned to the resource as the result of the `request` or `requestURL` command. The response event is fired within the initiating component when the response message is received.

```
interface BWSClient{
  command uint16_t request(
    ip6_addr_t *addr,
    uint16_t port,
    uint8_t id,
    uint8_t method,
    uint8_t content_type,
    uint8_t *content,
    uint8_t length );

  command uint16_t requestURL(
    ip6_addr_t *addr,
    uint16_t port,
    uint8_t *url,
    uint8_t method,
    uint8_t content_type,
    uint8_t *content,
    uint8_t length );

  event void response(
    uint16_t rid,
    error_t status,
    uint8_t content,
    uint8_t *payload,
    uint8_t length);   }
```

The versatile design of the BWS component has been enabled by a proper modification of the `6lowpan` component interfaces. The `IP6P` component has been re-engineered to exploit the flexibility of parameterized interfaces as used in a similar manner by the `ActiveMessageC`; this allows the BWS module to take full control over the 6LoWPAN subsystem using the `UDP` interface described below. Note that the component `IP6P` parameterizes the interfaces depending on the local UDP port: for this reason, it does not appear among the function arguments. This modification allows easy code reuse for processing incoming packets or for writing outgoing packets to different UDP ports.

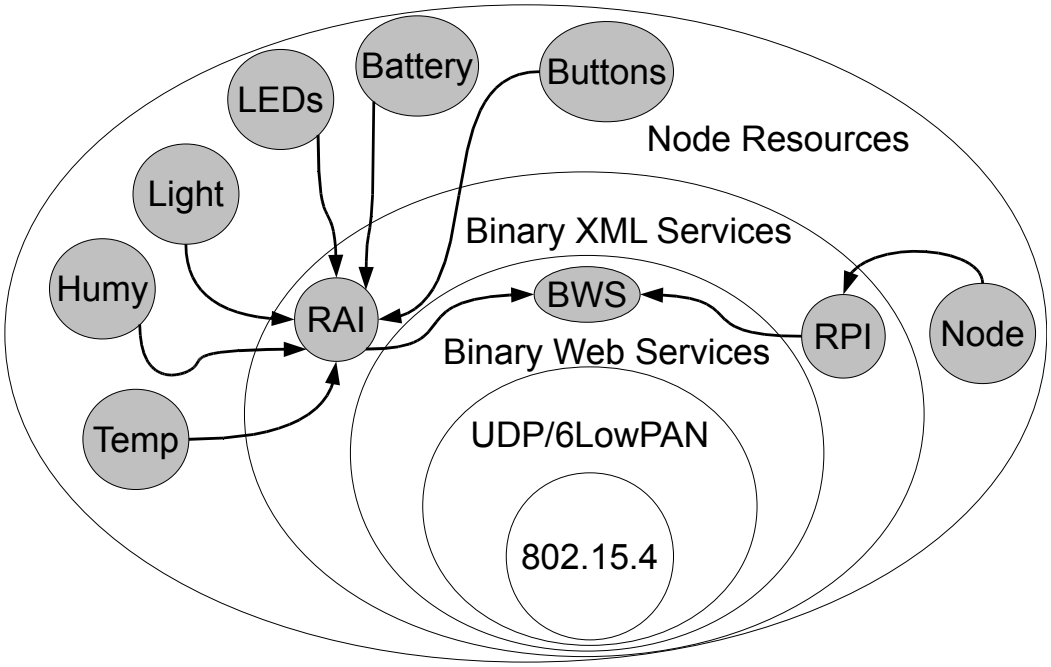

Figure 5.2: Components wiring in the SENSEI node.

```
interface UDP {
  command error_t sendTo(
    const ip6_addr_t *addr,
    uint16_t port,
    const uint8_t *buf,
    uint16_t len );
  event void sendDone(
    error_t result,
    void* buf);
  event void receive(
    const ip6_addr_t *addr,
    uint16_t port,
    uint8_t *buf,
    uint16_t len );
```

5.2.3 Constrained RESTful Environments (CoRE)

IETF formed this WG in 2010 with the main objective of defining the Constrained Application Protocol (CoAP), a RESTful protocol suitable for constrained environments. The REpresentational State Transfer (REST) paradigm refers to

designing APIs so that every data exchange can be made with the GET, POST, DELETE and UPDATE operations of the HTTP protocol.

The work of the CoRE WG has been chartered towards obtaining a web-oriented binary protocol, simple enough to be handled by severely limited devices, yet easy to map onto HTTP. The thrust behind the latter intent is driven by the pervasiveness of HTTP in the Web: enabling HTTP communication over constrained environments will further extend its applicability and will make it ubiquitous. The currently proposed protocol is trying to achieve this objective defining a binary representation of REST, which includes the most popular and useful features of HTTP.

Next-generation M2M environments are expected to be the killer application for this protocol: for instance, a lot of attention has been devoted to the design of publish/subscribe mechanisms, since this approach is deemed to be key for connecting limited devices and avoiding network congestion.

As many distributed content-generating networks, Smart Grids would experience various benefits from a web-like communication paradigm: in fact, web services are well-known in the traditional Internet for their applicability to almost every kind of application. Following this guideline, the WG is pushing the Constrained Application Protocol (CoAP) to be used for M2M communication, resulting in a web-compatible standard for M2M applicability.

By design CoAP is directly mappable to the current HTTP realization: by leveraging its intrinsic interoperability, the SG system design can be heavily simplified, by directly enabling each network device to interact with standard Internet languages and, at the same time, keeping the energy and traffic burden on the constrained environment low.

To give a brief example of CoAP's performance, an HTTP request with size in the order of some tens of bytes can be mapped into a CoAP request with the same functionalities in as few as 4 bytes. The latter occupation is more suitable for lower-layer frames in constrained environments. Moreover, CoAP's binary realization of REST eases the implementation of its paradigm on hardware with very limited computational power, as has been proven in the case study described in [11], where technical details on the design and realization of the complete protocol stack are given.

5.2.3.1 Constrained Application Protocol (CoAP) [10]

CoAP [98] is currently being defined within the CoRE [71] working group of the IETF, which aims at providing a REST–based framework for resource–oriented applications optimized for constrained IP networks and devices, by designing a protocol set able to cope with limited packet sizes, low-energy devices and unreliable channels.

CoAP is based on the REST architectural style sharing the objectives and the intrinsic limitation listed above. It is designed for easy stateless mapping with HTTP, and for providing M2M interaction. HTTP compatibility is obtained by maintaining the same interaction model, using a subset of the HTTP methods.

Nodes supporting CoAP provide flexible services over any IP network using UDP, and they also provide a solid communication framework to connect sensor nodes to the Internet. Any HTTP client or server can interoperate with CoAP–Ready endpoints by simply installing a translation proxy between the two devices. This will not be a burden for the proxy, since these translation operations have been designed not to be time and computationally demanding. Also, CoAP features a message layer between the application protocol and UDP to provide basic reliability and session matching support[1].

CoAP State Machines Even if CoAP has been defined to be implementable on constrained devices, designing such a CoAP implementation could be a complex error-prone time-consuming task. In fact, as [19] points out, implementors should take into account also infrequent events when designing their implementations. To this end, [19] aims at providing interesting examples of CoAP separate responses that are useful to aid CoAP implementers in understanding rare situations that may occur.

In fact, building a robust implementation of the CoAP state machine is complex on very constrained devices. This usually leads to simplify implementations by not accounting for unusual situations, apparently negligible. [19] describes rare message exchanges, with the intent of demonstrating possible risks for some simplified CoAP designs.

Figure 5.3 shows a first attempt to draw the CoAP state machine, assuming that it shall be implemented in a monolithic component design. The graph shows

[1]These functionalities are provided by the transport layer in the ISO/OSI stack.

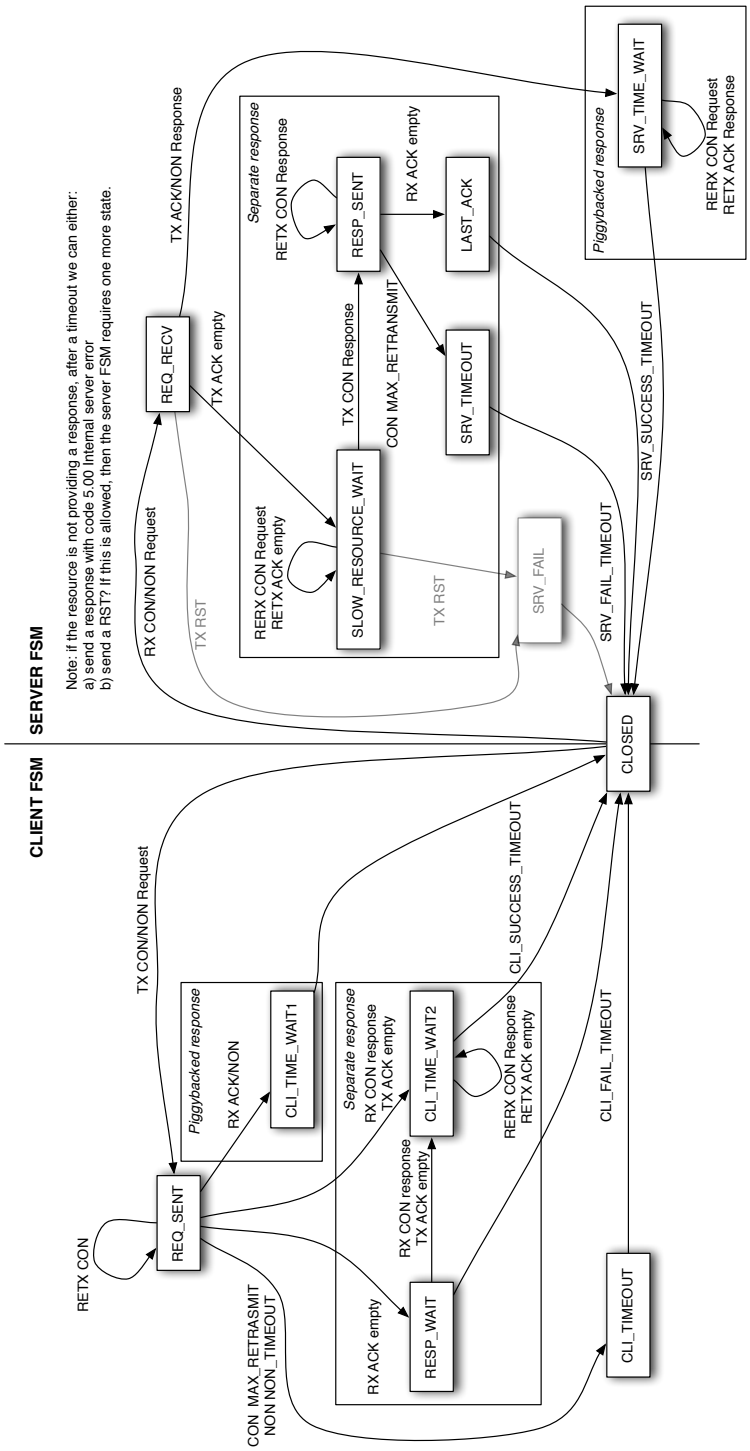

Figure 5.3: State machine for CoAP monolithic implementation

```
RETX_TIMEOUT = RESPONSE_TIMEOUT * [ 1, RESPONSE_RANDOM_FACTOR] * 2^(RETX-1)
 RETX_WINDOW = RESPONSE_TIMEOUT * RESPONSE_RANDOM_FACTOR * 2^(MAX_RETRANSMIT-1)
ACK0_TIMEOUT = RESPONSE_TIMEOUT * ACK0_TIMEOUT_REDUCTION (<1)
```

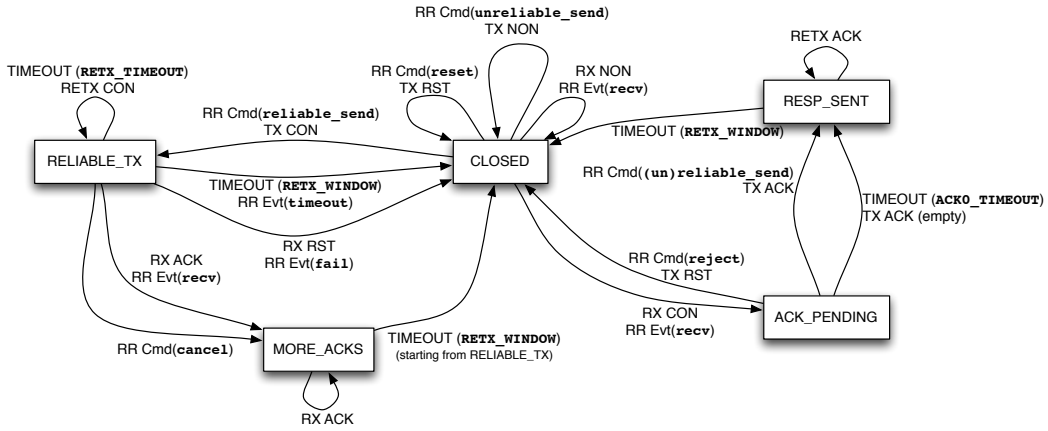

Figure 5.4: State machine for CoAP message-layer implementation

that the complexity required by such monolithic approach is high. In fact, to handle both message and request/response layers in a single software component, code complexity grows easily as soon as full compliance with the CoAP protocol is needed.

In Figures 5.4 and 5.5 we show a split design approach, where the message and request/response layers have been handled as different software components talking with each other through a specified software interface.

In particular, the required software interface to enable communication between the two components is as follows:

```
interface CoAPMessage {
  command error_t send (
    coap_session_t* session,
    uint8_t *data,
    uint16_t size,
    bool reliable );

  command void cancel (
    coap_session_t* session );

  event void recv (
    coap_session_t* session,
    uint8_t *data,
```

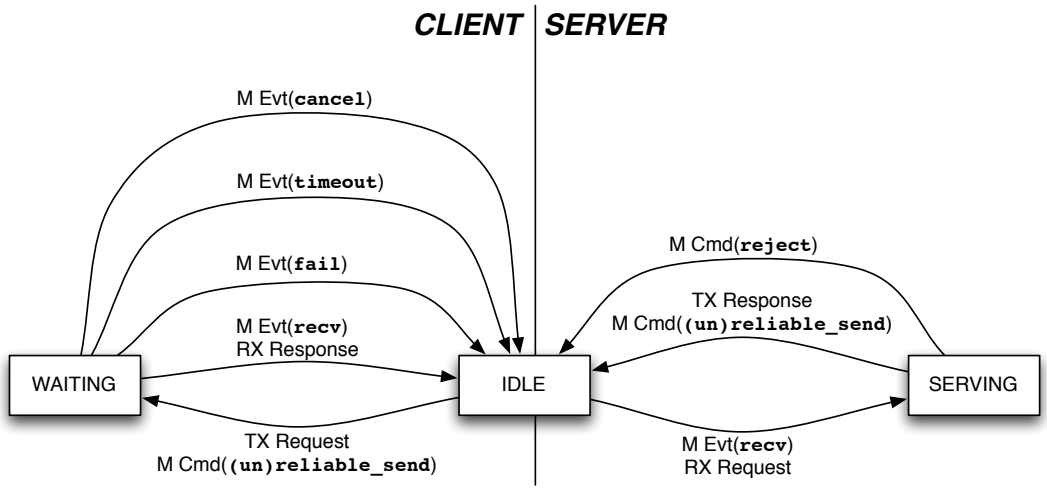

Figure 5.5: State machine for CoAP request/response-layer implementation

```
  uint16_t size );

event void fail (
  coap_session_t* session,
  error_t error ); }
```

The `coap_session_t` is a C struct containing information univocally identifying the session, that is *i*) the peer IP address and UDP port endpoint, *ii*) the Message ID (MID) assigned to the session, and *iii*) state information of the session itself, e.g., closed, waiting ack, or others, which can be useful to the request/response layer to correctly process that session.

The commands and the events represented in Figures 5.4 and 5.5 correspond to the ones described in the software interface. Commands not present in the software interface are mapped as follows: *i*) `reliable_send` and `unreliable_send` correspond to calling the `send` command with the `reliable` flag set as `true` or `false` respectively, *ii*) `reject` and `cancel` commands have equivalent semantics, so only the latter has been offered in the API, *iii*) `timeout` and `fail` have equivalent semantics as well, the latter has been offered together with a parameter indicating the reason of failure.

CoAP API description [11] We designed a CoAP implementation over Harvan's TinyOS 6LoWPAN library [106], which implements the first version of

the HC (RFC 4944) [81]. `CoAPP` is the REST component providing client and server functionalities; it handles session data regardless of its type (either client or server), thus optimizing its memory usage. The actual implementation of this component can handle up to COAP_MAX_TRANSACTIONS concurrent transactions [1]. Transactions are univocally identified by the *transaction ID*, that is an integer assigned locally to each request.

The `CoAPClient` interface provides the `CoAPP` module with a TinyOS command to send any arbitrary request to a CoAP endpoint, and a TinyOS event to manage the response it gets back. Next, we show the TinyOS code of the interface:

```
interface CoAPClient {
  command coap_tid_t request (
    coap_absuri_t* absuri,
    coap_method_t method,
    coap_content_t* content,
    bool acked );

  event void response (
    coap_tid_t tid,
    coap_status_t status,
    coap_content_t* content ); }
```

The interface defines different custom data types to provide better readability and high-level operations. When a `request` command is issued the user must provide *i)* `absuri` describing endpoint host, port and URI of the requested resource, *ii)* `method` specifying which method is used to access the requested resource, *iii)* `content` providing a pointer to the content to be attached to the request, if present, *iv)* `acked` to request a response message; the `request` command provides the user with the `coap_tid_t` internally assigned to the transaction. A `response` event is triggered when the related reply is received. This response contains *i)* a `tid` field identifying the transaction, *ii)* a `status` field containing the status code resulted after processing the request and *iii)* a pointer to the `content` piggybacked in the response.

[1] this value can be chosen arbitrarily at build time by trading between memory occupation and flexibility

The `CoAPServer` interface provides the `CoAPP` module with server capabilities: external components can use this interface to serve resources using a CoAP server. The `CoAPServer` and the the `CoAPClient` are complementary in the sense that commands issued using one interface trigger events managed by the other interface and viceversa.

```
interface CoAPServer {
  event void request (
    coap_rid_t rid,
    coap_absuri_t* uri,
    coap_method_t method,
    coap_content_t* content,
    bool toack );

  command error_t response (
    coap_rid_t rid,
    coap_status_t status,
    coap_content_t* content ); }
```

In order for a `request` to be properly processed, the following data is needed: *i)* a `rid` value internally assigned to univocally identify the request, *ii)* the `uri` of the requested resource, *iii)* `method` describing the access method, *iv)* a pointer to the `content`, if present, and *v)* a `toack` flag to signal if the client requested an ACK. The `response` command can be used by the serving module together with the following parameters, *i)* a `rid` to match the related request, *ii)* `status` value resulted from the processing of the request and *iii)* `content` pointer to data to be sent in the response.

The `CoAPServer` interface is characterized within the `CoAPP` module by a `port` parameter identifying on which UDP port the CoAP service has to be activated in the node.

The client/server architecture of the `CoAPP` module allows the implementation of lightweight Web services on constrained WS&AN nodes. Moreover, it makes it possible to implement M2M interactions, such as publish/subscribe, and to create multiple Web servers and services without burdening a constrained node system. As it will become evident from our experimental campaign, our design choices do not need heavy computational power, on the contrary the resulting `CoAPP` software proved to be fast and reliable in managing requests and responses.

Table 5.1: ROM/RAM utilization of TinyOS components

Component	ROM	RAM
TinyOS core	1396	4
802.15.4 and ActiveMessage	9258	327
UDP/6LoWPAN	5804	1983
CoAP	2668	1801
RAI/RPI	1752	548
libEXI	7134	1016
Subscription	1580	522
Resources	12402	526
HW drivers	7338	160
CoAP web-services	3632	208
EXI handling	1432	158

Experimental Results Table 5.1 shows the ROM/RAM footprint of our CoAP-enabled node implementation, which implements an extended version of the SENSEI TinyOS native-island node described in [4]. Our previous implementation led to a higher memory (ROM) consumption as it accounted for a separate implementation of the interface toward each component on the nodes (resources such as leds, on-board sensors and actuators, etc.), also considering the specific requirements of their XML schemas and the hardware drivers needed for their physical access. The current component instead leverages uniform interfaces which are reused for all components, leading to a lightweight implementation of Web servers on sensor devices.

In order to prove that CoAP components do not negatively impact the performance of the nodes, we set up an experiment with up to 20 CoAP servers running on a single telosb node (*serving node*). A second telosb node (*client node*) was used to send requests to these servers at a rate that was kept constant during each experiment and varied across them so as to highlight the dependence on this parameter. The outcomes of this test are shown in Figure 5.6 where we plot the CoAP request success probability (intended as the percentage of occurrences

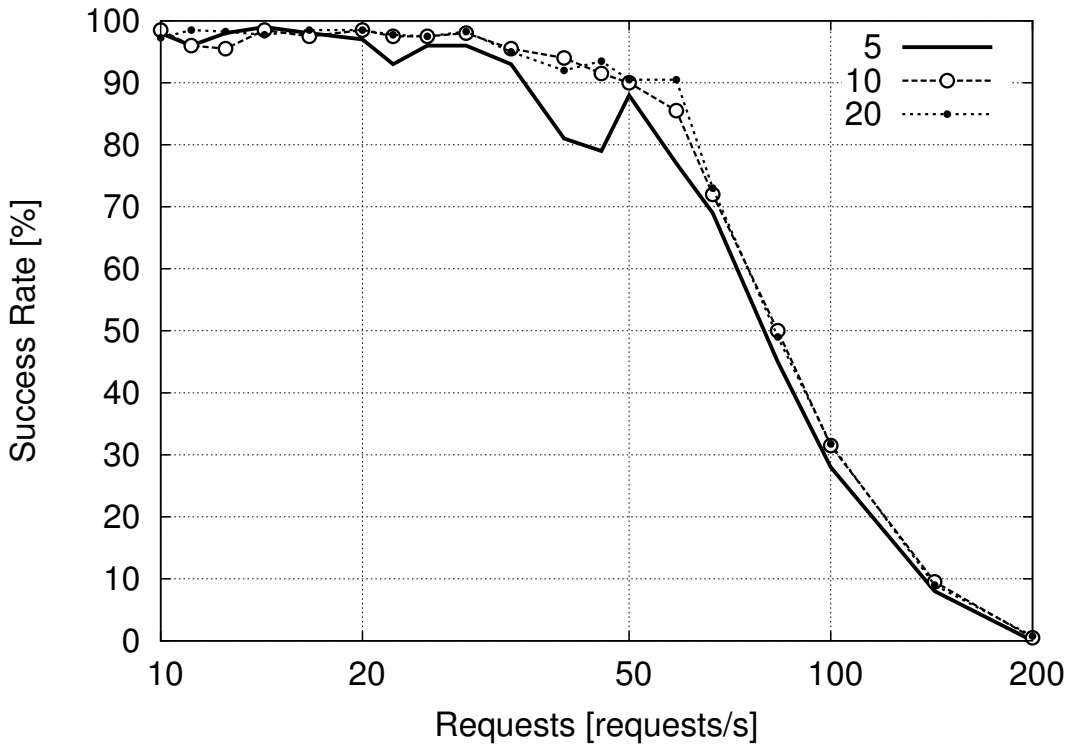

Figure 5.6: CoAP success rate vs. requests rate.

for which a request is successfully handled by the serving node) as a function of the request rate by the client node. The experiment has been run considering 5, 10 and 20 CoAP servers. Due to our efficient implementation of CoAP, the success probability depends only slightly on the number of servers running on the nodes. As expected, a very high (i.e., higher than 60 requests/second) request rate severely impacts the access performance but this is due to the inherent limitations of the old 6LoWPAN library [106] that we used during the experiments. Given that, we can conclude that our CoAP implementation scales well with the number of server instances without causing a major and noticeable decrease in the access performance.

5.2.4 Other IETF Contributions

During the standardization process at the IETF, a number of technical analyses, proposals and reviews have been contributed to the CoRE WG, towards the design

and specification of CoAP; we will not discuss that work here, the interested reader can find it in the CoRE mailing list [107].

Beyond this collaboration with the WG in the specification of CoAP, some other contributions have been sent to the IETF in the form of Internet Drafts on topics related to CoRE; these documents are briefly discussed in the following paragraphs.

Protocol analysis [17] This document highlighted some aspects of the protocol design not fully optimized.

In the first part of the document, the complexity required for implementing both the Token Option and the Message ID have been analyzed; the resulting output to the working group has been to discourage the use of the Token Option in the vast majority of the cases, where it is not strictly required. Moreover, the document points out that avoiding its usage results in higher efficiency in terms of protocol overhead, which is a valuable property for its intended use on constrained networks.

In the second part, transport-related aspects have been analyzed, and some technical proposals have been made, such as the availability of the response code class (e.g., 2xx, 3xx, etc.) in a distinct field, to support implementations which, for simplicity, do not want to understand all the available response codes. Afterwards, this feature has been added also to the CoAP protocol. Moreover, this draft proposed the introduction of a sequential MID (SMID), enabling partial reordering of Non-Confirmable messages. It covered also the possible application of SMID to transfer large resource representations.

Alternative transport [18] This document aims at exploring the benefits of designing a more general-purpose transport protocol for CoAP: it proposed the Constrained Messaging Protocol, which is designed to be implementated on constrained devices and proves to be suitable for transporting CoAP request/responses, as well as other Internet protocols.

CMP is a UDP protocol extension intended to reliably or unreliably transport messages from an endpoint to another, and to unreliably transmit messages to a multicast destination. CMP adds the following features on top of UDP: *i*) clear session definition supporting multiple messages; *ii*) reliable message transport

handling retransmissions and network duplication; *iii*) unreliable message transport partly handling duplication and out-of-order delivery; *iv*) reliable message fragmentation in multiple UDP datagrams.

Alive message [**20**] In the context of a Constrained RESTful Environment (CoRE), hosts could frequently be energy-constrained and be turned off most of the time for energy-saving purposes. In the case of a CoAP server, while it is offline, it is not available to serve requests. Clients desiring to access its resources have no way to understand when they will find it up again. This specification provides a simple new message that gives a CoAP server the ability to signal its current availability in the network, i.e., the "Alive" message.

An "Alive" message (ALV) indicates that a CoAP server is up and ready to serve requests. When a client receives an ALV message from a server, if it is interested in any resource served by it, the client can immediately send a request to it since the Alive message provides an indication of its current availability.

5.3 Efficient XML Interchange

The EXI W3C working group has been formed mainly with the task of mitigating the size and parsing complexity of the XML information set; this activity has been supported by the XML Binary Characterization WG, which identified and described a set of use cases where the regular XML format was not viable and called for a binary representation of it.

The result of the EXI WG is a valid tool to exchange structured information in bandwidth-limited networks, and to process a wide range of data obtaining acceptable performance even under severe limitations. The EXI format has been specifically designed to reach the broadest set of technologies, via a design principle focused on simple and efficient solutions.

The well-known flexibility of the XML data format properly fits the needs of scenarios featuring a large variety of data types. SGs are expected to be composed of a wide range of devices, varying in terms of vendors, capabilities and functionalities. For instance there will be devices for measuring or visualizing a particular set of data, devices in charge of monitoring the network behavior as

well as actuators interacting with the system.

A versatile data representation is of paramount importance for the implementation and user-friendliness of the whole system. A practical and compact XML representation coupled with a light-weight processing engine as those conceived in the EXI framework are key to this objective.

The standard itself defines both a schema-informed and a non-informed operating mode. While the best flexibility is obtained in the non-informed mode, this characteristic is deemed not to be mandatory in SGs. Therefore, it is better to resort to schema-informed operations, which achieve the best results in terms of processing and compression efficiency. The latter exploits available information about the document structure to generate a document grammar, i.e., a set of rules describing the organization of the contents within the document. The grammar information can then be used to define a set of short event codes. The schema-informed mode, coupled with the short event codes, allows for compression factors of up to 90% in terms of memory occupancy on the device.

EXI content format is particularly suitable for SG environments, where many devices are deployed with very specific tasks in different points of the grid: while leading to the generation of different types of data, all the devices forming a SG are likely to be describeable using a single schema, thus making the information directly useable within the SG with M2M communication only. For example, a data report may contain a set of contents and attributes (i.e., data) of specified type and length which are readily understandable by any device regardless of whether the report has been produced by powerline metering equipment or by a light/temperature sensor for green HVAC control.

As an added benefit, a schema-informed EXI stream results in a very compact version of an XML document, thus reducing the size of encoded information down to about 20 to 50 times less than the original, ASCII-encoded XML document [11]. The transmission of XML contents via EXI streams has been successfully applied on MSP430-based devices supporting a generic schema-informed EXI processor, instructed to interpret and process a specific schema via a grammar description designed to be easily accessible and processed in the limited memory of the device [11] The processor takes up about 7 kB of program size and 1 kB of RAM, which is an affordable memory cost for implementing web service capabilities in constrained hardware.

5.3.1 EXI compressor [11]

EXI compression exploits information about the document structure to internally generate small tags based upon the current XML schema, the current processing stage and the context. Also, tags data representation is optimized to be as compact as possible.

Although an efficient compression can be achieved from the XML schema, the standard defines other operating modes to produce a more compact representation of the XML file using only partial or no XML schema information.

The encoded XML document results in an EXI stream, which represents the document in binary format where every data tag of the document is encoded using an event code. Event codes are binary tags that preserve their value only in their assigned position within the EXI stream.

Thus, EXI implements event-based encoding: for efficient encoding, at any given point of an XML stream a set of grammars is used to understand which event is most likely to occur next. Such a set of grammars, representing the XML document structure, has to be produced before the actual EXI processing. The sequence of events describes the sequence of finite-state machines defined using each different grammar as transition function.

In an EXI stream every XML element is represented using a specific grammar; each grammar consists of a set of productions, defining the set of possible events in a specific state. EXI assigns an event–code (EC) to each production. The sequence of XML elements coded into ECs forms an EXI stream. When a new element of the EXI stream is parsed, a new grammar associated with the element is stacked upon the preceding one and the control is passed to a new automaton which is in charge of handling the new grammar, until the new element is completed and the control returns to the preceding routine.

We designed and implemented *libEXI*, a realization of the EXI processor that has been specifically targeted for resource constrained MCUs (e.g., Texas Instrument MSP430). The design required to limit the number of features implemented. libEXI is a byte-aligned and schema-informed EXI processor, which encodes EXI streams using a preprocessed grammar set (defining the XML schema in use) and a pre-processed C data structure set (representing a document compliant to the XML schema).

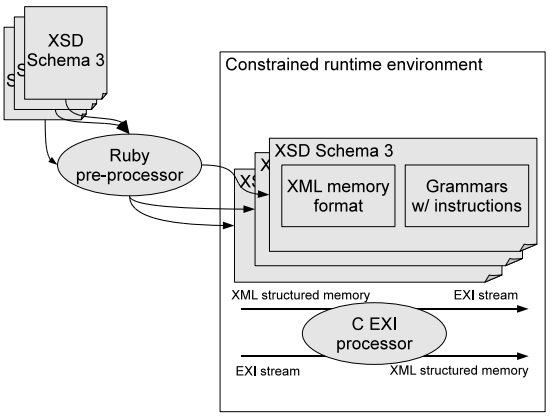

Figure 5.7: Usage diagram for the EXI processor and pre-processor.

Hence, libEXI can translate EXI streams into a structured memory representation which can be stored and processed by CPU-constrained devices. Bit-aligned encoding, even if very efficient, showed to be too complex to match our requirements.

As shown in Figure 5.7, our EXI library uses the results of a preprocessing phase: a Ruby pre-processor has to run on the XML schemas before libEXI can process EXI streams. This preprocessor extracts from any XML schema the set of grammars required to encode and decode EXI streams; in addition, it builds a set of C structures representing the XML document layers. The libEXI memory representation built by the pre-processor is an optimized translation of the XML document contents, needed for constrained devices to properly manage EXI streams.

The libEXI processor uses a grammar stack to encode/decode EXI streams. Grammars contain the list of events as well as the information of which grammar has to be stacked to handle the next part of the EXI stream, and into which production the current grammar will be when the control returns to it. Any new grammar piled in the grammar stack corresponds to a new execution of the encode/decode function call: in this way, there is a one-to-one mapping between the processor internal stack and the current grammars stack.

Table 5.2: XML compression performance: size of the compressed stream (bytes) and compression ratio (between parentheses)

Format	Schema-1	Schema-2	Schema-3
Uncompressed XML	591 bytes	242 bytes	229 bytes
Gzip	302 bytes (*0.51*)	206 bytes (*0.85*)	175 bytes (*0.76*)
enhanced Xmill (Xwrt) [101]	784 bytes (*1.33*)	431 bytes (*1.78*)	453 bytes (*1.97*)
XMLPPM [102]	262 bytes (*0.44*)	164 bytes (*0.68*)	128 bytes (*0.55*)
EXI w/o schema byte-aligned	298 bytes (*0.50*)	104 bytes (*0.43*)	99 bytes (*0.43*)
EXI w/o schema bit-aligned	237 bytes (*0.40*)	96 bytes (*0.40*)	83 bytes (*0.36*)
EXI w/ schema byte-aligned	58 bytes (*0.10*)	10 bytes (*0.04*)	41 bytes (*0.17*)
EXI w/ schema bit-aligned	27 bytes (*0.05*)	4 bytes (*0.02*)	26 bytes (*0.11*)

5.3.2 Results

In Table 5.2 we show the compression efficiency for EXI, also showing that of other compression schemes, i.e., Gzip, enhanced Xmill (Xwrt) [101] and XMLPPM [102]. The size of the Uncompressed XML document, which is taken as the reference document for the experiment, is expressed in bytes. For the compression schemes we show the size of the compressed documents (also in bytes) and their compression ratio (within parentheses in the table), defined as the ratio between the sizes of the compressed stream and that of the uncompressed XML. The XML document we picked for our experiments is described by simple schemas that are suitable for, e.g., environmental monitoring and binary actuation (i.e., the operations that we expect from a sensor node). As demonstrated by the experimental results in Table 5.2, EXI compression is extremely efficient especially for schema-informed XML compression, leading to compressed files that are as small as just 4 bytes, thus achieving a compression efficiency of 50 times (the inverse of the compression ratio). We remark that this is particularly important for resource constrained sensor nodes as EXI dramatically reduces the amount of data that has to be transferred (most likely via radio transmission) through the network.

As a last remark, we note that the bit-aligned mode is the most convenient

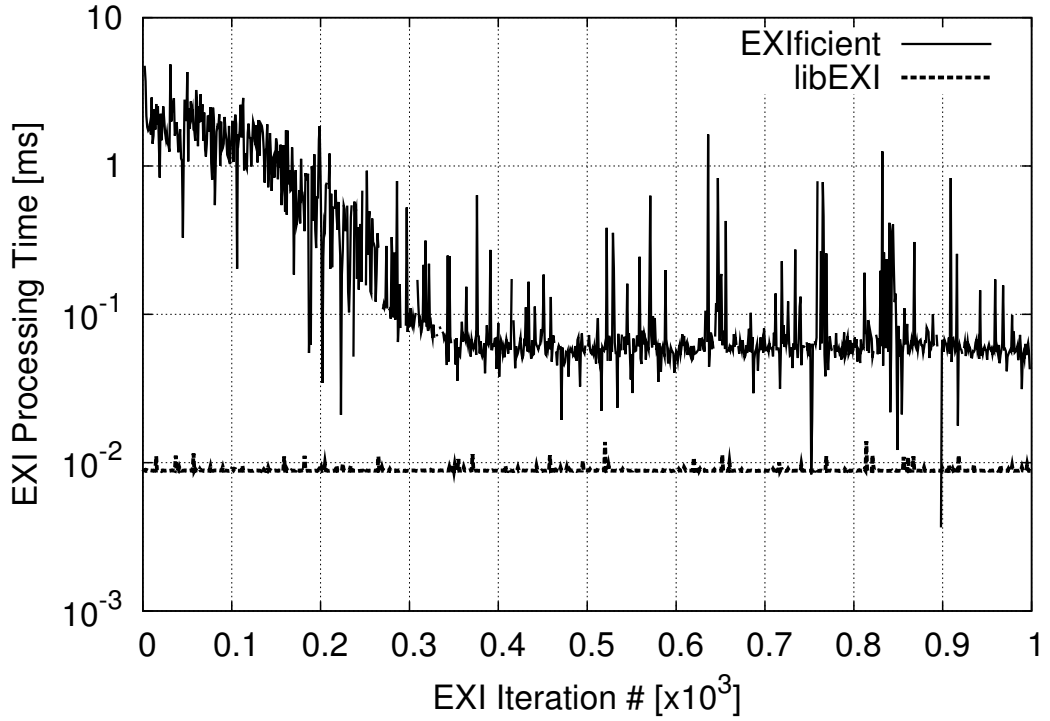

Figure 5.8: EXI processing time against iteration number.

choice, however, its implementation is more complex on MCUs working in byte-aligned mode. Thus, implementors may want to use the byte-aligned mode even though it provides inferior results.

As a third set of experiments, our EXI implementation has been extensively tested on a regular desktop PC and compared against EXIficient [108], a well-known and freely available Java implementation of EXI. The design criteria of EXIficient are very different from those of libEXI: Java was chosen due to its portability and all EXI features were implemented. However, the resulting implementation is not optimized for energy constrained devices and, as we show shortly, its performance is not consistent across repeated applications to the same document.

The steady-state XML throughput (number of processed XML elements per second) has been measured for libEXI and EXIficient. The latter can output an EXI stream at the maximum rate of about 0.9 millions of XML elements per second, whereas libEXI outputs about 6 millions XML elements per second.

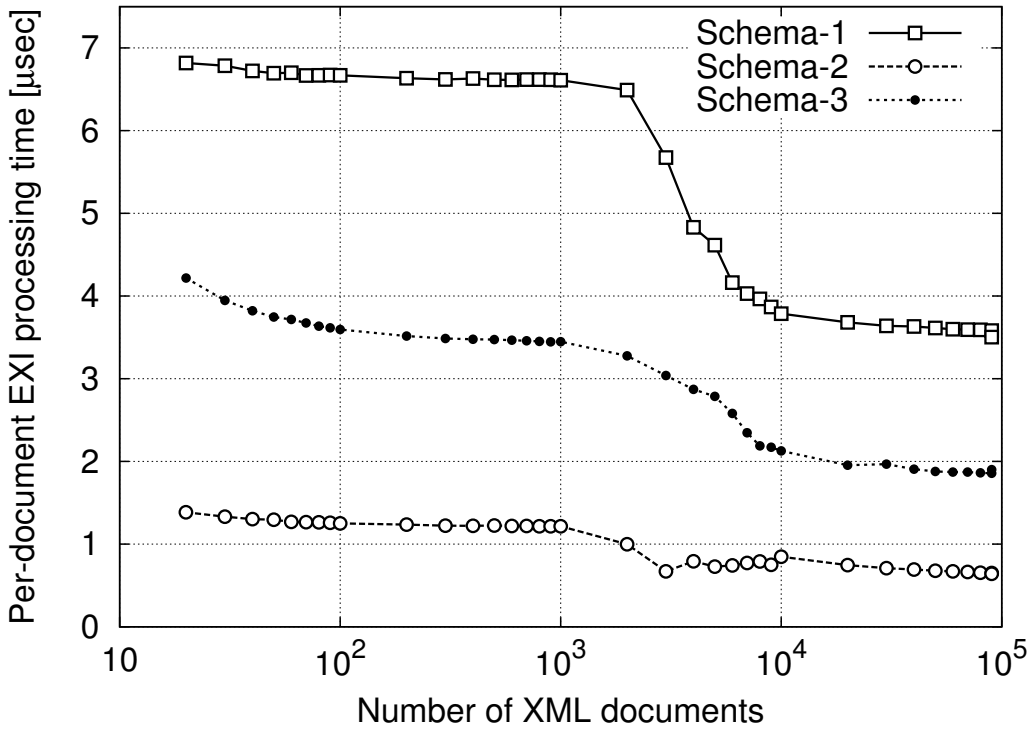

Figure 5.9: libEXI per-document processing time against number of XML documents.

Next, we look at the *XML processing time*, which is the time taken to compress an input XML file (and is inversely proportional to the XML throughput). Notably, the processing time of EXIficient decreases across repeated applications to the same XML file, showing a (non-negligible) transient phase at the beginning of which its performance is much worse (up to 50 times) than that in steady-state. This is shown in Figure 5.8, where we plot the average processing time for libEXI and EXIficient for a reference document containing 50 XML elements. As can be seen from the same figure, the processing time of libEXI is nearly constant across repeated compressions of the same XML file.

Finally, in Figure 5.9 we shown the average libEXI processing time for XML documents containing a single XML element. These small-sized documents are relevant to the IoT as they can represent queries to, e.g., acquire the readings of specific sensor nodes. Obtaining the compressed EXI stream of these documents is very fast (less than 10 micro-seconds), which is desirable for, e.g., a proxy

connecting devices within the Internet domain with the IoT. In fact, one of the main functions of this proxy will be that of performing conversions between EXI and XML. EXI will be the preferred format for the constrained devices residing within the IoT, whereas XML will be used by the more powerful computers located within the Internet.

6

Browsing the Internet of Things

The term Web 2.0, which has been introduced in 2005 [109], refers to a substantial change in the design of web applications; the new version of the Internet, while not asking for a similar technological upgrade, is nowadays more focused on the user experience, offering participatory content sharing, interoperability and social collaboration. Modern websites do not restrict users to be passive content consumers, but they enable cooperative interaction aimed at creating, sharing and exploiting contents in a virtual community.

Similarly, with the advent of the Internet of Things (IoT) [110], smart things are assuming a central role in the Internet community. A smart thing can be any device capable of processing and communication; hence, IoT devices can be as simple as a temperature sensor, or as complex as a Personal Digital Assistant (PDA) connected to a whole Body Area Network (BAN) designed for medical purposes [111], or anything in between these two examples.

Recent efforts to integrate the IoT into the Web 2.0 obtained some valuable results, such as Pachube [32] and SensorMap [27], aimed at providing the user with a web platform capable of visualizing networked things in a similar way as Google Maps does for Points of Interest. However, to the best of our knowledge, all these tools restrict users to using a simple predefined interface. In our opinion, a complete integration of smart things into the Web 2.0 will only be achieved when users are able to develop, deploy and exploit their own IoT applications as they already do for website and online contents. The key enablers for the success of such an integration are *i*) the adoption of capable open standards for web

communications [11], and *ii*) a next-generation "Web 2.0"-enabled application design framework for smart things [6].

The first part of this chapter describes how to smoothly bridge the CoAP application protocol with the Internet by using a web proxy that transparently operates the translation between CoAP and HTTP. Also some advanced HTTP-CoAP cross-protocol proxying techniques will be presented, which are still currently under discussion at the IETF [12].

In the second part, we address the need for a web application framework for the IoT by presenting the design of WebIoT [6]. WebIoT is a novel web application framework, based on the Google Web Toolkit [112], which provides users with simple methods for integrating smart things into a flexible visualization tool, for manipulating them both graphically and functionally, and for managing them and their interactions. Our framework has been developed leveraging on the following principles: thing-centric design, modularity and web service communications.

6.1 Cross-Protocol HTTP-CoAP Mapping [12]

CoAP is designed beyond communicating only with CoAP endpoints, but spanning across the REST-based protocols domain to support HTTP. HTTP interworking is a central goal while aiming to integrate into "the Web" smart constrained objects while pursuing the Internet of Things paradigm.

CoAP achieves this design goal thanks to the REST architectural style shared with HTTP [12], and in particular to the fact that both CoAP and HTTP can interoperate through proxies; a REST intermediary between CoAP and HTTP endpoints is an ideal candidate for unobtrusive and transparent translation between the two protocols, without posing further requirements either on the client or on the server. Self-described REST messages may be handled by stateless intermediaries and either parsed or represented in any REST-based dialect, i.e., CoAP or HTTP.

The mapping process between CoAP and HTTP is straightforward, based on the fact that equivalent methods, response codes and options are present in both protocols, thus enabling a simple translation process by applying a static mapping.

Even if resources are identified using URIs in CoAP and HTTP, URI domains

are distinct between them; for this reason an additional URI mapping process may be required, in order to support endpoints unaware of both URI domains. For example, thanks to this process, a CoAP resource can be made available at a regular HTTP URI so that older web clients can access it even if completely unaware of the CoAP protocol domain.

Such a protocol translation, usable even by legacy HTTP agents, should be provided by a transparent, unobstrusive and effective proxy; this goal may be obtained by implementing and deploying it in different ways. Among the various techniques it is worth to mention that it is possible to provide such a service even through an Interception Proxy (RFC3040); this proxy deployed in a network location suitable for traffic interception will automatically redirect client requests to itself and provide protocol translation transparently, thus obtaining a seamless Web extension across the two protocol domains.

This proxy-based approach leaves space to even more complex communication patterns across HTTP and CoAP, such as multicast communication. Leveraging the multiplexing role of the proxy in such architecture, further steps towards connecting smart object clusters to the Internet may include extending its functionalities to support, for example, the mapping of unicast HTTP requests to multicast CoAP messages, by aggregating multiple responses in a single HTTP response payload.

6.1.1 Cross-Protocol Proxies

A device providing cross-protocol HTTP-CoAP mapping is called an HTTP-CoAP cross-protocol proxy (cross proxy).

At least two different kinds of HC proxies exist: *i*) *One-way cross proxy*: translates from a client of a protocol to a server of another protocol but not vice-versa, and *ii*) *Bidirectional cross proxy*: translates from a client of both protocols to a server supporting one protocol.

1-way and 2-way HC proxies are realized using the following general types of proxies:

- **Forward proxy (F)**: It is a proxy known by the client (either CoAP or HTTP) used to access a specific cross-protocol server (respectively HTTP or CoAP). Main feature: server(s) do not require to be known in advance

by the proxy (ZSC: Zero Server Configuration).

- **Reverse proxy (R)**: It is a proxy known by the client to be the server, however for a subset of resources it works as a proxy, by knowing the real server(s) serving each resource. When a cross-protocol resource is accessed by a client, the request will be silently forwarded by the reverse proxy to the real server (running a different protocol). If a response is received by the reverse proxy, it will be mapped, if possible, to the original protocol and sent back to the client. Main feature: client(s) do not require to know in advance the proxy (ZCC: Zero Client Configuration).

- **Interception proxy (I)**: This proxy can intercept any origin protocol request (HTTP or CoAP) and map it to the destination protocol, without any kind of knowledge about the client or server involved in the exchange. Main feature: client(s) and server(s) do not require to know or be known in advance by the proxy (ZCC and ZSC).

A server side proxy is placed in the same network domain of the server. Conversely a client side is in the same network domain of the client. Differently from these two cases, the proxy is said to be external.

For example, an HTTP-CoAP Reverse Cross (HCRC) Proxy is accessed by web clients only supporting HTTP, and handles their requests directed to CoAP servers by mapping them to CoAP, and mapping back the received response to HTTP. This mechanism is transparent to the client, which may assume that it is communicating with a regular HTTP server.

Typically, the HCRC Proxy is expected to be located at the server side, in particular deployed at the edge of the constrained network. The arguments supporting SS placement in this case are the following:
TCP/UDP: Translation between HTTP and CoAP also requires a TCP to UDP mapping; the UDP performance over the unconstrained Internet may not be adequate. In order to minimize the number of required retransmissions and overall reliability, TCP/UDP conversion SHOULD be performed at a server side placed proxy.
Caching: Efficient caching requires that all the CoAP traffic is intercepted by the same proxy, thus a server side placement, collecting all the traffic, is strategical for this need.

Multicast: To support CoAP using local-multicast functionalities available in the constrained network, the cross proxy MAY require a network interface directly attached to the constrained network.

6.1.2 Cross-Protocol URI Mapping

Motivation — A Uniform Resource Identifier (URI) provides a simple and extensible means for identifying a resource. It enables uniform identification of resources via a separately defined extensible set of naming schemes [113].

URIs are formed of at least three components: scheme, authority and path. The scheme is the first part of the URI, and it often corresponds to the protocol used to access the resource. However, the scheme does not imply that a particular protocol is used to access the resource.

Both CoAP and HTTP implement the REST paradigm, so, in general, the same web resource, i.e., identified by the same URI, can be accessed using either one of these protocols.

This could happen as long as the URI scheme of the target resource is supported by the client; however, web clients may support only a limited set of schemes. Example: HTTP clients typically support only 'http' and 'https' schemes.

Whenever there does not exist a URI to access the resource with a scheme supported by the client, communication may still happen if the cross proxy supports mapping URIs to a supported scheme.

URI mapping — Identifies the act of providing an alternative URI to access a target resource.

Example: Assume that the target resource is "coap://node.coap.something.net/foo". A possible URI mapping could be "http://node.something.net/foobar".

In the previous example the scheme changes between the mapped URI and the original one; this special kind of URI is defined here as cross-protocol URI (or cross URI). Cross proxies may provide cross URIs to enable clients supporting only a limited set of schemes, to have access even to resources identified by a scheme they do not natively support.

However, when a cross-protocol URI exists, the authority and path parts of the URI may change as well. For example, assume that the following re-

source exists - “coap://node.coap.something.net/foo”; the resource identified by “http://node.coap.something.net/foo” may not exist or be non-equivalent to the one identified by the ’coap’ scheme.

Generally speaking, the process of providing cross URIs could be complex, since a proper mechanism to statically or dynamically (discover) map the resource is needed. Two static HC URI mappings are discussed along the following lines.

- *Homogeneous Cross URI*: A cross URI is defined as homogeneous, when on two different schemes the very same authority and path identifies the same resource. E.g., “coap://node.coap.something.net/foo” and “http://node.coap.something.net/foo” identify the same resource.

- *Embedded Cross URI*: A cross URI containing in its own URI path, the authority and path of the target URI, is said to be an embedded cross URI. E.g., the cross URI “http://hc-proxy.something.net/coap/node.coap.something.net/foo”, identifying the URI “coap://node.coap.something.net/foo”, explicitly contains the target URI in its path.

Through cross-protocol URI mappings, cross-protocol proxies can interoperate even with HTTP web clients that are not aware at all of CoAP, thus easing its adoption and interoperation with the traditional Internet formed by regular web browsers.

6.1.3 Advanced Mappings [13]

By leveraging the multiplexing role of the cross proxy in a constrained network, and as a further step toward realizing efficient communication between web browsers and smart objects, proxy functionalities may be extended to support more advanced cross-protocol interactions, such as: *i*) concurrent tunneling of IPv4 in IPv6 combined with the HTTP-CoAP mapping, *ii*) the mapping of unicast HTTP requests to multicast CoAP messages, and the consequent aggregation of multiple responses in a single HTTP response payload and *iii*) the establishment of an observe relationship through the proxy using well-known HTTP bidirectional techniques.

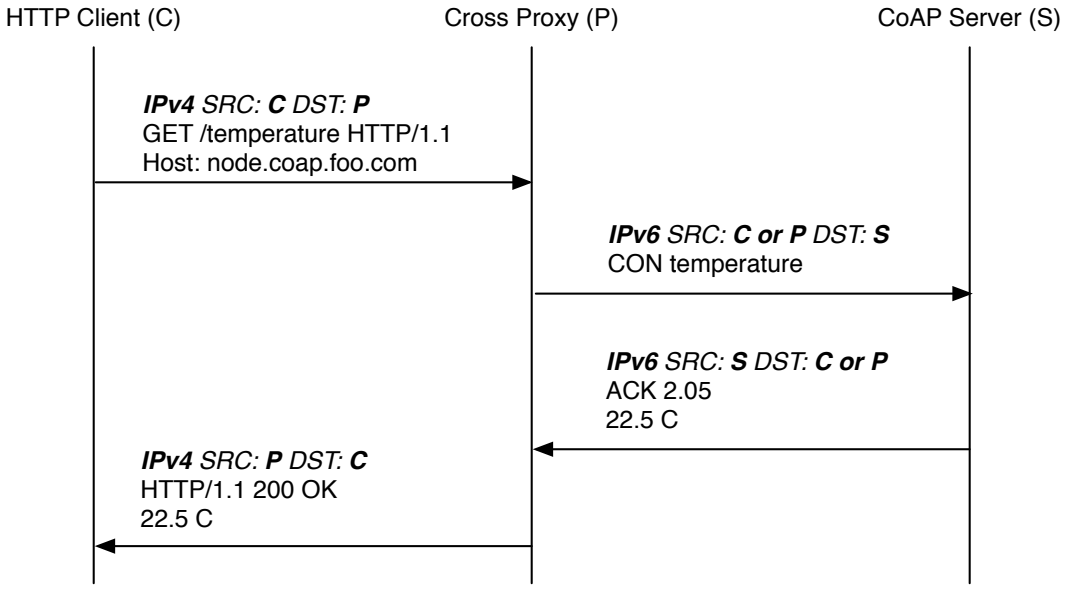

Figure 6.1: An example of mapping from HTTP/IPv4 to CoAP/IPv6.

6.1.3.1 HTTP/IPv4-CoAP/IPv6

Since IPv4 is still the dominant addressing technology used nowadays, and taking into account that a pervasive deployment of constrained nodes exploiting the IPv6 address space is expected: a v4/v6 gateway will be required to enable IPv4-only hosts to access CoAP resources offered by 6LoWPAN nodes.

To avoid complexity and reduce the number of required network services, the cross proxy itself could be in charge of operating the v4/v6 mapping together with the cross-protocol translation.

The v4/v6 translation could be simply operated by exploiting the URI authority information available in the HTTP Host header, in conjuction with conveniently compiled DNS entries that contain the actual v4/v6 mapping.

Assume that the DNS A entry for the target CoAP server, e.g., "node.coap.something.net", contains the IPv4 address of the cross proxy, whereas the AAAA entry contains the regular record, i.e., the address of the CoAP server. Since the A record points to the cross proxy, it will receive the HTTP/v4 traffic for the target server, then it can operate the HTTP-CoAP mapping and use the AAAA record to forward the request to the involved server. When a response

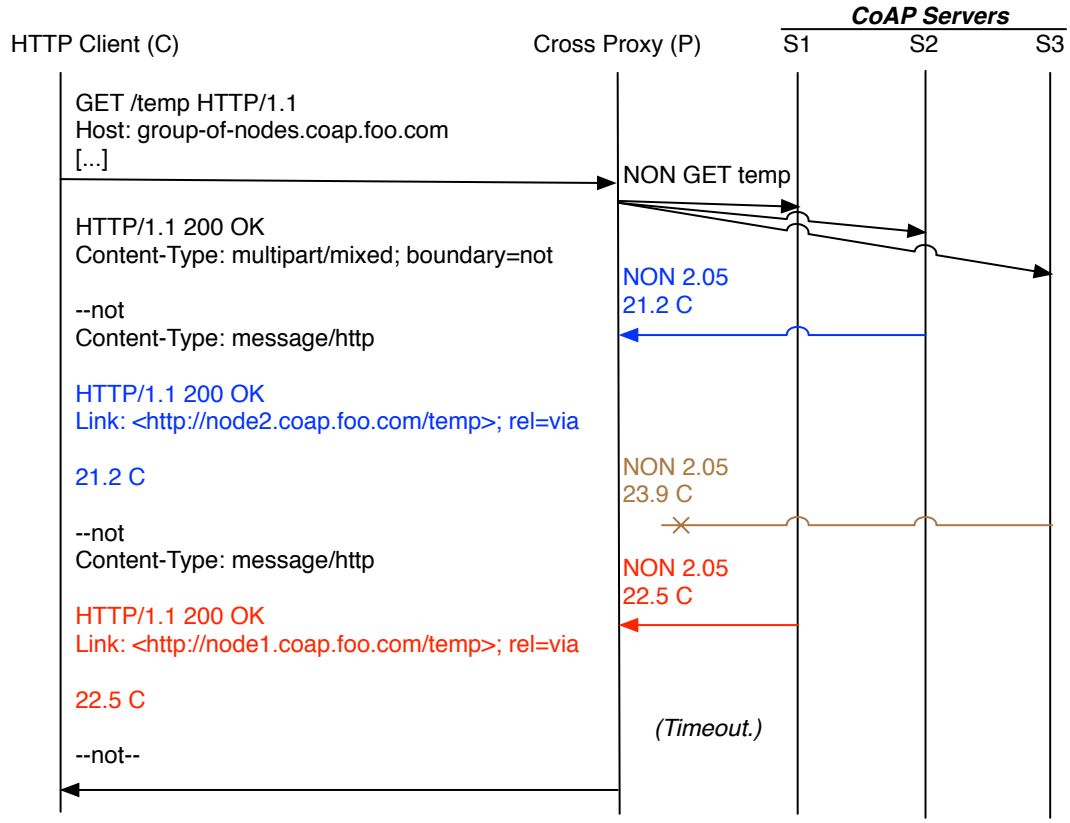

Figure 6.2: An example of mapping from unicast HTTP to multicast CoAP.

comes back, it will be translated to HTTP and forwarded to the waiting client.

Figure 6.1 is a graphical representation of the message exchange, when the cross proxy is also operating the v4/v6 mapping. If P is an interception cross proxy, it emits the CoAP request with the IPv6 of C as source address.

In order to obtain a working deployment for HTTP/IPv6 clients, an interception cross proxy deployment should be used, or Internet AAAA records should not point to the node anymore and the cross proxy should use a different DNS database pointing to the node.

6.1.3.2 Multicast

Figure 6.2 shows an HTTP client (C) requesting the resource "/foo" to a group of CoAP servers (S1/S2/S3) through an HC proxy (P) that uses IP multicast to

send the corresponding CoAP request.

The example proposed in the above diagram does not make any assumption on which underlying group communication technology is available in the constrained network. Some detailed discussion is provided about it along the following lines.

C makes a GET request to group-of-nodes.coap.something.net. This domain name may resolve either to the address of P, or to the IPv6 multicast address of the nodes (if IP multicast is supported and P is an interception proxy), or the proxy P is specifically known by the client that sends this request to it.

To successfully start multicast proxying operation, the HC proxy MUST know that the destination URI involves a group of CoAP servers, e.g., the authority "group-of-nodes.coap.something.net" is known to identify a group of nodes either by using an internal lookup table, using DNS paired with IPv6 multicast, or by using some other special technique.

A specific implementation option is proposed to further explain the proposed example. Assume that DNS is configured such that all subdomain queries to coap.something.net, such as group-of-nodes.coap.something.net, resolve to the address of P. P performs the HC URI mapping by removing the 'coap' subdomain from the authority and by switching the scheme from 'http' to 'coap' (result: "coap://group-of-node.something.net/foo"); "group-of-nodes.something.net" is resolved to an IPv6 multicast address to which S1, S2 and S3 belong. The proxy handles this request as multicast and sends the request to the multicast group .

6.1.3.3 Observe

Figure 6.3 shows the interaction between an HTTP client (C), an HC proxy (P), and a CoAP server (S) for the observation of the resource "temperature" (T) available on S.

C manifests its intention to observe T by including the Expect Header in the request; if P or S do not support this interaction, the request MUST fail with "417 Expectation Failed" return code. In the presented example, both P and C support this interaction, and the subscription is successful, as stated by the "206 Partial Content" return code.

At every notification corresponds the emission of an HTTP chunk formed by a single part, which has a "message/http" payload containing the full mapping of the notification. When the observation is dropped by the CoAP server, the

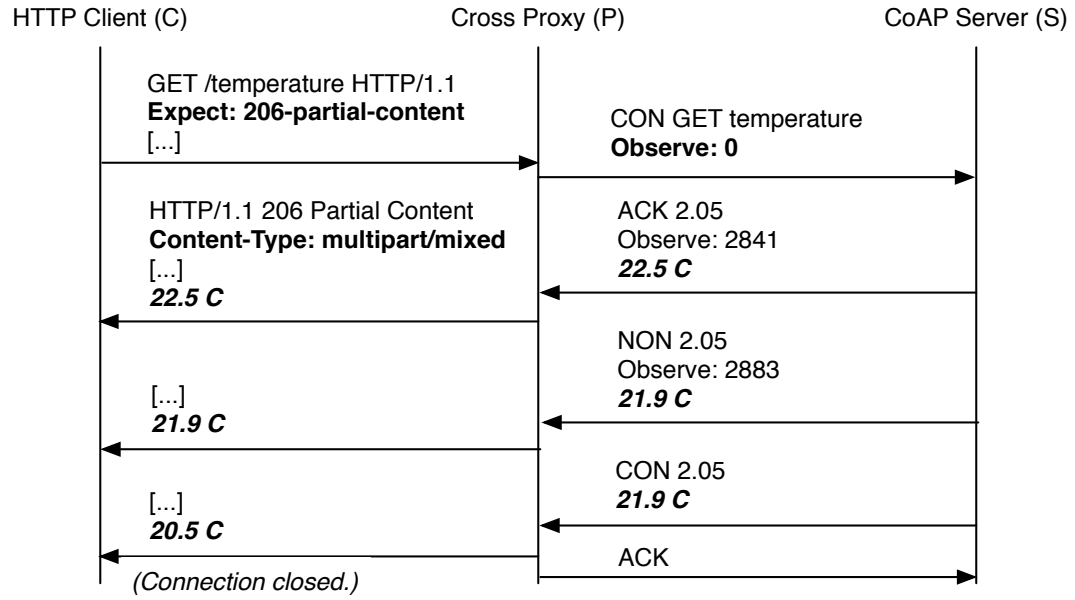

Figure 6.3: An example of mapping from HTTP streaming to CoAP observe.

HTTP streaming session is closed.

6.2 WebIoT [6]

WebIoT is a plugin based web application framework, which makes it possible for easy and quick development of graphical interfaces for the management of IoT networks. A heterogeneous device set can be visualized and controlled through an extensible user interface (UI) and backend application framework.

Figure 6.4 provides a snapshot of the WebIoT interface: the central UI element, which determines the visualization mode, is provided by a module of the web application, thus enabling transparent substitution and run time switching. The sidebar contains UI elements defined by plugins. The two panels are in charge of operations and settings, respectively. The toothed wheel on the sidebar allows for toggling the selected plugin from operation mode to configuration mode.

6.2.1 Design principles

Our design focused on a small set of design principles: *i)* Thing-centrality, *ii)* Modularity and *iii)* RESTful interactions. Although new IoT applications are appearing every day, these principles provides our framework the needed flexibility and adaptability to support them. Figure 6.5 highlights the main components of WebIoT.

Thing-centric design — The core of the framework consists of a single component providing a *uniform container* for `Thing` objects, which are abstractions used to represent real things in the framework. A `Thing` is defined by *i*) `ThingFeature`s, specifying the device characteristics and *ii*) `ThingDataSourceFeature`s, virtualizing the data sources available on it.

Thanks to this generalization, WebIoT can handle an arbitrary set of heterogeneous objects, which can be shared across the software components. In addition, plugins can leverage on these abstractions to implement generic behaviors on specific features: e.g., a map visualization plugin will show only objects defining a position feature.

High modularity — All the framework functionalities are provided by plugins: *i)* the background map overlays are offered by the `Maps` plugin, *ii)* the set of known devices is cooperatively built by a set of plugins, *iii)* the device control is managed through the `Things` plugin.

Plugins are totally independent and cooperate by sending or receiving specific events. Their shared functions are implemented in the web application core for enhancing code reuse. Also, heterogeneous devices may use different communication protocols, thus a common and extensible device interaction scheme for harmonizing different access interfaces is needed.

REST paradigm — The WebIoT framework has been developed as a *web centric application* based on the REpresentational State Transfer paradigm [4], due to the following reasons. First and foremost, web services are becoming more and more popular, and while this is due to their ease of access and maintenance, they proved to be a valid interaction model to access a wide spectrum of services.

Moreover, WebIoT can provide direct access to devices and to other web services through web interactions: thus, regardless of whether functionalities are provided by `Thing`s or are available in the Web, WebIoT can interact with them

Figure 6.4: A snapshot of WebIoT.

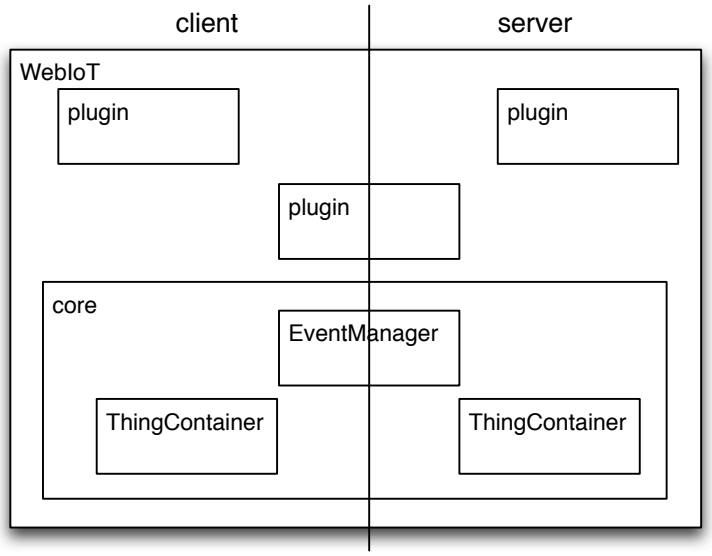

Figure 6.5: WebIoT architecture.

identically.

Last but not least, the framework is developed using the Google Web Toolkit (GWT) [112], allowing cross-platform development of both client-side and server-side components, using a uniform Java based programming language. The communications between client and server components are easily arranged using GWT translating component interfaces and Remote Procedure Calls (RPC), into low-level network interactions with servlets. Finally, GWT translates the client-side Java code into Javascript for multi-browser compatibility.

6.2.2 Core Services

`Thing` objects have a central role in WebIoT, since they provide real thing abstractions by reusing the same software object. Any `Thing` object is fully defined by the set of `ThingFeature`s associated to it. A `ThingFeature` is a specific characteristic that defines the object in WebIoT. A `ThingDataSourceFeature` virtualizes a specific information source available on the related device, and is technically implemented as a special class of a `ThingFeature`; specific data sources should further extend this class of features, e.g., for a temperature sensor a `ThingTempSourceFeature` class may be defined. Identification and univoc-

ity of `ThingFeature`s, where needed, is guaranteed by the fact that a `Thing` cannot have multiple features with the same name, e.g., multiple temperature sensors present on the same device will have different names. For instance, a commonly used `ThingFeature` is the device position, which can be known a priori, user defined or, in case of a GPS equipped device, derived from available `ThingDataSourceFeature` readings obtained from the GPS sensors.

WebIoT core is in charge of implementing the shared backend functionalities, which are summarized in the following categories[1]: *i)* handling `Thing` container; *ii)* managing `Event` registration, processing and dispatching; *iii)* providing the web authentication functions and process; *iv)* saving and reloading the state of the web application.

Modularity is based upon two different shared objects: *i)* `ThingContainer`, a `Thing` objects database and *ii)* `Event`s, shared message structures for inter-component interactions. The `ThingContainer` stores and indexes the features available in each object, by type and by name, in order to enable the components to quickly access the subset of `Thing` objects with the required characteristics, e.g., all the `ThingDataSourceFeatures` present on a device, or a specific sensor requesting it by name. `Event`s are defined as global objects, which are dispatched when a *significant change of state* occurrs, so that any affected component may take actions.

Common functions, such as authentication and handling of favorites, have been placed in the core and help make development quicker for applications by defining and reusing general purpose features. Authentication functions and process provide the means for identifying a user by providing a single sign-on process to the various components of the web application. Handling of favorites consists in a generalization that allows plugins to export and reload their operational states in XML format.

IoT applications often imply complex interactions, such as selecting a large number of things and assigning them a series of common operations; saving such complex interactions through favorites helps the user to easily perform such tasks without wasting time in repetitive command sequences.

[1]Due to space constraints, in this paper it is not possible to give a detailed description of WebIoT internal components.

6.2.3 Event-driven communication

Given that the overall software features will be offered by the set of plugins collaboratively, the model used for communication and interaction between a heterogeneous set of them has a very critical role in the software architecture.

Whereas typical component interactions are usually characterized through the definition of shared APIs, the use of a fixed software interface model leads to strict requirements on the set of components forming the whole software. This software characteristic is known as tight coupling and, even if it is usually simpler to design, it leads to lower flexibility in the system that is formed by hardly reusable components.

Event-driven inter-component communication is known to provide a looser coupling model between plugins, thus enabling easier component interconnection, higher code reuse and an overall higher flexibility of the system itself.

Events are defined as global objects available across the software, and their definition requires careful consideration to minimize the event dispatching overhead. In general an event is required when a "significant change of state" in a component occurs, so that any other interested component may take action depending upon it.

In WebIoT any plugin could be an event producer, but events will be dispatched only to modules that have expressed interest with respect to that specific event: this interest is expressed by enforcing that every component must register to relevant events.

Any event is characterized by its identifier and is enhanced with specific properties; in addition events may support the following functionalities: *i*) having a specific callback that each event consumer must call reporting the resulting state after event processing, *ii*) being combinable so that multiple events can be aggregated into a single one.

All the side information related to the handling of such special classes of events is stored in an `EventEnvelope` object attached to them, and is required by the event handling process to correctly perform the event dispatch operation.

Moreover an event may be related to a specific set of `Thing` objects, and, in this circumstance, using *targeted dispatching* reduces the event handling overhead. Using this technique, the references to `Thing` objects travel with the event itself

Table 6.1: `ThingsContainer` events supported in the core

`NewThing`	A new `Thing` object has been created
`DeleteThing`	A `Thing` object has been deleted
`ThingShowInfo`	A `ThingInfoPanel` has been opened
`ThingFeatureUpdate`	A `ThingFeature` property has been updated
`ThingDataUpdate`	A data source has updated data
`ThingSelectionUpdate`	A set of selected `Thing` objects have changed

inside the `EventEnvelope`, helping the dispatching module in determining which components are interested in receiving events related to the attached objects, and providing to each destination component only the subset of objects in which it will be interested.

The definition of events has an important role in the WebIoT context, because typical events involve specific `Thing` objects, and having a convenient way to route events related to objects across the plugins highly reduces plugin complexity and dispatching overhead.

Core event definition — WebIoT core defines a set of events enabling basic interactions among plugins, as shown in Table 6.1. The outlined events are focused on the interaction between the generic plugin and the `ThingContainer` object.

Components, depending upon their role in the overall framework, provide or consume a specific set of such `Event`s. For example a plugin may be interested in dispatching object creation and deletion `Event`s, which are, in turn, captured and consumed by interested plugins, e.g., a graphic module.

A special class of events are `RemoteEvent` objects, which are serializable, and can thus be passed between the application server and the web clients. This generalization allows smoother communication between parts of the plugins running on the web clients and their counterpart on the server; moreover, an event generated in a web client may be directly dispatched to other web clients or targeted to a subset of those clients by using targeted dispatching.

6.2.4 Plugin model

A plugin is a piece of application which interacts with the core using well-known interfaces; although plugins can be totally independent from one another, inter-plugin communication can be achieved using `Event`s.

Any plugin can belong to one or more of the following classes: *i) Visualizer*, to define UI parts, *ii) Provider*, to define `Thing` objects, and *iii) Manager*, to operate on them. Detailed descriptions of each class and examples of plugins belonging to them are offered in Sections 6.2.4.1, 6.2.4.2 and 6.2.4.3

Building the whole system using a cooperative approach easily allows the implementation of software combining any number of providers, visualizers and managers. This is obtained thanks to the loose coupling provided by the event-driven communication model. In addition, when working on some specialized `Thing` objects, ad hoc providers or visualizers may be required and built; they can still interact within the same software framework: general purpose plugins can integrate information from those objects, whereas specific plugins could work on the subset of specialized objects only.

6.2.4.1 Visualizers

A visualizer plugin can offer any element of the overall UI (e.g., the central element, the sidebar, etc.); a plugin providing the central UI element is called a *Base Visualizer*. Its graphical content is built using information contained in the `ThingContainer` object, by representing the subset of the supported `ThingFeature` and `ThingDataSourceFeature` objects.

Visualizers can easily be shared among different applications and, according to our experience, a map visualizer usually fits most of the IoT application use cases. However, advanced application requirements can be satisfied by extending the *Base Visualizer* with specialized UI provided by custom plugins.

Maps plugin — the *Maps* plugin specializes WebIoT with a *georeferenced user interface* providing object representations over a geographic map by using a Google Maps widget as its central UI element.

Through a map representation, it is also easy to specify object characteristics using graphic elements, the *graphical side information*, such as color, icon, label and size. WebIoT provides a wide range of `ThingFeature`s to easily enrich object

representation using this graphical side information.

6.2.4.2 Providers

A provider plugin adds new `Thing` objects to the `ThingContainer` and implements specific interactions with them; there are no constraints on how the provider should fetch and interact with such objects, nor on the type of objects that can be handled. For better modularization, technology-specific behaviors should be provided as high-level functionalities (where possible), while hardware specific implementations should be realized by the plugin itself.

On the current WebIoT implementation, we have developed two different providers. One implements the interactions with our department-wide testbed exploiting its software interfaces for firmware reprogramming, power management, node reservation and serial forwarding. This provider must be deployed on the application server managing the testbed itself.

The other provider, called NC-HTTP, interacts with a simple daemon accessing nodes physically connected via USB. NC-HTTP exports an HTTP interface, enabling the browser to directly interact with the provider using cross-origin resource sharing communication [114], without the requirement that the exchange goes through an application server.

6.2.4.3 Managers

A plugin operating on the `ThingContainer` and on its `Thing` objects is called a *Manager* plugin. A manager plugin will build on high-level functionalities offered by providers and will exploit general purpose visualizers to represent its specific content.

Implementing a manager plugin on top of a working set of visualizers and providers is an easy operation, and allows a high level of code reuse. Also, managers may be specialized to work on a subset of the `Thing` objects, thus enabling a quick development of the UI parts of some feature-specific application.

Things plugin — Exploiting the `Thing` object abstraction, we implemented the `Things` manager plugin, as an example of Manager plugins, which handles visualization and selection options of the `Thing` offered by the various providers.

WebResources plugin — Another plugin, successfully implemented in the

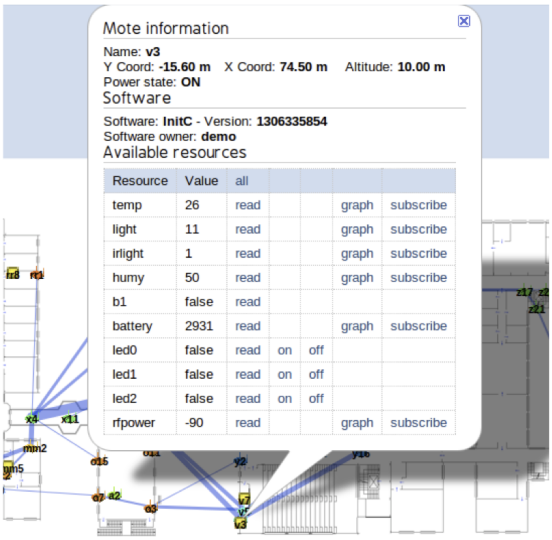

Figure 6.6: A snapshot of the WebResources plugin.

framework, is meant for `Thing` objects featuring CoAP [4] communication and runs the required software on the backend server in order to establish IP interconnectivity with the involved smart objects. The web resources plugin automatically detects nodes published on a reverse HTTP-CoAP proxy [12] by the web client and attaches a specific `ThingFeature` object containing all the resources a device offers to the related `Thing` object.

Every `Thing` that has the web resources `ThingFeature` will be visualized with an extended InfoPanel offering direct access to those resources, as shown in Figure 6.6: using cross-site resource sharing communication with the reverse HC proxy, resources can be accessed directly by the user web client.

7

Conclusions

This thesis has been devoted to the design, implementation and experimentation of a full-feature Internet of Things protocol stack, which is an important requirement to foster real adoption of such an innovative technology.

Along the dissertation we have evaluated available proposals for the realization of such a protocol stack, by active implementation and experimental evaluation, highlighting the issues that may technically undermine the straightforward adoption of such technologies.

This experience has allowed us to propose innovative technical approaches to solve these issues, such as the back pressure congestion control technique and the application of virtual memory to network buffers in constrained devices which have been shown in Chapter 4. Moreover, it has contributed to the realization of new standard protocols for the IoT, such as the pioneering work done on the Binary Web Service protocol, which contributed to the subsequent CoAP standardization, as well as the fruitful collaboration with the CoRE working-group on the definition of such protocols as discussed in Chapter 5.

During this thesis a constant attention has been devoted to practical experimentation and application of such technologies. Among others, these activities include: *i*) the deployment of the WISE-WAI Testbed, *ii*) the design of TinyNET (see Chapter 2), *iii*) the realization of SENSEI nodes (see Chapter 3), and *iv*) the design of WebIoT (see Chapter 6). These applied research experiences, which have been carried out in collaboration with SIGNET colleagues, have guided us in the definition of protocol requirements, and also have given us the possibility

to perform experimentation in realistic environments.

The results we have shown demonstrate the applicability of such technologies for the realization of the Internet of Things vision; we hope that this vision will be put into practice in the next years, but only a widespread adoption will demonstrate their usability and will feed in new research topics for innovation in this field.

List of Publications

The work presented in this thesis has appeared in the articles reported below.

Conferences

[C1] **Angelo P. Castellani**, Paolo Casari, and Michele Zorzi. TinyNET: A Tiny Network Framework for TinyOS. In *Proc. of IEEE IWCMC '09*, pages 580–585, 2009.

[C2] **Angelo P. Castellani**, Nicola Bui, Paolo Casari, Michele Rossi, Zach Shelby, and Michele Zorzi. Architecture and Protocols for the Internet of Things: A Case Study. In *Proc. of IEEE PerCom, WoT workshop*, Mannheim, Germany, April 2010.

[C3] **Angelo P. Castellani**, Muhammad Ikram Ashraf, Zach Shelby, Mika Luimula, Yuha Yli-Hemminki, and Nicola Bui. BinaryWS: Enabling the Embedded Web. In *Proc. of FNMS '10*, June 2010. poster paper.

[C4] **Angelo P. Castellani**, Mattia Gheda, Nicola Bui, Michele Rossi, and Michele Zorzi. Web services for the Internet of Things through CoAP and EXI. In *Proc. of IEEE ICC, RWFI workshop*, Kyoto, Japan, June 2011.

[C5] **Angelo P. Castellani**, Moreno Dissegna, Nicola Bui, and Michele Zorzi. WebIoT: A Web Application Framework for the Internet of Things. In *Proc. of IEEE WCNC, IoT-ET workshop*, April 2012.

[C6] **Angelo P. Castellani**, Giulio Ministeri, Marco Rotoloni, Lorenzo Vangelista, and Michele Zorzi. Interoperable and globally interconnected Smart

Grid using IPv6 and 6LoWPAN. In *Proc. of IEEE ICC, SaCoNeT workshop*, June 2012.

Journals and Book Chapters

[J1] Paolo Casari, **Angelo P. Castellani**, Angelo Cenedese, Claudio Lora, Michele Rossi, Luca Schenato, and Michele Zorzi. The WIreless SEnsor networks for city-Wide Ambient Intelligence (WISE-WAI) project. *MDPI Journal of Sensors*, 9(6):4056–4082, June 2009.

[J2] **Angelo P. Castellani**, Paolo Casari, and Michele Zorzi. TinyNET - A Tiny Network Framework for TinyOS: Description, Implementation and Experimentation. *Wireless Communications and Mobile Computing*, 10(1):101–114, January 2010.

[J3] Carsten Bormann, **Angelo P. Castellani**, and Zach Shelby. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing*, 16(2):62–67, March 2012.

[J4] Nicola Bui, **Angelo P. Castellani**, Paolo Casari, and Michele Zorzi. The Internet of Energy: A Web-enabled Smart Grid system. *IEEE Network Magazine*, 26(3), July 2012.

[J5] Nicola Bui, **Angelo P. Castellani**, Paolo Casari, Michele Rossi, Lorenzo Vangelista, and Michele Zorzi. *Smart Grid Communications and Networking*, chapter Implementation and performance evaluation of wireless sensor networks for smart grid, pages 324–350. 2012.

[J6] **Angelo P. Castellani**, Michele Rossi, and Michele Zorzi. Back Pressure Congestion Control for CoAP/6LoWPAN Networks. *Elsevier Ad Hoc Networks*. (submitted to Special Issue on From M2M Communications to the Internet of Things: Opportunities and Challenges).

IETF Internet-Drafts

[I1] **Angelo P. Castellani** and Mattia Gheda. CoAP overhead: protocol analysis and reduction proposals. IETF Internet Draft draft-castellani-core-coap-overhead-01, 2011.

[I2] **Angelo P. Castellani**. Constrained Messaging Protocol: an UDP protocol extension useful for CoAP and other protocols. IETF Internet Draft draft-castellani-core-transport-00, 2011.

[I3] **Angelo P. Castellani**. Learning CoAP separate responses by examples. IETF Internet Draft draft-castellani-lwig-coap-separate-responses-00, 2012.

[I4] **Angelo P. Castellani**, Salvatore Loreto, Akbar Rahman, Thomas Fossati, and Esko Dijk. Best practices for HTTP-CoAP mapping implementation. IETF Internet Draft draft-castellani-core-http-mapping-05, 2012.

[I5] **Angelo P. Castellani** and Salvatore Loreto. CoAP Alive message. IETF Internet Draft draft-castellani-core-alive-00, 2012.

IAB Position Papers

[P1] **Angelo P. Castellani**, Salvatore Loreto, Nicola Bui, and Michele Zorzi. Quickly interoperable Internet of Things using simple transparent gateways. In *Proc. of IAB Workshop Interconnecting Smart Objects with the Internet*, March 2011.

[P2] Thomas Fossati, **Angelo P. Castellani**, and Salvatore Loreto. (Un)trusted intermediaries in CoAP. In *Proc. of IAB Workshop Smart Objects Security*, March 2012.

Ringraziamenti

Ringrazio il Signore, per essersi mostrato nella mia vita ed avermi fatto conoscere il Suo Amore, senza di Lui sarei perso nel mio egoismo.

Ringrazio mia moglie, i miei figli e mio nipote, la vostra presenza mi ha sostenuto nei momenti difficili e mi ha profondamente aiutato umanamente in questi anni, senza di voi mi chiedo dove sarei ora.

Ringrazio i miei genitori, per aver sempre creduto in me ed avermi stimato nel lavoro e nella vita, non sarei arrivato a questo senza il vostro amore.

Ringrazio il mio supervisore Michele Zorzi, per avermi permesso di fare questa esperienza, avermi guidato nei momenti difficili, aver creduto in me fin dall'inizio ed avermi accompagnato fino alla fine.

Ringrazio Nicola Bui, Michele Rossi e Paolo Casari per essermi sempre stati d'aiuto ogni volta che ne avevo bisogno, avermi spronato e guidato costantemente durante tutti questi anni.

Ringrazio tutti i miei colleghi del SIGNET con cui ho avuto il piacere di lavorare in questi anni, fra gli altri Nicola Bressan, Mattia Gheda, Moreno Dissegna, Riccardo Manfrin, Giulio Ministeri, Francesco Fornasiero, Marco Visonà, Anicet Foba Togue, per aver condiviso con me questa esperienza, senza ciascuno di voi questo lavoro non mi sarebbe stato possibile.

I would like to thank all the IETF guys partecipating to the CoRE WG, working with you was a great chance for professional and human growth.

Angelo Paolo

References

[1] Paolo Casari, Angelo P. Castellani, Angelo Cenedese, Claudio Lora, Michele Rossi, Luca Schenato, and Michele Zorzi. **The WIreless SEnsor networks for city-Wide Ambient Intelligence (WISE-WAI) project**. *MDPI Journal of Sensors*, **9**(6):4056–4082, June 2009.

[2] Angelo P. Castellani, Paolo Casari, and Michele Zorzi. **TinyNET: A Tiny Network Framework for TinyOS**. In *Proc. of IEEE IWCMC '09*, pages 580–585, 2009.

[3] Angelo P. Castellani, Paolo Casari, and Michele Zorzi. **TinyNET - A Tiny Network Framework for TinyOS: Description, Implementation and Experimentation**. *Wireless Communications and Mobile Computing*, **10**(1):101–114, January 2010.

[4] Angelo P. Castellani, Nicola Bui, Paolo Casari, Michele Rossi, Zach Shelby, and Michele Zorzi. **Architecture and Protocols for the Internet of Things: A Case Study**. In *Proc. of IEEE PerCom, WoT workshop*, Mannheim, Germany, April 2010.

[5] Nicola Bui, Angelo P. Castellani, Paolo Casari, and Michele Zorzi. **The Internet of Energy: A Web-enabled Smart Grid system**. *IEEE Network Magazine*, **26**(3), July 2012.

[6] Angelo P. Castellani, Moreno Dissegna, Nicola Bui, and Michele Zorzi. **WebIoT: A Web Application Framework for the Internet of Things**. In *Proc. of IEEE WCNC, IoT-ET workshop*, April 2012.

[7] Angelo P. Castellani, Muhammad Ikram Ashraf, Zach Shelby, Mika Luimula, Yuha Yli-Hemminki, and Nicola Bui. **BinaryWS: Enabling the Embedded Web**. In *Proc. of FNMS '10*, June 2010. poster paper.

[8] Angelo P. Castellani, Giulio Ministeri, Marco Rotoloni, Lorenzo Vangelista, and Michele Zorzi. **Interoperable and globally interconnected Smart Grid using IPv6 and 6LoWPAN**. In *Proc. of IEEE ICC, SaCoNeT workshop*, June 2012.

[9] Angelo P. Castellani, Michele Rossi, and Michele Zorzi. **Back Pressure Congestion Control for CoAP/6LoWPAN Networks**. *Elsevier Ad Hoc Networks*. (submitted to Special Issue on From M2M Communications to the Internet of Things: Opportunities and Challenges).

[10] Carsten Bormann, Angelo P. Castellani, and Zach Shelby. **CoAP: An Application Protocol for Billions of Tiny Internet Nodes**. *IEEE Internet Computing*, **16**(2):62–67, March 2012.

[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 *Proc. of IEEE ICC, RWFI workshop*, Kyoto, Japan, June 2011.

[12] Angelo P. Castellani, Salvatore Loreto, Akbar Rahman, Thomas Fossati, and Esko Dijk. **Best practices for HTTP-CoAP mapping implementation**. IETF Internet Draft draft-castellani-core-http-mapping-05, 2012.

[13] Angelo P. Castellani, Salvatore Loreto, Akbar Rahman, Thomas Fossati, and Esko Dijk. **Best practices for HTTP-CoAP mapping implementation**. IETF Internet Draft draft-castellani-core-advanced-http-mapping-00, 2012.

[14] Nicola Bui, Angelo P. Castellani, Paolo Casari, Michele Rossi, Lorenzo Vangelista, and Michele Zorzi. *Smart Grid Communications and Networking*, chapter Implementation and performance evaluation of wireless sensor networks for smart grid, pages 324–350. 2012.

[15] Angelo P. Castellani, Salvatore Loreto, Nicola Bui, and Michele Zorzi. **Quickly interoperable Internet of Things using simple transparent gateways**. In *Proc. of IAB Workshop Interconnecting Smart Objects with the Internet*, March 2011.

[16] Thomas Fossati, Angelo P. Castellani, and Salvatore Loreto. **(Un)trusted intermediaries in CoAP**. In *Proc. of IAB Workshop Smart Objects Security*, March 2012.

[17] Angelo P. Castellani and Mattia Gheda. **CoAP overhead: protocol analysis and reduction proposals**. IETF Internet Draft draft-castellani-core-coap-overhead-01, 2011.

[18] Angelo P. Castellani. **Constrained Messaging Protocol: an UDP protocol extension useful for CoAP and other protocols**. IETF Internet Draft draft-castellani-core-transport-00, 2011.

[19] Angelo P. Castellani. **Learning CoAP separate responses by examples**. IETF Internet Draft draft-castellani-lwig-coap-separate-responses-00, 2012.

[20] Angelo P. Castellani and Salvatore Loreto. **CoAP Alive message**. IETF Internet Draft draft-castellani-core-alive-00, 2012.

[21] **WISE-WAI project web site**, 2008.

[22] Kevin A. Delin. **The Sensor Web: A Macro-Instrument for Coordinated Sensing**. *Sensors*, **2**(7):270–285, 2002.

[23] Steve H. L. Liang, Arie Croitoru, and C. Vincent Tao. **A distributed geospatial infrastructure for Sensor Web**. *Computers & Geosciences*, **31**(2):221 – 231, 2005.

[24] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. **MoteLab: a wireless sensor network testbed**. In *ACM/IEEE IPSN*, pages 483–488, April 2005.

[25] Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. **Kansei: a testbed for sensing at scale**. In *ACM/IEEE IPSN*, pages 399–406, Nashville, Tennessee, USA, 2006.

[26] Riccardo Crepaldi, Albert Harris, Alberto Scarpa, Andrea Zanella, and Michele Zorzi. **SignetLab: deployable sensor network testbed and management tool**. In *ACM SenSys*, pages 375–376, Boulder, Colorado, USA, 2006.

[27] Suman Nath, Jie Liu, Jessica Miller, Feng Zhao, and Andre Santanche. **SensorMap: a Web site for sensors world-wide**. In *ACM SenSys*, pages 373–374, New York, NY, USA, 2006.

[28] Guillermo Barrenetxea, Francois Ingelrest, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. **SensorScope: Out-of-the-Box Environmental Monitoring**. In *ACM/IEEE IPSN*, pages 332–343, Los Alamitos, CA, USA, 2008.

[29] Christian Prehofer, Jilles van Gurp, and Cristiano di Flora. **Towards the Web as a Platform for Ubiquitous Applications in Smart Spaces**. In *RSPSI Workshop at Ubicomp*, pages 16–19, Innsbruck, Austria, 2007.

[30] Dominique Guinard. **Towards the web of things: Web mashups for embedded devices**. In *ACM MEM WWW*, 2009.

[31] Tobias Baumgartner, Ioannis Chatzigiannakis, Maick Danckwardt, Christos Koninis, Alexander Kröller, Georgios Mylonas, Dennis Pfisterer, and Barry Porter. **Virtualising Testbeds to Support Large-Scale Reconfigurable Experimental Facilities**. In *Proc. of the 7th European Conference on Wireless Sensor Networks (EWSN 2010)*, **5970**, pages 210–223. Springer, Heidelberg, 2010.

[32] Usman Hague. **Pachube :: connecting environments, patching the planet**, 2010.

[33] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. **From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices**. In *Architecting the Internet of Things*, pages 97–129. Springer Berlin / Heidelberg, 2011.

[34] CrossBow Inc. **TelosB Mote Platform**, 2009.

[35] CrossBow Inc. **eKo Pro Series System**, 2010.

[36] Kevin Klues, Gregory Hackmann, Octav Chipara, and Chenyang Lu. **A component-based architecture for power-efficient access control in wireless sensor networks**. In *Proc. of ACM SenSys*, pages 59–72, Sydney, Australia, November 2007.

[37] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. **The nesC language: A holistic approach to networked embedded systems**. In *Proc. of ACM PLDI*, San Diego, CA, 2003.

[38] Philip Buonadonna, Jason Hill, and David Culler. **Active message communication for tiny networked sensors**, 2001.

[39] Xiaofan Jiang, Jay Taneja, Jorge Ortiz, Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, David Culler, Philip Levis, and Scott Shenker. **An architecture for energy management in wireless sensor networks**. *ACM SIGBED Review*, **4**(3), July 2007.

[40] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. **The Tenet architecture for tiered sensor networks**. In *Proc. of ACM SenSys*, pages 153–166, Boulder, CO, October 2006.

[41] The ZigBee Alliance. **ZigBee Specification**.

[42] Joseph Polastre, Jonathan W. Hui, Philip Levis, Jerry Zhao, David Culler, Scott Shenker, and Ion Stoica. **A unifying link abstraction for wireless sensor networks**. In *Proc. of ACM SenSys*, San Diego, CA, November 2005.

[43] David Culler, Prabal Dutta, Cheng Tien Ee, Rodrigo Fonseca, Jonathan W. Hui, Philip Levis, Joseph Polastre, Scott Shenker, Ion Stoica, Gillman Tolle, and Jerry Zhao. **Towards a sensor network architecture: lowering the waistline**. In *Proc. of USENIX HotOS*, Santa Fe, NM, June 2005.

[44] Adam Dunkels, Fredrik Osterlind, and Zhitao He. **An adaptive communication architecture for wireless sensor networks**. In *Proc. of ACM SenSys*, pages 335–349, Sidney, Australia, November 2007.

[45] Reinhardt Karnapke and Joerg Nolte. **COPRA - A communication processing architecture for wireless sensor networks**. In *Euro-Par 2006 Parallel Processing*, pages 951–960. Springer Berlin, 2006.

[46] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. **The emergence of networking abstractions and techniques in TinyOS**. In *Proc. of USENIX NSDI*, pages 1–14, San Francisco, CA, March 2004.

[47] Rajnish Kumar, Santashil Palchaudhuri, and Umakishore Ramach. **System support for cross-layering in Sensor Network Stack**. In *Mobile Ad-hoc and Sensor Networks*. Springer Berlin, 2006.

[48] Christophe J. Merlin and Wendi B. Heinzelman. **An information-sharing architecture for wireless sensor networks**. In *IEEE SECON*, 2006. demo session.

[49] The Contiki OS, 2005. [link].

[50] Michele Rossi, Michele Zorzi, and Ramesh R. Rao. **Statistically assisted routing algorithms (SARA) for hop count based forwarding in wireless sensor networks**. *Springer Wireless Networks Journal*, **14**(1):55–70, February 2008.

[51] **Sense & Sensitivity by Orange Lab**, 2009.

[52] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley, November 2009.

[53] EU Integrated Project. **SENSEI: Integrating the physical with the digital world of the network of the future**, 2008.

[54] Adam Dunkels and Jean Philippe Vasseur. **IP for Smart Objects**. IPSO Alliance White Paper No. 1, Sept. 2008, 2008.

[55] Jonathan W. Hui and David E. Culler. **IP is Dead, Long Live IP for Wireless Sensor Networks**. In *Proc. of ACM SenSys*, November 2008.

[56] Thomas Luckenbach, Peter Gober, Stefan Arbanowski, Andreas Kotsopoulos, and Kyle Kim. **TinyREST - a protocol for integrating sensor networks into the internet**. In *Proceedings of REALWSN*, Stockholm, Sweden, June 2005.

[57] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. **Tiny web services: design and implementation of interoperable and evolvable sensor networks**. In *Proceedings of ACM SenSys*, Raleigh, NC, November 2008.

[58] Dogan Yazar and Adam Dunkels. **Efficient Application Integration in IP-Based Sensor Networks for Emerging Energy Management Systems**. In *Proceedings of ACM Buildsys*, Berkeley, CA, US, Nov. 3 2009.

[59] Lars Schor, Philipp Sommer, and Roger Wattenhofer. **Towards a Zero-Configuration Wireless Sensor Network Architecture for Smart Buildings**. In *Proceedings of ACM Buildsys*, Berkeley, CA, US, Nov. 3 2009.

[60] Michele Rossi, Nicola Bui, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, and Michele Zorzi. **Code Dissemination in Wireless Sensor Networks using Fountain Codes**. *IEEE Trans. Mobile Comput.*, 2010. Accepted for publication.

[61] Barack Obama and Joe Biden. **New energy for America**, August 2008.

[62] Alexander Jung. % bf Building the Internet of Energy Supply. Spiegel Online International, December 2010.

[63] Nicola Bressan, Leonardo Bazzaco, Nicola Bui, Paolo Casari, Lorenzo Vangelista, and Michele Zorzi. **The Deployment of a Smart Monitoring System Using Wireless Sensor and Actuator Networks**. In *Proc. of IEEE SmartGridComm*, October 2010.

[64] Vijay K. Sood, Daniel Fischer, Mikael Eklund., and Tim Brown. **Developing a communication infrastructure for the Smart Grid**. In *Proc. of IEEE EPEC*, Montréal, Canada, October 2009.

[65] Oliver Kramer and Benjamin Satzger. **Power Prediction in Smart Grids with Evolutionary Local Kernel Regression**. In *Hybrid Artificial Intelligence Systems*, **6076** of *Lecture Notes in Computer Science*, pages 262–269. Springer, 2010.

[66] Michael Gravely. **DEFINING THE PATHWAY TO THE CALIFORNIA SMART GRID OF 2020**, 2009.

[67] Kris Pister, Pascal Thubert, Sicco Dwars, and Tom Phinney. **Industrial Routing Requirements in Low-Power and Lossy Networks**. IETF RFC 5673, October 2009.

[68] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[69] John Schneider and Takuki Kamiya. **Efficient XML Interchange (EXI) Format 1.0**. W3C Working Draft, 2008.

[70] **Application Protocols for Low-power V6 Networks (6lowapp) WG**, 2009.

[71] **Constrained RESTful Environments (core) WG**, 2010.

[72] Jonathan W. Hui and David E. Culler. **IPv6 in Low-Power Wireless Networks**. *Proceedings of the IEEE*, **98**(11):1865 –1878, November 2010.

[73] H. Tschofenig and J. Arkko. **Report from the Smart Object Workshop**. RFC (Informational), April 2012.

[74] **IPv6 over Low power WPAN (6lowpan) WG**, 2005.

[75] **Routing Over Low power and Lossy networks (roll) WG**, 2008.

[76] Tim Winter, Pascal Thubert, Anders Brandt, Jonathan W. Hui, Richard Kelsey, Philip Levis, Kris Pister, Rene Struik, Jean Philippe Vasseur, and Roger K. Alexander. **RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks**. RFC 6550 (Proposed Standard), March 2012.

[77] Cedric Chauvenet, Bernard Tourancheau, Denis Genon-Catalot, Pierre-Emmanuel Goudet, and Mathieu Pouillot. **A Communication Stack over PLC for Multi Physical Layer IPv6 Networking**. In *2010 First IEEE International Conference on Smart Grid Communications, SmartGridComm*, pages 250–255, Gaithersburg, Maryland, USA, October 2010.

[78] Daniel Popa, Jorjeta Jetcheva, Nicolas Dejean, Ruben Salazar, Jonathan W. Hui, and Kazuya Monden. **Applicability Statement for the Routing Protocol for Low Power and Lossy Networks (RPL) in AMI Networks**. IETF Internet Draft draft-ietf-roll-applicability-ami, 2011.

[79] Johanna Nieminen, Basavaraj Patil, Teemu Savolainen, Markus Isomaki, Zach Shelby, and Carles Gomez. **Transmission of IPv6 Packets over Bluetooth Low Energy**. IETF Internet Draft draft-ietf-6lowpan-btle, 2011.

[80] Iperf Team. [link].

[81] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan W. Hui, and David Culler. **Transmission of IPv6 Packets over IEEE 802.15.4 Networks**. RFC 4944 (Proposed Standard), September 2007.

[82] Jerome H. Saltzer, David P. Reed, and David D. Clark. **End-to-End Arguments in System Design**. *ACM Transactions on Computer Systems*, **2**(4):277–288, November 1984.

[83] V. Jacobson. **Congestion avoidance and control**. In *ACM SIGCOMM*, Stanford, CA, US, August 1988.

[84] L. Tassiulas and A. Ephremides. **Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multi-hop radio networks**. *IEEE Transactions on Automatic Control*, **37**(12):1936 –1948, December 1992.

[85] Scott Moeller, Avinash Sridharan, Bhaskar Krishnamachari, and Omprakash Gnawali. **Routing without routes: the backpressure collection protocol**. In *IEEE IPSN*, Stockholm, Sweden, April 2010.

[86] Chieh-Yih Wan, Shane B. Eisenman, and Andrew T. Campbell. **CODA: Congestion Detection and Avoidance in Sensor Networks**. In *ACM SenSys*, Los Angeles, CA, US, November 2003.

[87] V. Michopoulos, Lin Guan, G. Oikonomou, and I. Phillips. **A comparative study of congestion control algorithms in IPv6 Wireless Sensor Networks**. In *IEEE DCOSS*, Hangzhou, China, June 2011.

[88] R. Braden. **Requirements for Internet Hosts - Communication Layers**. RFC 1122 (Standard), October 1989.

[89] **IEEE Standard 802 Part 15.4: Wireless MAC and PHY Specifications for Low-Rate WPANs**, September 2006.

[90] **ns-3 Network Simulator**.

[91] J. Nagle. **Congestion Control in IP/TCP Internetworks**. RFC 896, January 1984.

[92] S. Massoud Amin and Bruce F. Wollenberg. **Toward a smart grid: power delivery for the 21st century**. *IEEE Power and Energy Magazine*, **3**(5):34–41, Sept-Oct 2005.

[93] Fangxing Li, Wei Qiao, Hongbin Sun, Hui Wan, Jianhui Wang, Yan Xia, Zhao Xu, and Pei Zhang. **Smart Transmission Grid: Vision and Framework**. *IEEE Transactions on Smart Grid*, **1**(2):168–177, September 2010.

[94] Juan Pablo Conti. **The Internet of Things**. *Communications Engineer*, **4**(6):20–25, Dec-Jan 2006.

[95] Sumit Roy, Vikram Jandhyala, Joshua R. Smith, David J. Wetherall, Brian P. Otis, Ritochit Chakraborty, Michael Buettner, Daniel J. Yeager, You-Chang Ko, and Alanson P. Sample. **RFID: From Supply Chains to Sensor Nets**. *Proceedings of the IEEE*, **98**(9):1583–1592, September 2010.

[96] Jeffrey Fischer. **NFC in cell phones: The new paradigm for an interactive world**. *IEEE Communications Magazine*, **47**(6):22–28, June 2009.

[97] Yu-Ju Lin, H.A Latchman, Minkyu Lee, and S. Katar. **A power line communication network infrastructure for the smart home**. *IEEE Wireless Communications*, **9**(6):104–111, December 2002.

[98] Zach Shelby, Klaus Hartke, Carsten Bormann, and Brian Frank. **Constrained Application Protocol (CoAP)**. IETF I-D draft-ietf-core-coap-10, 2012.

[99] Sherif Sakr. **XML compression techniques: A survey and comparison**. *Journal of Computer and System Sciences*, **75**:303–322, 2009.

[100] **DTDPPM Compressor**.

[101] **Xwrt: enhanced XMill compressor**.

[102] James Cheney. **Compressing XML with Multiplexed Hierarchical PPM Models**, 2004.

[103] World Wide Web Consortium (W3C). **XML Technology**.

[104] Troy Wolverton. **ZigBee radio chips could allow remote use of home electronics**. Los Angeles Times, April 2010.

[105] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. **Hypertext Transfer Protocol – HTTP/1.1**. IETF RFC 2616, 1999.

[106] Matus Harvan and Jurgen Schoenwaelder. **TinyOS Motes on the Internet: IPv6 over 802.15.4 (6lowpan)**. *PIK - Praxis der Informations - verarbeitung und Kommunikation*, **31**:244–251, 2008.

[107] **CoRE mailing-list Archives**.

[108] **EXIficient: an Open Source Implementation of the W3C Efficient XML Interchange (EXI) Format Specification**.

[109] Tim O'Reilly. **What Is Web 2.0**, 2005.

[110] Michele Zorzi, Alexander Gluhak, Sebastian Lange, and Alessandro Bassi. **From today's INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view**. *IEEE Wireless Communications*, **17**(6):44–51, December 2010.

[111] Nicola Bui and Michele Zorzi. **Health Care Applications: A Solution Based on The Internet of Things**. In *ISABEL*, Barcelona, Spain, October 2011.

[112] San Murugesan. **Understanding Web 2.0**. *IT Professional*, **9**:34–41, July 2007.

[113] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. **Uniform Resource Identifier (URI): Generic Syntax**. RFC 3986 (Proposed Standard), January 2005.

[114] Anne van Kersteren. **Cross-Origin Resource Sharing**. W3C Working Draft, 2010.