UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

Tesi di Laurea Magistrale in

INGEGNERIA INFORMATICA

RESTful management system for Wireless Sensor Network devices

Relatore

Prof. Michele Zorzi

Correlatore

Dott. Angelo P. Castellani

Correlatore

Dott. Moreno Dissegna

Laureando

Michele Mattiuzzo

Anno Accademico 2010/2011

Alla mia famiglia,
a Veronica,
a tutti i miei più cari amici.

Abstract

The thesis has been conceived at the SigNET laboratory of the University of Padova. The project has consisted on the design and the development of a software system for managing the devices that compose a Wireless Sensor Network. The created software is based on the REST architectural style, so it has been designed to provide the access to the network functionalities by offering web resources to the clients. The heart of the project is focused on the management of the elaborative nodes of the network. These devices have a direct control on the sensor nodes. The developed software runs on them as an internal daemon, with the aim of monitoring the state of the sensors and managing the requests of the users. The project has also included the creation of a Web Application with which integrate the functionalities provided by WebIoT, a multi-platform Web Application created by the SigNET research group. The aim of this part of the work was making the system transparent to the users, who can thus make requests on the available resources through a complete web interface. All the implementation choices have been made with a particular attention to ensure high levels of modularity, portability and scalability.

Contents

Chapter 1

Introduction

Abstract: In this chapter are presented some concepts about the Wireless Sensor Networks (WSN) and the Internet of Things (IoT). Finally will be discussed the purpose of the thesis by describing the main aims of the project.

1.1 Wireless Sensor Networks

A Wireless Sensor Network can be defined as a network of devices, denoted as nodes (or motes), which can sense the environment and communicate the information gathered from the monitored field (e.g., an area or volume) through wireless links [1]. The data is forwarded to a sink node (sometimes denoted as controller or monitor) that can use it locally or is connected to other networks (e.g., the Internet) through a gateway. The nodes can be stationary or moving. They can be aware of their locations or not. They can be homogeneous or not. Monitoring and communication are performed cooperatively by the nodes [1]. Some of the most widely used sensors that can be applied on the nodes are able to determine the level of luminosity, temperature or humidity of the environment, making possible to a large set of applications to store and use the collected data. The nodes are also equipped with RAM, ROM, a radio chip and a USB connector. The latter replaces the use of batteries, so that to obtain lower costs in the management of the power supply. The node firmware can be replaced using wired or wireless reprogramming, but by using USB it's possible to achieve better performance. The sinks are usually connected each other in an Ethernet-based LAN. A single-sink approach to the management of the WSN is possible but, clearly, it is not preferable to the multi-sink one, as it suffers from the lack of scalability: by increasing the number of nodes, the amount of data gathered by the sink increases and once its capacity is reached, the network size can not be augmented [1]. In principle, a multi-sink WSN can be scalable (i.e.,

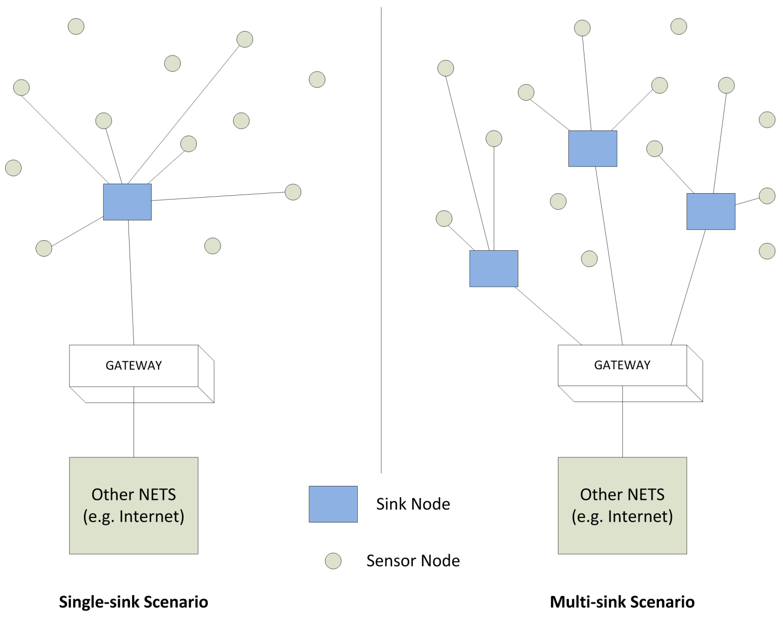


Fig. 1.1: Comparison between the SINGLE-Sink and the MULTIPLE-Sink approaches.

the same performance can be achieved even by increasing the number of nodes), however, a multi-sink WSN does not represent a trivial extension of a single-sink case for the network engineer [1]. The communication among sinks requires the development of specific protocols, made on the basis of a suitable criterion, such as, for example, minimum delay, maximum throughput, minimum number of hops, etc..
The nodes of a WSN can be distributed in a very wide environment, so the implementation and evaluation of sensor network applications, middleware and communication protocols is a very difficult task. Unfortunately, real experiments with distributed systems like sensor networks quickly become very cumbersome if the number of nodes exceeds a few dozens. In fact, all the phases of the experiment: deployment of the nodes in the desired, possibly heterogeneous and hierarchical, configuration; making changes in the software of individual nodes; and last but not least, conducting exper-

iments which include both data processing and self-reconfiguration of the network are very, very difficult without a targeted, specialized support [2]. For all but the smallest experiments a dedicated infrastructure supporting the above listed steps is necessary. This infrastructure - from now on referred to as testbed - makes it possible to create, modify and observe the target configuration (both hardware and software) in its whole complexity including nodes, communication protocols, middleware and application [2]. In the world a lot of testbeds have been designed and deployed, showing similarities and differences. In the last years even the SigNET group of the University of Padova, within the WISE-WAI project [3], has set up a large testbed, covering most of the department's buildings. This testbed and some others will be presented in Chapter 2.

1.2 Internet of Things

To date, the vast majority of Internet connections worldwide are devices used directly by humans, such as computers and mobile handsets. The main communication form is human-human. In a not distant future, every object can be connected. Things can exchange information by themselves and the number of "things" connected to the internet will be much larger than the number of "people" and humans may become the minority of generators and receivers of traffic . We are entering a new era of ubiquity, we are entering the Internet of Things era in which new forms of communication among human and things, and things themselves will be realized [4]. The Internet of Things is a technological revolution that represents the future of computing and communications, and its development needs the support from some innovative technologies [4]. Even though we can connect anything does not mean things can communicate by themselves. So new smart things should be created which can process information, self-configure, self-maintain, self-repair, make independent decision, eventually even play an active role in their own disposal. Things can interact, they exchange information by themselves. So the form of communication will change from human-human to human-thing to thing-thing [4]. In this context, the research focuses on finding new solutions to make things intelligent, or at least able to store and share information. A cornerstone in this scope is undoubtedly the RFID technology. Radio Frequency Identifiers (RFIDs) were first introduced to overcome the limitations of the barcode technology and primarily focus on tagging objects by attaching an individual identifier to them [5]. While the original idea was to tag items for retail and logistics, it is foreseen that the application of RFID tags to any object around us will open up the possibility to develop a huge number of disruptive services [6]. Although some privacy concerns have been raised, RFIDs have become part of our everyday life. Despite the recent improvements in RFID technology (e.g., in terms of miniaturization), further developments need to be realized, especially in the areas of energy harvesting and batteries, integration into materials, and cost [6]. The Wireless Sensor Networks have been recognized as very suitable systems for the management of these issues. 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 [7].

1.3 Representational State Transfer

The architectural style underlying the Web is called Representational State Transfer or simply REST. REST answered the need of the Internet Engineering Task Force (IETF) for a model how the Web should work. It is an idealized model of the interactions within an overall Web-application. Roy T. Fielding defines REST in [8] as a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.
The most interesting aspect of REST, regarding the thesis project, is the concept of "resource". Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los An-

geles"), a collection of other resources, a moniker for a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource [8]. A resource can also be defined as a conceptual mapping to a set of entities. Only the semantics of a resource are required to be static, entities may change over time. For example in a version control system the Version 1.2 always points to the same entity whereas HEADs entity changes over time. This concept allows an author to reference a concept instead of a single representation [9].
REST uses resource identifiers to distinguish resources. In a Web-environment the identifier would be an Uniform Resource Identifier (URI) as defined in the Internet RFC 2396 [10]. All REST components perform actions on representations of resources. Representations capture the current or the intended state of a resource and can be transferred components [9].
A representation consists of:

- a sequence of bytes (the content)
- representation metadata (describing the content)
- metadata describing the metadata (e.g. hash sums or Cache-Control)

REST is not tied to a specific data format as long as all components can process the data. A intermediary cache, for example, does not need to know the semantics of the data, only if the request or response is cacheable or not. The data format of a representation is called media type. Some media types can be processed by computers, others are intended primarily for the human reader and few can be automatically processed and viewed by a human reader [9]. In modern Web-services a commonly used representation is XML (Extensible Markup Language) as defined by the World Wide Web Consortium (W3C) in [11].
The REST architectural style is suitable for developing applications for WNSs, since the most common needs of the WSN users, like getting information on the nodes, reprogramming them or switching them on and off, can be seen as resources that can be requested to the system.

1.4 Purpose of the thesis

The thesis has been conceived at the SigNET laboratory of the University of Padova. The SigNET group is in front line in studying and developing solutions and protocols for the WSNs. The project has consisted on the design and the development of a software system for managing the devices that compose a Wireless Sensor Network. The generic term ”device” has not been chosen by chance, indeed the project intends to cover the high-level management of all the components of a WSN, starting from the motes, going through the sink nodes and finishing with the client side.
The developed software is based on the REST architectural style, so it has been designed to provide the access to the network functionalities by offering resources to the clients. The heart of the project is focused on the management of the sink nodes. These devices have an important relevance in the economy of the WSNs, since they represent a sort of bridge from the motes to the outside. The created software will be executed on them as a daemon. Each instance is able to keep up-to-date a local list of connected motes, which permits storing all the useful information about the motes and their states, to share this list with the other units, to accept requests on the available resources, to forward the requests through the network and, last but not least, to provide clients an updating service.
The project has also included the creation of a Web Application with which integrate the functionalities provided by WebIoT, a multi-platform Web Application created by the SigNET research group(see Chapter 2 for further information). The aim of this part of the work is to make the system transparent to the users, who can thus make requests on the available resources through a simple and complete interface. The Web Application is able to interact with the daemon. Thanks to the resource offered by the sink nodes, the clients are able to arrange requests, to manage responses and to update their information on the WSN. The produced software will be used to facilitate the management of the WISE-WAI testbed, replacing and integrating parts of the currently used system. This does not mean that it cannot be used also to manage other kind of WSN. The aim is precisely to reach high levels of portability, modularity and scalability, by dedicating a particular attention to the design phase.

1.5 Document structure

Next chapters are organised as follows. In Chapter 2 are reported the descriptions of some testbed architectures and a brief introduction to WebIoT. All the system design and implementation aspects will be discussed in detail in Chapter 3, while a description of the executed tests and the obtained results is given in Chapter 4. The conclusion is given in Chapter 5.

Chapter 2

Related work

Abstract:
In this chapter are described three different testbed architectures: the WISE-WAI testbed, which has been developed at the University of Padova and it is where this thesis takes place, TWIST, designed and deployed at the University of Berlin, and finally MoteLab, developed at the Harvard University. Within the section dedicated to the WISE-WAI testbed can also be found a briefly introduction to the WebIoT project, which is strictly correlated to a part of the thesis work.

2.1 The WISE-WAI testbed

The "Wireless Sensor networks for city-Wide Ambient Intelligence (WISE-WAI)" project [3] aims to demonstrate the feasibility of large-scale wireless sensor network deployments, whereby tiny objects integrating one or more environmental sensors (humidity, temperature, light intensity), a microcontroller and a wireless transceiver are deployed over a large area, which in this case involves the buildings of the Department of Information Engineering at the University of Padova [12]. The overall objective of the WISE-WAI project is precisely to exploit the potential of WSNs by designing and evaluating a system architecture for a flexible, heterogeneous, and energy-efficient network, including the specification of applications. The WISE-WAI project also aims at deploying a wireless sensor network testbed on a large scale, that will be employed to simulate deployments of a large number of nodes over a wide territory [12].

In order to implement the above concepts and provide a flexible and reconfigurable platform for testing algorithms and solutions for WSNs, the set up testbed includes wireless nodes as well as networking devices that allow fast communication with the sensors (e.g., for reprogramming purposes). Every node is connected to the network backbone through a Universal Serial Bus (USB) cable, which also provides power supply: this avoids battery wastage and continuous replacements during setup and test phases. During actual operations, however, 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 [12] . The WISE-WAI testbed, whose architecture is shown in figure

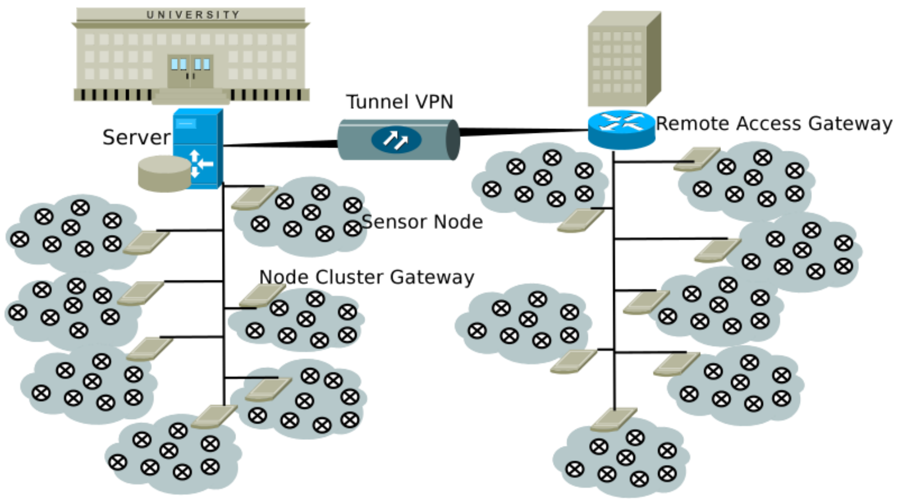


Fig. 2.1: The WISE-WAI Testbed architecture [12].

2.1, is composed by approximately 350 sensor nodes, which are connected, via USB hubs, to tiny embedded computers that act as Node Cluster Gateways (NCGs). The NCGs are in charge of interacting 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 reprogram, reset, and power up or down the nodes, or to adapt the node behavior on the fly as needed by ongoing measurement tasks. The gateways are connected via an Ethernet backbone to a local Remote Access Gateway (RAG), that manages local communications within specific portions of the network. All remote access gateways are finally connected to a central server through Virtual Private Network (VPN) tunnels. The server is the main point of access for the communication to, and the management of, the whole WSN [12] . The NCGs, which cover the role of sink nodes, are organized hierarchically. The described architecture is scalable, easy to replicate in case the network needs to be extended, and in addition its components can easily be reached and replaced for maintenance. In particular, it should be noted that the full USB 2.0-compliant hubs employed in the testbed allow a sort of hard sensor reset, which is accomplished by powering off the port to which the sensor is connected. Thanks to this function, the sensors need not be manually disconnected, in case they do not respond to software reset commands. For all these reasons, dividing the network into smaller subsets that are managed through gateways, while using embedded PCs and USB hubs to ease remote control, provides a better solution. In particular, NCGs are a key component of the network hierarchy. They are small computers of the ALIX [13] series. These computers exploit the Power-over-Ethernet (POE) standard technology [12], they are equipped with USB interfaces, an Ethernet socket, a CPU, RAM, a flash memory and a PCI slot. Every ALIX is connected to 1, 2 or 4 USB hubs, which make available 4 ports. The running Operating System is the Voyage [14] 0.5.2 distribution, where the Linux kernel (version 2.6.26) has been patched to make available further functionalities, as the possibility to interact with the USB hubs. Finally, the

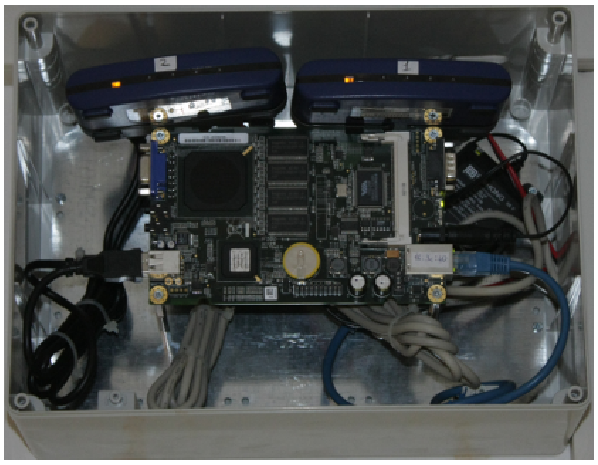


Fig. 2.2: A Node Cluster Gateway [3].

wireless embedded sensors chosen for the testbed are the TelosB nodes [15]. TelosB is a prototyping WSN platform that comes equipped with temperature, humidity and light sensors. They can be directly connected to other

devices through an embedded USB port. A wireless connection is also directly available through an implementation of the ZigBee protocol stack. TelosB nodes use the CC2420 radio chip for ZigBee-compliant communications in the 2.4 GHz band, in accordance to the IEEE 802.15.4 standard. Their maximum transmission power is 1 mW within the 2400-2480 MHz bandwidth. Their transmission rate of 250 kbps is foreseen to be enough to support all wireless sensor network applications that have been considered in the WISE-WAI project [12].
The main part oft the software produced during the thesis work is a daemon that will be run on the NCGs. The daemon will integrate and extend the capabilities of the testbed, providing a dynamic tool for monitoring and reconfiguring the network. The software can be thought as a distributed web server, where the various units collaborate in order to manage the information about the testbed status and satisfy the external requests.

2.2 WebIoT

The WebIoT project, developed by the SigNET group, aims to provide a HTTP-based Web Application for managing testbeds. Through a GUI (Graphical User Interface) which exploits Google Maps to create the lowest graphical layer, the clients can see all the motes deployed in the testbed and their spatial position. The clients, interacting with the application, can submit performance tests (such as testing the routing performance) or control the motes, for example they can turn motes on/off or inspect the software installed on them. It should be noted that the presence of the NCGs is completely hidden to the users, who perceive instead the network as a whole. In figure 2.3 is reported a screenshot of the main page of Fandango, the WebIoT release specific for the WISE-WAI testbed.

WebIoT stores the information about the WSN in a MySQL Database, to which it connects using the JDBC (Java DataBase Connectivity) API. WebIoT has been developed using GWT (Google Web Toolkit) [16], which is a powerful tool for the creation of complex web-oriented applications. GWT allows the programmers to use the same language, i.e. Java, for developing both the serve-side and the client-side, effectively substituting the using of technologies as ASP, PHP, JSP or AJAX. The client-side part of the code is automatically translated into JavaScript and HTML, while the server-

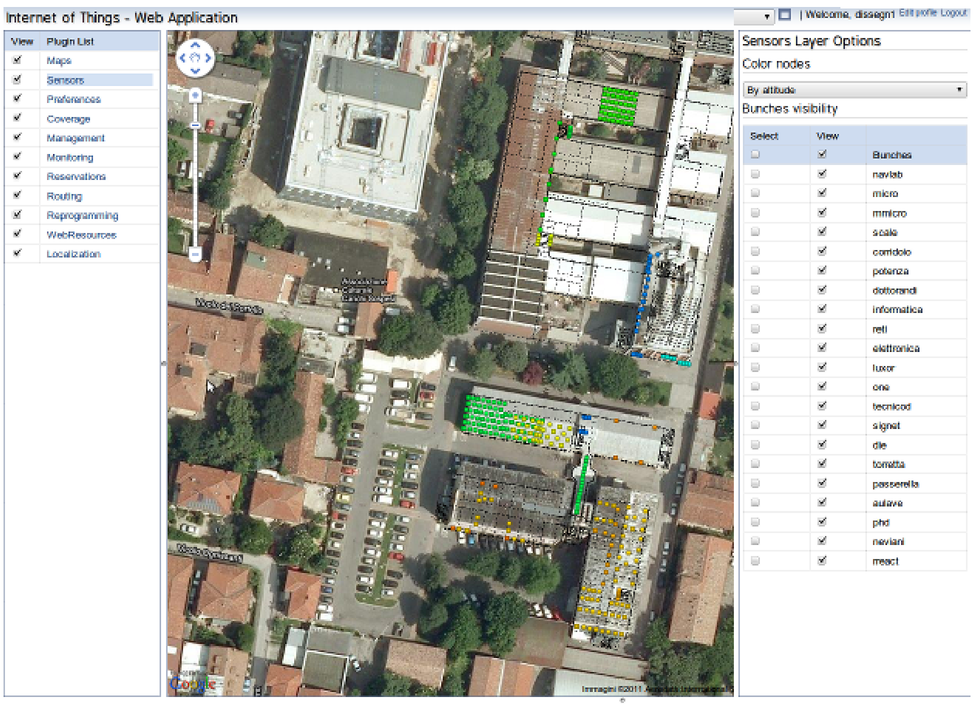


Fig. 2.3: Screenshot of the WebIoT Home Page.

side part is left in Java, since most of the web servers supports the JSP technology.

2.3 TWIST

In [2] the authors describe TWIST (TNK Wireless Indoor Sensor network Testbed), a scalable and reconfigurable testbed for wireless indoor experiments with Sensor Network. TWIST is based on cheap off- the-shelf hardware and uses open-source software. Figure 2.4 depicts the hardware architecture of TWIST. The sensor nodes expose suitable hardware interfaces that supports external powering, reprogramming, as well as out-of-band exchange of configuration, debug and application data. All those features are made available by the use of the USB interface. On the software side, the operating system running on the sensor nodes satisfies several basic requirements. First, it provides a suitable execution environment for the application logic of the SUE (System Under Examination). Secondly, it supports node configuration, instrumentation of the application code and allow for out-

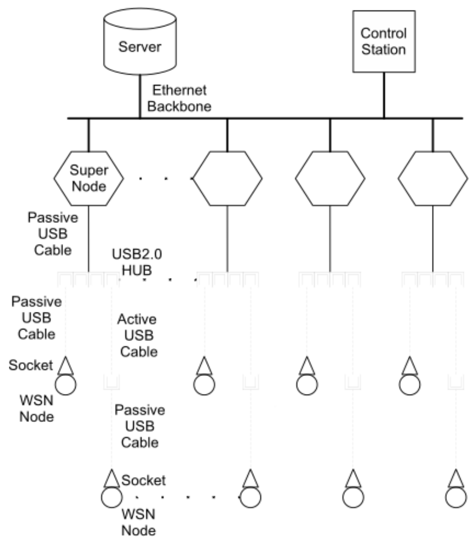


Fig. 2.4: The TWIST's architecture [2].

of-band communication with the super nodes over the USB infrastructure. The adopted operating system is TinyOS [17], which provides a generic and lightweight execution platform for sensor network applications. The testbed sockets, seen as plain hardware, are nothing more than the points where the USB interface of the sensor node attaches to the USB infrastructure of the testbed. The architectural significance of these points is, however, much greater. The sockets have unique identifiers, and their geographical position is known and does not change over time, thus it is possible to associate the node identifiers to the socket identifiers and hence to their geographic position. The USB hubs give TWIST one of its most powerful capability: the binary power-control over its ports. This makes possible to remotely turn on and off the sensor nodes in the testbed. In order to ensure scalability to the system, the sensor nodes are connected to super nodes, small computers which support a secondary communication technology that forms the testbed backbone. Several Python scripts run locally in the super nodes so as to provide the various testbed functionalities, like sensor nodes programming, executing power-control, collecting debug and application data, and more. The effective management of a large number of super nodes requires self-configuration capabilities. For the most basic system parameters like the super node IP address and the DNS server address, TWIST relies on the DHCP (Dynamic Host Configuration Protocol) protocol. Time synchro-

nization is achieved using NTP (Network Time Protocol). Because of the limited flash memory on the super nodes, the root file-system is provided over the network using NFS (Network File System). The server and the control station interact with the super nodes using the testbed backbone, so they support the same communication technology. Due to the critical role of the server (it contains the testbed database, provides persistent storage for debug and application data from the SUE, runs the daemons that support the system services in the network, etc.) its hardware resources are adequately dimensioned to guarantee high levels of availability. At the heart of the server is the PostgreSQL database that stores a number of tables including configuration data like the registered nodes (identified by the NodeIDs), the sockets and their geographical positions (identified by the SocketIDs) as well as the dynamic bindings between the SocketIDs and NodeIDs. The database is also used for recording debug and application data from the SUE. The control station, which is attached to the backbone, is in charge of activate the Python scripts running on the super nodes. To speed up this task the control station exploits the multithreading paradigm, creating a separate thread for each of the super nodes. Every such thread then starts the Python scripts on its associated super node via the ssh remote command execution. This permits parallelizing the operations over the sensor nodes, obtaining largely reduced execution times.
There are several common issues between the TWIST and the WISE-WAI projects. First of all, the testbed architectures are very similar, specially about the roles of super nodes and NCGs. Here the main difference is that the TWIST control station functionalities, in the case of the WISE-WAI testbed, are delegate to the server. Both the systems make use of a database with which store the information on the sensor nodes. Furthermore, each testbed relies on a backbone network that separates the SUE data from those of the testbed. The principal difference might be that TWIST, unlike the WISE-WAI project which includes the development of the WebIoT application, does not provide users with a web-based interface to the system.

2.4 MoteLab

MoteLab, a Web-based sensor network testbed developed at Harvard University, is described in [18]. MoteLab consists of a set of permanently- deployed sensor network nodes connected to a central server which handles reprogramming and data logging while providing a web interface for creating and scheduling jobs on the testbed. MoteLab accelerates application deployment by streamlining access to a large, fixed network of real sensor network devices; it accelerates debugging and development by automating data logging, allowing the performance of sensor network software to be evaluated offline. Additionally, by providing a web interface MoteLab allows both local and remote users access to the testbed, and its scheduling and quota system ensure fair sharing. The sensor nodes are connected each other through a

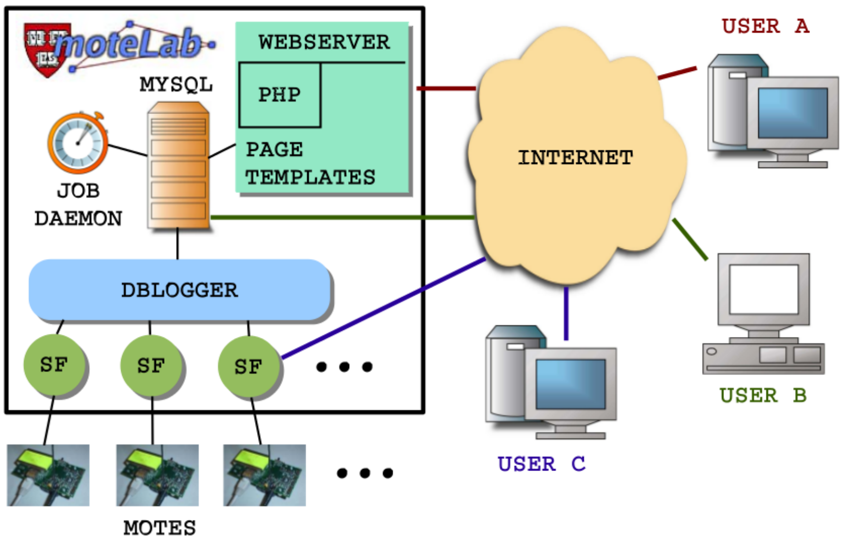


Fig. 2.5: The MoteLab's architecture [18].

Ethernet network, using Crossbow MIB-600 hardware interfaces. A central server handles scheduling, reprogamming nodes, logging data, and providing a web interface to users. Users access the testbed using a web browser to set up or schedule jobs and download data. MoteLab consists of several different software components. The main pieces are:

- MySQL Database Backend : Stores data collected during experiments, information used to generate web content, and state driving testbed operation.

- Web Interface : PHP-generated pages present a user interface for job creation, scheduling, and data collection, as well as an administrative interface to certain testbed control functionality.

- DBLogger : Java data logger to collect and parse data generated by jobs running on the lab.
- Job Daemon : Perl script run as a cron job to setup and tear down jobs.

Each of the mentioned jobs consists of some number of executables and testbed nodes, a description mapping each node used to an executable, several Java class files used for data logging, and other configuration parameters, such as whether or not to perform power profiling during the experiment. Users can submit the jobs using the web interface. The job daemon is in charge of schedule the jobs and start them when possible. During a job execution users can access the motes via the Ethernet backbone. The produced data are available both via the web interface and via direct connections to the MySQL Database.
MoteLab shows different architectural choices than the other two tesbeds. MoteLab sensor nodes are connected via Ethernet, thus it is not possible to remotely manage their power supply. There are no super nodes, and the tasks of which the TWIST control station is in charge are left to the server. MoteLab focuses the emphasis on the web interface and the management of the user jobs. The system is built so as to serialize the access to the testbed resources, but on the other hand the awakening of the scheduled job is done automatically.
The thesis project, which aims to integrate the WISE-WAI testbed functionalities, tries to catch the vantages introduced by the different solutions, reinterpreting the role of the NCGs in a dynamic way.

Chapter 3

Design and implementation

Abstract:
In this chapter firstly will be described the system architecture and the design principles which have been adopted during the creation of the software. In the following sections will be discussed all the implementation choices.

3.1 Design Principles

Within the design phase of the project have been emerged some guide lines. They have been adopted during the implementation of the software. Here they are discussed in a global perspective.

1. Modularity: both the daemon and the Web Application have to be organised in a way that allows to easily add new functionalities to the system without change the core structure. In addition, more modular is a software more it is easy to debug and maintain, and the readability of its code sensibly improves. A good level of abstraction for what concerns the key-concepts of the context (e.g. ”mote”, ”sink node”, ”resource”, ”operation”, ”module”, ”updating service”, etc.) should make simpler to achieve this aim.

2. Usability: interacting with the system should be simple, both in making requests and in interpreting the responses, and the clients should be able to access to a large set of resources. The choice of using XML for interchanging data certainly helps to achieve this aim.

3. Scalability: it concerns the system’s capability to maintain unaltered

its performance when its size increases, or when it receives requests involving a growing number of nodes. In order to reach a good level of scalability it's important to keep in mind the following issues:

- minimize the number of information messages the sink nodes, so that the network traffic is limited;
- maximize the parallelism in executing the required operations;
- avoid the creation of bottlenecks.

4. Portability: it should be possible to deploy the system independently from the sensor network and the type of the sink nodes. If the software architecture shows a good level of modularity, then the implementation and the integration of network-specific modules should be less difficult, with the possibility to make the system more portable.

3.2 System architecture

3.2.1 Deployment

The purpose of the thesis has already been described in a general way in Chapter 1, in this section it will be presented more in detail. First of all will be discussed the system as a whole.

In Figure 3.1 it's reported the deployment UML diagram of the final solution, where it is possible to see the various links between the components of the system. This diagram refers to the deployment of the software within the WISE-WAI testbed. As shown in figure, users can interact both with the daemons, which run on ALIXes, and WebIoT, which runs on the WISE-WAI server. The daemons are able to manage connections from several clients and a client can connect to all the ALIXes. Clients, using the Web Application provided by WebIoT, a web browser or any other suitable software, make HTTP requests on the resources offered by the daemons. All the data transmitted to or received from the daemons are in XML: this choice ensures a simple and standardized way to access and exploit the system. The ALIXes are connected in an Ethernet LAN and the daemons, in order to communicate each other, use HTTP requests on the available resources.

The logical network topology is hierarchical, it composes a tree where every node is aware only of his children. Every sink node has a configuration file,

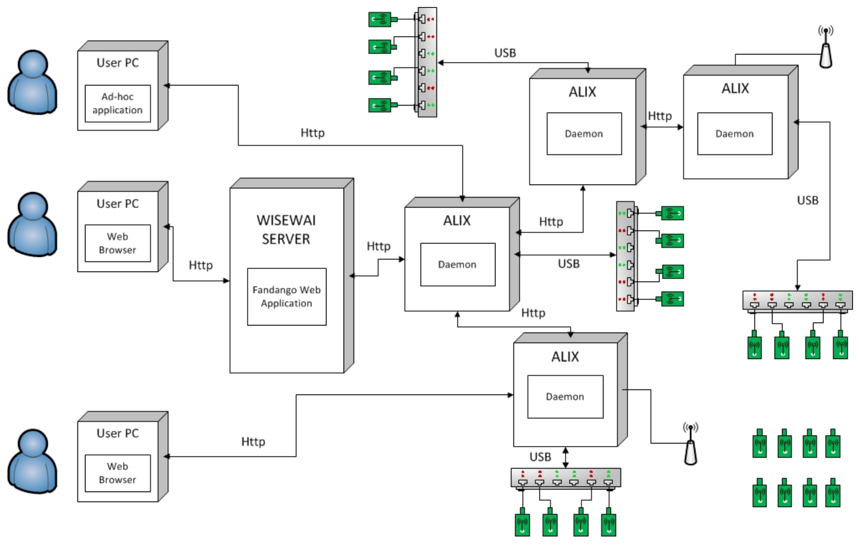


Fig. 3.1: Deployment diagram of the system.

arranged off-line before the execution of the daemon, where it can find as the principal information the list of his children in the network hierarchy.

3.2.2 Resources

In Chapter 1 have been mentioned the resources that the clients can request to the system, now they are listed more in detail.

1. list of motes: returns all the available information about the motes.
2. updating service: allows a client to keep itself up-to-date with respect to the list changes.
3. on, off: allow a client to switch on/off the USB port to which the motes are connected.
4. merge: a client can merge the information about two or more motes to obtain a single mote; this resource is useful when, for example, a mote is recognized both as a wireless mote and as a USB mote.
5. load firmware: allows to reprogram the motes with a new firmware.
6. set position: makes possible to specify a position information for a mote.

7. serial connection: creates a direct socket connection to a mote, so it enlarge the client-mote interaction possibilities.

Furthermore, the daemons are able to detect the connection/disconnection of the motes; when this happens they simply update the list and communicate the change through the updating services. Another important issue concerns the system's behaviour after a sink node crash. In this case the parent (if any) of the crashed node realizes what has happened, takes care to update its information and starts trying to restore the connection. All the implementation details about these issues can be found in Section 3.3.

3.2.3 Information management

The system maintains the information about the Wireless Sensor Network through the management of a list of data. Every list element contains information over a mote, which in turn consist in a list of identifiers and a list of features. Every id uniquely identifies the corresponding mote in the network, it is composed by a (name, value) pair, where the name indicates the type of the id (e.g. the reference number or the USB path of the mote device) and the value is a string field. A feature, unlike an id, represents a state or a capability of the mote, for example the power state, the loaded firmware code or the position of the mote. Even a feature consists of a (name, value) pair. Each sink node stores its own list, which is composed by the data about the directly connected motes and the data got from its children in the network hierarchy. The Web Application, which interacts with the sink node at the top of the tree, collects and stores the information on the global list. By exploiting the resources offered by the daemon, the application is able to change in real-time its information about the state of the WSN.
An important fact that deserves to be emphasized is the existence of a direct link among the "active" resources offered by the daemons (i.e. those which require some intervention on the physical motes, unlike the "passive" resources) and the features of the motes. If a client makes a request on one of those resources then it activates the execution of an operation on some features of the involved motes, with the result of change their values and trigger the update process.

3.3 Daemon

3.3.1 Employed technologies

The sink nodes have a crucial role in the management of large WSNs. Their tasks are to create a direct interface to the motes, to store and share all the information on them, to offer resources on which accept external requests and to reconfigure the network after modifications or crashes. As previously discussed, the sink nodes of the WISE-WAI testbed are connected in via Ethernet LAN. The HTTP protocol is very suitable for the interchange of messages over the network, especially since the aim is to create a RESTful application. The daemon, therefore, has to perform the function of web server, it has to accept HTTP requests recognizing the corresponding resources from their URI and then send back HTTP responses. To ensure a high level of usability all the payloads are made in XML. As far as concerns the system scalability, there are some interesting issues to describe. First of all, the network traffic results very limited by the adoption of Bidirectional HTTP techniques, which enable asynchronous, "server-initiated" communications from a server to a client as well as communications from a client to a server [19]. The HTTP Streaming protocol can be used for interchanging messages the sink nodes. The basic idea on which it relies is to keep a request open indefinitely. The HTTP streaming mechanism never terminates the request or closes the connection, even after the server pushes data to the client. This mechanism significantly reduces the network latency because the client and the server do not need to open and close the connection [19]. The basic life cycle of an application using HTTP streaming is as follows:

1. The client makes an initial request and then waits for a response.
2. The server defers the response to a poll request until an update is available, or until a particular status or timeout has occurred.
3. Whenever an update is available, the server sends it back to the client as a part of the response.
4. The data sent by the server does not terminate the request or the connection. The server returns to step 3.

The HTTP streaming mechanism is based on the capability of the server to send several pieces of information in the same response, without terminating the request or the connection [19]. Consider two sink nodes, called A and B, where A is the parent of B within the network topology. A asks B to establish an updating service connection by requiring the corresponding resource offered by B. The latter then responds with a HTTP Streaming connection, which can be set up by using some specific HTTP headers. B will send chunks of data every time happens a list modification, while A will be ready to accept and elaborate them. In this way it is possible to reduce the number of connections in the network, obtaining all the already discussed vantages. The main drawback of this protocol is about the possible presence of network intermediaries. The HTTP protocol allows for intermediaries (proxies, transparent proxies, gateways, etc.) to be involved in the transmission of a response from the server to the client. There is no requirement for an intermediary to immediately forward a partial response, and it is legal for the intermediary to buffer the entire response before sending any data to the client (e.g., caching transparent proxies). HTTP Streaming will not work with such intermediaries [19]. This problem can be bypassed by exploiting another bidirectional HTTP protocol, the HTTP Long Polling. Also this mechanism attempts to minimize both the latency in server-client message delivery and the use of processing/network resources [19].The clients send requests to the server, and the latter responds only when a particular event, status, or timeout has occurred. Once the server sends a long poll response, typically the client immediately sends a new long poll request. Effectively, this means that at any given time the server will be holding open a long poll request, to which it replies when new information is available for the client. As a result, the server is able to asynchronously ”initiate” communication [19]. The basic life cycle of an application using HTTP long polling is as follows:

1. The client makes an initial request and then waits for a response.

2. The server defers its response until an update is available or until a particular status or timeout has occurred.

3. When an update is available, the server sends a complete response to the client.

4. The client typically sends a new long poll request, either immediately upon receiving a response or after a pause to allow an acceptable la-

tency period [19].

Clearly this mechanism introduces more overhead than HTTP Streaming, but it has the vantage to eliminate the problem concerning the network intermediaries, so it is more suitable for the connections coming from outside the system.
Another important aspect inherent to the scalability principle is the parallelism. If the daemons perform their jobs in a sequential way then it is clear that the system will not be able to achieve good performances, especially when its size is not trivial. For this reason, the monitoring of the connected motes and the management of the client requests are done simultaneously. In this way, the information about the state of the network can be kept fresh as much as possible, without interfering with the interactions of the clients. Even the requests are performed in parallel, unless they require active resources that share one or more motes, condition which imposes that they have to be partially serialized. In order to implement the described parallelism has been adopted the multithreading paradigm. Each request, service or monitoring procedure is executed in a separate thread. Multithreading permits sharing the data structures and the computing resources among different threads, so as to simplify their interaction and to better use the CPU potential. Furthermore, it offers several techniques for manage the concurrency between the threads. The accesses to the shared data structures are serialized using semaphores, while the communications between the threads are managed by using condition variables.
As already discussed, the daemon has even the role of web server. The programming language with which implement the software, besides being efficient and suitable for low computational performance hardware, should offer libraries or built-in functions able to help the programmer in doing his job. Moreover, if there were a language that provides an integrated multithreading TCP (Transmission Control Protocol) web server it would really be the top. Well, Python [20] is the solution. Python is an Object-Oriented programming language and is based on C [21] routines, thus it combines the advantages of a fast and low-level language with those typical of the object-oriented paradigm. The Python libraries include a high-level framework based on the use of sockets, the ”socketserver” module [20]. Within the latter can be found various kind of server classes, based for example on UDP (User Datagram Protocol) or TCP, and some request handler classes,

which are instantiated by the server in order to satisfy the client requests. At the beginning of the thesis work the last stable Python version was the 3.1.3, but the daemon is compatible also with the 3.2.1 and 3.2.2 interpreter versions (higher versions have not been tested).

3.3.2 Global view

Before diving into the details of the single modules, is useful to analyse the software at a global perspective. In figure 3.2 are described the principal designed classes and some of their connections. The UML diagram of the classes is not meant to be an analytic discussion about the software architecture, rather it's conceived to highlight the key-concepts of the system. In the central part of the diagram are reported the classes for the definition of the core data structures. The "MoteList" attribute of the "ListHandler" class is the point where all the available information converge. This list is composed by "Mote" objects and is reachable from the other classes by the methods of the list handler. Every mote has one or more id and some features. Each of these is represented by a "Field" object, indistinctly from the fact that it is an id or a feature. The semantic separation is done within the mote, where are defined two different lists, one for each kind of field.
The "Resource" class abstracts the concept of REST resource. The various types of resources are represented by extensions of this class, as reported in figure 3.3. It is important to notice that the updating services are activated starting from a request on the corresponding resources.

The direct interactions with the motes are done using the interface modules. As shown in figure 3.4, every interface extends the "Interface" class, from which it inherits a private list of motes as attribute. Every private list will contribute to compose the global list managed by the list handler. It is completely transparent the existence of a superstructure where all the lists converge. In this way the various interfaces have the possibility to define the ids and the features of their local motes. This leads to get a completely configurable system, where the interfaces have to take care only of their own local lists. If two or more interfaces define the same id or feature for a mote, i.e. the mote is recognized from more than one interface, the corresponding information is aggregated at the list handler level, obtaining a single global mote.

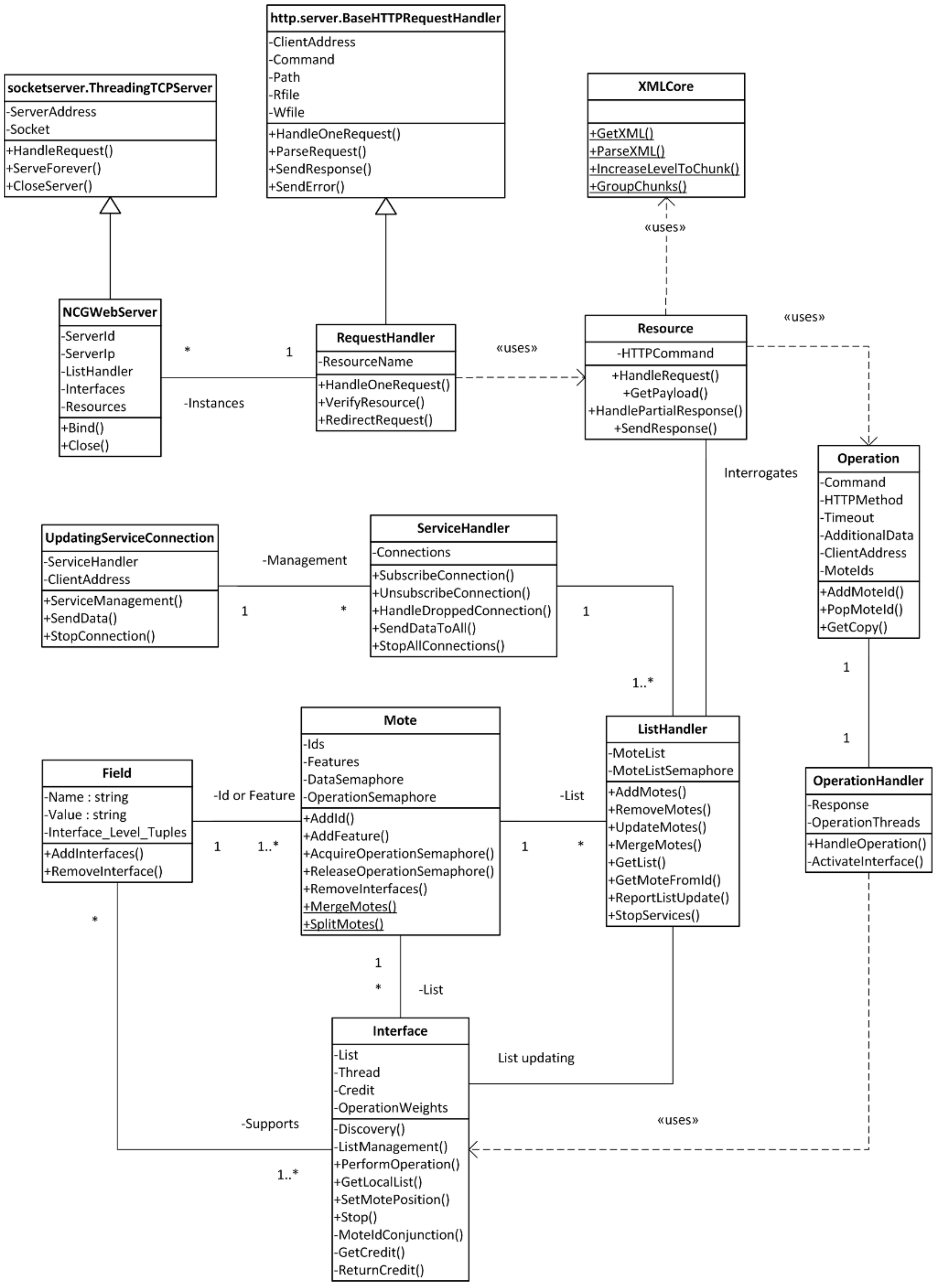


Fig. 3.2: Daemon - UML Diagram of the classes.

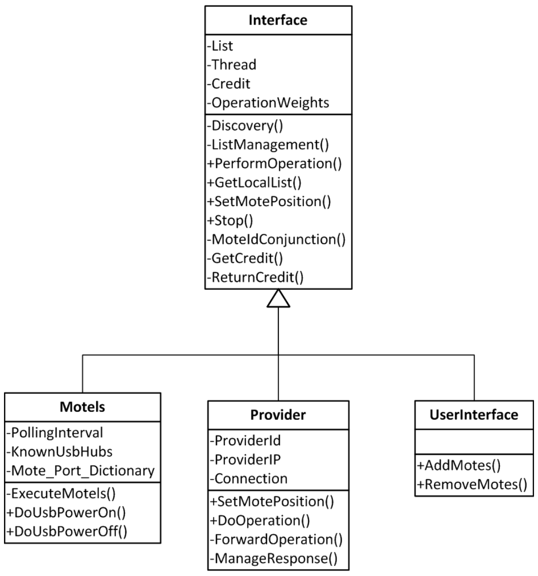


Fig. 3.3: Implemented extensions of the "Interface" class.

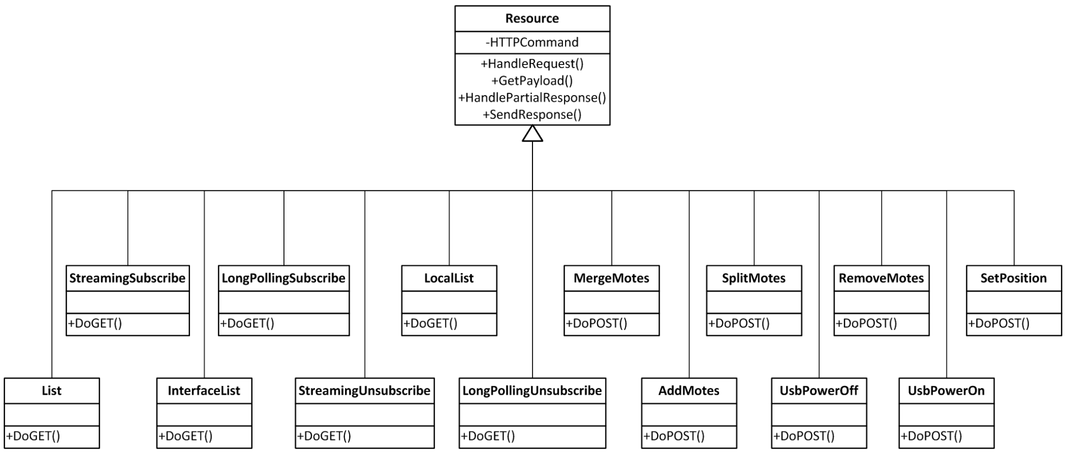


Fig. 3.4: Implemented extensions of the "Resource" class

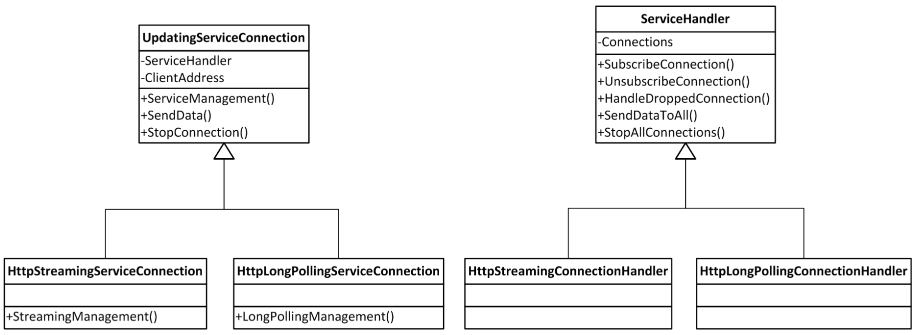


Fig. 3.5: Implemented extensions of the "UpdatingServiceConnection" and the "ServiceHandler" classes

The interfaces are also in charge of monitoring the status of the motes. This task is done by using dedicated threads. Every time the internal thread of an interface becomes aware of a modification on a mote state, firstly it updates its private list, then it reports what has happened to the list handler and it updates the global list. Finally it communicates the event to the outside through the updating services. Once these tasks have been executed, the thread returns to its monitoring activity.
The interfaces are used also for what concerns the management of the network hierarchy. Every NCG can have some children, to which it can forward requests and from which it receives the responses and the information about their list of motes. Within the configuration file every daemon can find its own list of children. Once it obtains the list from parsing the file, the daemon creates an interface for each found NCG. These interfaces use the HTTP Streaming technique to retrieve from the children the list of motes and the relative updates. Once the initial local list is created, the mechanism of interaction with the list handler is perfectly identical to that used by the other interfaces.
The client requests are accepted by an instance of the "NCGWebServer" class, which offers the functionality of web server. This class inherits some of its properties from the Python "socketserver.ThreadingTCPServer" class. In figure 3.6 is reported the life cycle of a client request. Every time a request is received from the web server, the latter creates a new thread and instantiates a "RequestHandler" object. The new thread uses the request handler to parse the client request, obtaining the URL, the HTTP command and the eventual payload. Once it has verified that the URL refers to an existing

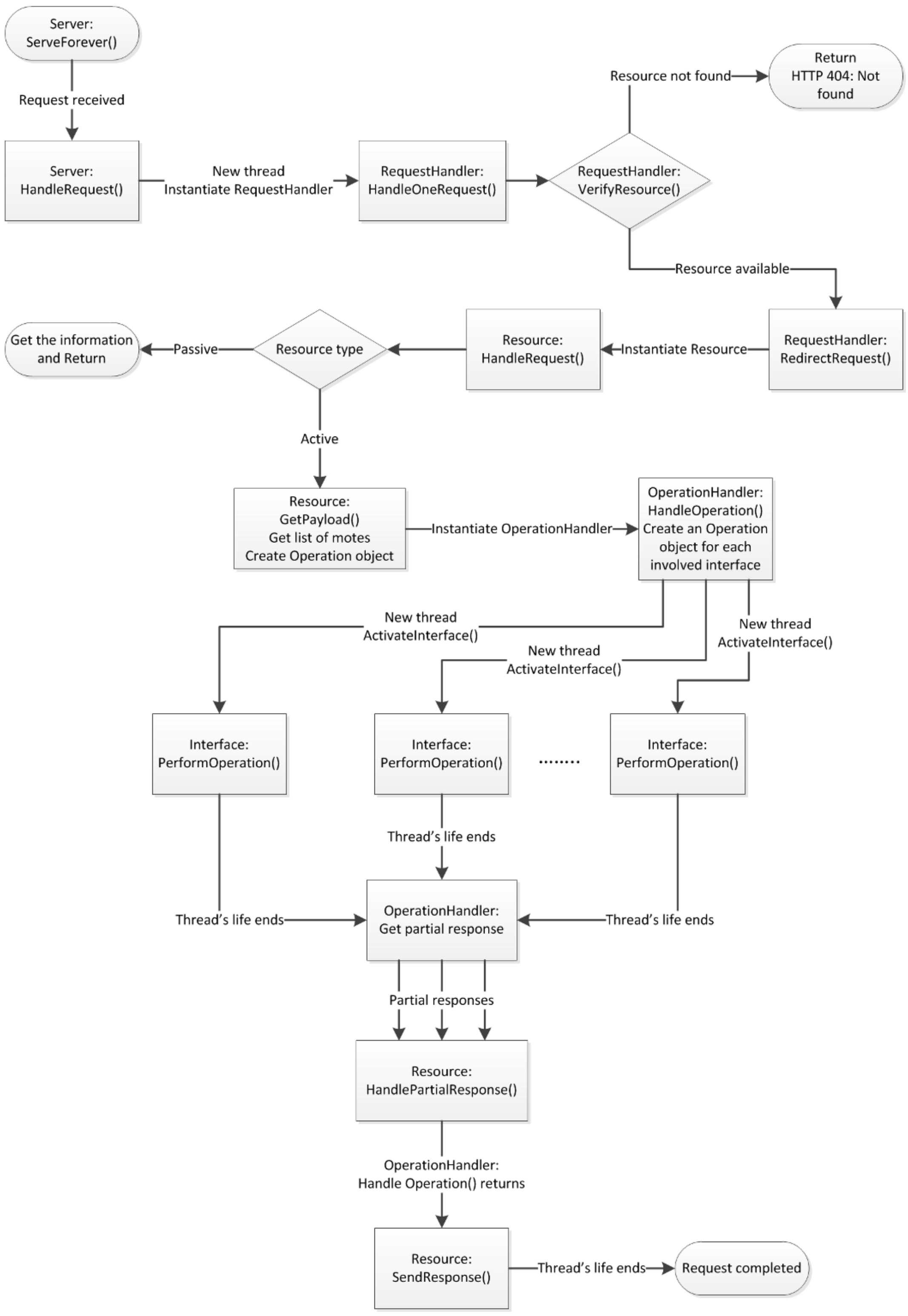

Server: ServeForever()
Request received
Server: HandleRequest()
New thread Instantiate RequestHandler
RequestHandler: HandleOneRequest()
RequestHandler: VerifyResource()
Resource not found
Return HTTP 404: Not found
Resource available
RequestHandler: RedirectRequest()
Instantiate Resource
Resource: HandleRequest()
Resource type
Passive
Get the information and Return
Active
Resource: GetPayload() Get list of motes Create Operation object
Instantiate OperationHandler
OperationHandler: HandleOperation() Create an Operation object for each involved interface
New thread ActivateInterface()
New thread ActivateInterface()
New thread ActivateInterface()
Interface: PerformOperation()
Interface: PerformOperation()
Interface: PerformOperation()
Thread's life ends
Thread's life ends
Thread's life ends
OperationHandler: Get partial response
Partial responses
Resource: HandlePartialResponse()
OperationHandler: Handle Operation() returns
Resource: SendResponse()
Thread's life ends
Request completed

Fig. 3.6: Life cycle of a client request.

resource, it instantiates the correct extension of the ”Resource” class. The resources are divided in active and passive. The passive resources are mainly used to get some information about the status of the system, while the active ones permit to intervene on the physical motes. The thread in charge of the management of an active resource instantiates the”Operation” and the ”OperationHandler” classes so that to activate the various interfaces. The operation handler is used to create a sub-list of motes and a thread for each of the interested interfaces. Each of these threads is used to perform the operation on the sub-list by calling the ”perform operation” method of the interfaces. Every time a thread finishes to execute the method it returns to the operation handler a part of the final response, which is composed by a set of ”mote - (code, message)” associations. The codes and the messages overload the HTTP semantics in order to express the possible results of the execution of an operation on a mote. The defined pairs are the following.

```
200 - Operation successful.
404 - Mote not found.
405 - Operation skipped.
408 - Timeout expired.
409 - The mote is busy.
410 - The mote is not reachable.
500 - Operation failure.
501 - Operation unsupported.
```

Every time the operation handler gets a partial response, this is returned to the resource and then is used to build up a part of the XML final response. Once the latter is ready, the thread sends it to the client by using the ”send response” method of the resource. It is important to notice that, simultaneously to the execution of the operations, the monitoring threads of the interfaces continue to be active. Every intervention on the physical motes is registered from these threads, which update the local and the global lists and report all the changes to the clients of the updating services. The module responsible to offer the methods for the XML management is called ”XMLCore”. This module has been thought as the point where grouping all the methods for the XML creation and parsing. In this way every future modification or extension to the current use of the XML can be done by changing only this module, avoiding confusion and time wasting.

Another important feature of Python is the capability of doing dynamic imports. This possibility is exploited by the daemon in its booting phase. Every interface and resource module that the daemon finds in its source folder are loaded and memorized in internal lists. This behaviour permits adding and removing the modules without changing the code.
The designed software architecture shows high levels of modularity and portability. The developers could theoretically adapt it to every kind of sensor network only extending the ”Interface” and ”Resource” classes, while maintaining the central structure. The creation of new resources permits increasing the possibilities of interaction with the motes and the system, while new interfaces can make the software suitable for other kind of networks. Furthermore, the software shows to be configurable at various levels. First of all, the definition of the information about the motes is left to the interfaces, which also offer the methods to intervene on them. In this way the clients can easily access to the system in all its capabilities. Finally, even the internal mechanism which connects the resources to the interfaces is adaptable to the different cases. The link is made up by the ”Operation” class, whose instances can be totally configured according to the various necessities.

3.3.3 Implementation choices

In this section will be described the functioning of the principal internal mechanisms of the daemon. This analysis aims to deepen the discussion about the role of the different classes, showing and giving reasons for decisions taken.
Mote and Field.
The abstraction of the concept of mote has lead to the definition of the ”Mote” and ”Field” classes. To identify a mote there are more than one ways, then is useful to store for each mote a list of ids. A mote have also some peculiarities, like the power state, the loaded firmware or the geographic position. These are called features and are memorized in a specific list for each mote. Both the ids and the features are (name, value) pairs: their structures are identical, only their roles are different. This represents the motivation to the fact that they are all modelled with the same class, called ”Field”. Every field, besides have a name and a value, has a list of (interface, level) tuples. As already said, the ids and the features of a mote

are defined by the interfaces from which it has been recognized. Since two or more interfaces could identify the same mote, every field of the motes in the global list can potentially be indicated by more than one interface, so it is necessary to store them all. When this happens there could be a conflict in the attribution of the field value. This is managed with applying the following simple concept: the last valid value is the correct value (the field should have the same value for every interface). When a mote stops to be recognized by an interface, the latter will be eliminated from the involved lists. When the list ofan id becomes empty the id is removed from the list of its mote. The features, instead, are not removed. This permits not losing the related information if other interfaces continue to recognize the mote. When the list of the ids of a mote becomes empty, then it means that it is no longer recognized by any interfaces, so the mote can be removed from the global list. The level associated to every interface is used when a client makes a request on an active resource that involves a feature. Only the interfaces that offer a method with which satisfy the request can be included in the list. The level value defines the usage priority of an interface, so that to chose the best when two or more of them are in list. The lower is the level value the higher is the priority.
A mote, besides the two described lists, offers two semaphores with which is possible to manage the thread concurrency on it. The data_semaphore is used to serialize the read/write accesses to the mote attributes, while the operation_semaphore has the function to ensure that only one operation can be executed on the mote at a time (for example there could be a "power on" request at the same time of a "power off" one). The "Mote" class also provides some static methods, including "merge motes" and "split motes". The first method is used to join two or more motes into one. This can be useful when two or more interfaces identify the same physical mote but there are no common pieces of information. The users, which are aware of the situation, can use a dedicated resource to execute the merge. The second method instead tries to divide a mote into distinct motes, i.e. motes recognized from distinct sets of interfaces.

List Handler.

The list handler stores the global list of the motes and offers methods with which intervene on it. Furthermore, it defines a semaphore for the management of the concurrency over the list and it maintains the references to the updating service handlers. The interfaces, which are not aware of the status of the global list, are in charge of interacting with the list handler in order to keep updated the information on the motes. Every interface, when its local list of motes undergoes a change, calls the correct method of the list handler and thus updates the global list. The available methods are the following.

1. add motes: it accepts a list of motes; if a mote in the list shares at least an id with one mote already present in the global list, then the new mote is merged with the global one, else it is simply added to the global list.

2. remove motes: it accepts a list of motes; every received mote is firstly individuated in the global list, secondly the reference to the calling interface is removed from each field of the found global mote. If the list of the ids of the global mote becomes empty, the mote will be eliminated.

3. update id: it accepts a list of (id, new_value) tuples; firstly for each id is searched the corresponding mote, then, once found, is executed the update.

4. update feature: it accepts a list of (list of ids, feature, new_value) tuples; in this case the interface has to indicate all the ids of the mote in order to identify it. This is required to manage an eventual simultaneous change on the value of an id.

5. merge/split motes: these methods are used to explicitly merge some motes into one or obtain as many as possible distinct motes starting from a mote that is the result of a previous merge.

All the described methods return the XML data associated to the events which they represent. These data are collected by the interfaces and then used to report the events to the outside. The list handler offers other important methods with which interact with the global list or get a link with the updating service handlers. These methods are:

1. get list: returns the whole global list.

2. get mote from id: accepts an id and returns, if any, the corresponding global mote.

3. get feature interfaces: accepts an id and the name of a feature, returns, if it finds the mote and the feature, the list of the interfaces that define the feature.

4. report list update: accepts a string which is delivered to the updating service handlers in order to send data to the requesting clients.

5. stop services: stops all the updating service connections.

As already discussed, the role of the list handler is central in the economy of the system, since it permits aggregating and managing all the information about the motes.

XML Core.

The "XMLCore" module includes all the methods for the management of the XML data. The methods are called from the other classes for the following purposes. Parsing of:

i) the payloads of the client requests;

ii) the updating data received from other daemons;

iii) the operation response data;

iv) the position information about a mote.

XML creation starting from:

i) a list of motes;

ii) a list of ids;

iii) the results of an operation;

iv) the modifications to a list of motes after the happening of an event;

v) the position information about a mote.

Furthermore, the module offers methods for specific needs of other classes. Even though the XML is used by the daemon in various contexts, the structure of the XML data maintains a good level of uniformity. Here are shown some illustrative examples of how some XML data are organized.

Response to a request on the ”list” resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<mote\_list>
    <mote>
        <ids>
            <reference level="0">XBOW0000@127.0.0.1</reference>
            <usb\_path level="0">2-2.1@127.0.0.1</usb\_path>
        </ids>
        <features>
            <usb\_power level="0">on</usb\_power>
            <firmware level="0">None</firmware>
            <position level="0">None</position>
        </features>
    </mote>
    <mote>
        <ids>
            <reference level="0">XBSM7XGM@127.0.0.1</reference>
            <usb\_path level="0">2-2.4@127.0.0.1</usb\_path>
        </ids>
        <features>
            <usb\_power level="0">on</usb\_power>
            <firmware level="0">None</firmware>
            <position level="0">None</position>
        </features>
    </mote>
</mote\_list>
```

Data produced as result of the disconnection of a mote:

```
<?xml version="1.0" encoding="UTF-8"?>
<mote\_list>
    <mote>
        <event type="3">
            <ids>
                <reference>XBSM7XGM@127.0.0.1</reference>
                <usb\_path>2-2.4@127.0.0.1</usb\_path>
            </ids>
            <old\_feature>
                <usb\_power level="0">on</usb\_power>
            </old\_feature>
            <new\_value>off</new\_value>
        </event>
    </mote>
    <mote>
```

```
        <event type="1">
            <old\_id>
                <reference>XBSM7XGM@127.0.0.1</reference>
            </old\_id>
        </event>
    </mote>
</mote\_list>
```

Response to a request on the ”usb power off” resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<mote\_list>
    <mote>
        <id>
            <reference>XBT0HG7N@192.168.0.4</reference>
        </id>
        <result>
            <code>200</code>
            <message>usb\_power:off</message>
        </result>
    </mote>
</mote\_list>
```

When level value is specified next to the name of a field it refers to the level of the highest priority interface which supports the field.

NCG Web Server and Request Handler.

Python offers a set of modules that implement various kinds of server. The more suitable for the daemon is the ”ThreadingTCPServer” class of the ”socketserver” package. The TCP protocol, since it is connection-oriented, permits to keep opened the HTTP Streaming connections. Furthermore, ”ThreadingTCPServer” founds the management of the client requests on the multitheading paradigm. ”ThreadingTCPServer” inherits its properties from the ”TCPServer” class, which in turn extends the ”BaseServer” class. The principal attributes which ”ThreadingTCPServer” inherits from the super classes are the server address (IP, port) and the socket object where accept the requests. The most important inherited methods are:

1. server bind, activate and close: these are used to bind the socket to the server address, to start the socket listening and to close it when the server is shutdown.

2. serve forever: until the shutdown request is not forwarded to the server

it waits for connections on the socket; when a request arrives it activates the ”handle request” method.

3. handle request: firstly it gets the parameters of the connection, as the dedicated socket and the client address; secondly it calls the ”process request” method that instantiates a request handler giving it the obtained parameters as arguments.

”ThreadingTCPServer” overrides the ”process request” method in order to handle every request in a different thread. The ”NCGWebServer” class, as already said, extends ”ThreadingTCPServer”. When the NCG web server is created it imports the packages of the interfaces and the resources. Python permits specifying the eventual code that has to be executed during the importing of a package. This possibility is very useful since, combined with the folder and files exploration routines, every module in the packages can be identified and imported dynamically. The references to the resources and the interfaces are maintained by the NCG web server in two dictionaries (i.e. associative arrays). After parsing the configuration file, the NCG web server instantiates the list handler and activates the interfaces. Once the initial discovery of the motes is completed, the server binds the socket to its address and becomes ready to serve clients. Every time the server receives a request, it instantiates the ”RequestHandler” class. The latter extends the ”http.server.BaseHTTPRequestHandler” class, which provides the the basic tools for handling the requests. Its principal attributes are the client address, the http command, the request path (URL) and two file object, one open for reading and positioned at the start of the optional input data part and one open for writing. Once the request handler has been instantiated by the server, the ”handle one request” method is activated in order to verify the availability of the required resource and, if so, instantiate the corresponding class. If the resource is not found the request handler sends back to the client the 404 HTTP code (”Not Found”). Notice that it is possible to get the resource name from the URL parameter, as specified by REST. When the response data become ready, the request handler calls the ”send response” method and then it closes the connection.

Resources.

Every extension of the ”Resource” class represents a REST resource which can be required from the clients. The URLs which identify the resources are composed by the IP address of the server, a slash character and the resource name (e.g. http://10.1.128.1:8000/list). The ”Resource” class defines the basic properties of the resources. The principal attributes, besides the reference to the request handler from which it has been instantiated, are the HTTP command (e.g. GET, POST, etc.) and a list of XML lines. This list is used to accumulate the partial XML responses which will be used to create the final response. The HTTP command is set up by the request handler according to the client request. The main methods of the class are the following.

1. handle request: it activates, if any, the method associated to the specified HTTP command called ”do_<command>()”. Not all the HTTP commands are accepted by all the resources, since for some of them would be meaningless. For example, the resource ”list” accepts only the GET command, while the resources which require a payload accept only the POST command. These methods are implemented by the extension classes of ”Resource”. If the method associated to the command is not available, the HTTP ”501 - Not implemented” response will be sent to the client.

2. handle partial response: it receives in input the data resulting from the execution of an operation and, by calling a method of the”XMLCore” module, it obtains the XML data with which it fills the ”xml lines” attribute.

3. read payload: it reads the eventual payload of the request and then it returns it as a string.

4. get mote list: it parses the request payload and it returns the involved motes of the global list.

5. send response: it is used to join the XML lines into a single string and then to send it to the client.

The classes that define the various resources have only to implement the methods in response to the different HTTP commands. The implemented resources are the following.

1. list: it interrogates the list handler to get the global list of the motes, it creates the XML response and then it sends it to the client.

2. list of interfaces: it returns the list of the names of the interfaces that the server has instantiated.

3. <interface name>/local list: it returns the local list of the specified interface.

4. streaming subscribe/unsubscribe: these resources are used to handle the HTTP Streaming connections. The subscription implies the creation of a new ”HttpStreamingServiceConnection” object, which is then added to the list managed by the ”HttpStreamingConnection-Handler” instance. When the server receives a ”streaming unsubscribe” request the eventual open connection is closed and removed from the list.

5. long polling subscribe/unsubscribe: these resources are used to handle the HTTP Long Polling connections. The mechanism is similar to the one used for the HTTP Streaming, with the difference that a long polling connection is closed as soon as a response is sent to the client. The latter thus has to forward a new subscribe request every time it receives data. The classes created for the management of the HTTP Long Polling are ”HttpLongPollingServiceConnection” and ”HttpLongPollingConnectionHandler”.

6. merge/split motes: users can merge the information about many motes to obtain a single mote. This makes sense when the same physical mote is recognized by two or more interfaces, but the list handler is not aware of the situation. This occurs when the interfaces do not share any information on the mote, thus it is not possible to automatically combine them. Users can also split the information on a mote into a set of distinct motes. These resources uses the list handler methods in order to intervene on the global list and to report the change to the clients of the updating services.

7. add/remove motes: the daemon offers to the users a dedicated interface (called ”user interface”) with which specify fake motes (this can be useful during the tests but it also can be used, combined with the merge resource, to add fields to the real motes). These resources use the methods of the user interface and then report the list modifications to the clients of the updating services.

8. usb power on/off: these resources permit controlling the USB power supply of the motes, switching on and off the ports of the hubs. Once the list of the mote ids given by the client is read from the request payload, the resources call the list handler methods to get the list of the interested motes. For each mote is selected the interface with the highest priority (the lower level) and it is associated to the id specified in the payload. The list of (id, interface) pairs is then used as a parameter of the "Operation" object that encapsulates the client request. The operation handler takes charge of the operation and activates the interfaces. Finally, when the "xml lines" list is completely filled, the resources compose the response and send it to the client.

9. set position: the users can change the spatial position information of the motes. The position is treated as a feature supported by several interfaces. This permits the information to be maintained in memory when a part of the interfaces stops to recognize the mote. The resource sets the position feature both of the local motes (i.e. the motes internal to the local lists of the interfaces) and of the global mote, then it reports the change to the clients of the updating services.

Updating Services.
The Bidirectional HTTP techniques, as already discussed, represent very suitable mechanisms to provide clients with updating services. A client, in order to subscribe to a service, has only to require the corresponding resource. Each connection is treated in a separate thread, so the updating services work in parallel. The "BaseUpdatingConnection" class defines the properties shared between the two alternatives. The attributes in common are the client address and the threading management variables, while the shared methods are "stop connection" and "send response and header". The "HttpStreamingServiceConnection" and "HttpLongPollingServiceConnection" instances are maintained in a list internal to the corresponding handler. The latter are instantiated by the list handler using the "HttpLongPollingConnectionHandler" and the "HttpStreamingConnectionHandler" classes. The handlers offers some useful methods:

1. subscribe connection: this method is used by the resources to add the new connection to the list or renew the existent one.

2. unsubscribe connection: it removes from the list a connection and it closes it.

3. send data to all: activates the ”send data” method of all the connections in the list.

4. stop all connections: activates the ”stop connection” method of all the connections in the list.

Follows now a description of the implementation of the two different techniques.

- HTTP Streaming: every streaming connection object execute the following internal routine: 1 send to the client the HTTP response code and the headers; 2 while the service is not closed: 2.1 wait for data; 2.2 if there are available data then send a chunk to the client; 3 send the closure chunk. Every time the ”report list update” method of the list handler is called, the streaming handler receives the request to forward the update through the connections. This mechanism ensures that as soon as a chunk is available this is sent to the clients. The HTTP headers have a particular relevance, since they are the tool with which specify the type of connection. The most important header is the ”Transfer-Encoding” one, which is set to the value ”chunked”. This tells to the client that the transferring of the response is done a chunk at a time, as the HTTP Streaming requires. When a new streaming connection is created, the first chunk that will be sent represents the complete list of motes. This ensures that the initial information is the latest as possible. If a connection comes from a client already subscribed, the old connection is replaced by the new one.

- HTTP Long Polling: every long polling connection remains in waiting until a response is available. When this happens, the response is sent to the client and the connection is closed. The long polling handler, unlike the streaming handler which uses a simple list, maintains a dictionary where the keys are the client addresses and the values are (long polling connection, list of partial responses) tuples. This makes the handler able to distinguish the old clients from the new ones, since their addresses are already registered in the dictionary. The dictionary permits also memorizing the data which become available while a registered client has yet to restore the connection. When the new request arrives the stored data are joined and sent as a single response. The first response to a new client is the whole global list, as in the case of the HTTP Streaming.

Operation and Operation Handler.
The active resources, in order to forward the client requests to the interfaces, instantiate both the "Operation" and the "OperationHandler" classes. The operation is completely configurable, since its principal role is to be an information container. The principal attributes of an operation are: a name, the HTTP command of the request, a timeout value, the client address, an "additional data" field and a list of mote ids. Each id in this list is associated to an interface, which is the one with the higher priority that supports the feature involved in the operation. The attributes are all set up by the resource which, as soon as the operation is ready, calls the "handle operation" method of the operation handler. A brief description of the functioning of this method can be found in Section 3.3.2, now it is discussed more in detail. First of all, the thread which is performing the method creates a new operation object for each interface that is associated to at least a mote id. Every id is then added to the corresponding operation's list. For each interface is then created a new thread (called "sub-thread" from now on) with which activate the "perform operation" method of the interface. While the sub-threads execute, the main thread remains in waiting on a condition variable. Each time a sub-thread returns from the interface's method it adds its part of the response to a dedicated list and it wakes the main thread. The latter then retrieves the data and calls the "handle partial response" method of the resource. The partial responses provided by the sub-threads are nothing but the dictionaries containing the operation results. Once all the sub-threads have finished their tasks the "handle operation" method returns and the main thread can proceed with the management of the response.

Interfaces.
The modularity and portability of the developed software are granted from the concept of interface. The interfaces are instantiated by the server during the booting of the system. The server waits until the interfaces communicate that their discovery phases are completed. This mechanism ensures

that when the server starts to accept the external requests the list of motes is already formed and available. The "Interface" class defines the common attributes and methods that the various interfaces will inherit. The principal attributes are the following.

1. list of motes: every interface stores its own list of motes. This leads to get some advantages. Firstly, during a monitoring cycle the comparison between the old information and the new one is done locally, i.e. without involving the global list. This fact implies that every interface can work in parallel with respect to the others, thus the creation of bottlenecks is avoided. The global list will be engaged only when occurs a local list modification. Secondly, the interfaces are not aware of the existence of the others, so the developers can focus the attention on the internal mechanisms, without worrying about the global functioning of the system. Finally, when an operation requires to intervene on a set of motes, the modifications to the global motes can be done separately from those on the local motes, reducing the time in which the global list semaphore is engaged.

2. credit and weight of the operations: every interface defines a weight for each supported operation. The weights are chosen in the real interval [0,1] and their values are maintained by each interface in an apposite dictionary. This information is used to limit the number of parallel threads that can run at the same time. The interfaces are provided with an initial amount of credit equal to one. Every time a thread wants to execute an operation on a mote, it has to get a quantity of credit equal to the weight of the operation. When the credit is no longer available, the threads have to wait until the others finish their execution and free their share. This mechanism actually permits to control the parallelism degree of the operative threads, making possible to find the system configuration that permits to get the best performance.

3. stop event: these attribute is used by the server in order to stop the activities of the interfaces.

The "Interface" class offers several useful methods. Here are described the most relevant.

1. discovery: this method is used to create the initial list of motes and report to the server that the interfaces are ready. The interfaces can override this method according to their internal mechanisms.

2. perform operation: this method is called by the sub-threads which have been activated by the operation handler. The method receives the operation object and, after the operation execution, returns the dictionary of the results. The keys of this dictionary are the mote objects and the values are the (code, message) pairs. The "perform operation" method is composed by the following steps:

   1 set up a timer which defines the available time to execute the operation (the timeout value can be obtained from the operation); when the timer expires stop all the pending activities and set the results of the remaining motes to "408 - Timeout expired";

   2 check if the interface implements the required operation;

   2.a if the method that implements the operation (from now on called "operative method") does not exist then set all the mote results to "501 - Operation unsupported";

   2.b if the operative method exists then continue to step 3;

   3 create a dictionary where each key is represented by a local mote chosen between those involved by the operation and each value is a (mote id, global mote) pair; the id is the one specified in the operation's list and the mote is the related one in the global list;

   4 check the value of the operation weight;

   4.a if the operation weight is zero then the operation is performed as a whole by the sub-thread; in this case the operative method requires in input the operation object, the list of the involved motes and the dictionary created during the execution of step 3; once its execution is finished, it returns the response dictionary and an eventual list of (mote, feature, new_value) tuples; the latter will be used to update the global motes;

   4.b if the operation weight is greater than zero:

   4.b.1 create a thread for each of the interested motes;

   4.b.2 each thread tries to get a quantity of credit equal to the weight of the operation;

4.b.3 when a thread obtain the needed credit it tries to acquire the operation_semaphore of the global mote: if it succeeds then it continues to step 4.b.4, else it sets the mote result to ”409 - The mote is busy”;

4.b.4 each thread calls the operative method, executes the operation, gives back its credit share and reports the operation result; in this case the operative method requires in input only the operation object and the local mote; once its execution is finished, it returns the (code, message) response and the eventual (mote, feature, new_value) tuple;

4.b.5 create the response dictionary starting from the single tuples;

5 update the global list if necessary;

6 return the response dictionary;

It is important to highlight the difference between the steps 4.a and 4.b. In the first one, the thread executes the operative method directly on the whole list of motes, while in the second one it creates a set of new threads. This implies that the interfaces has to implement the operative method according to the weight of the corresponding operation.

3. get credit: the calling thread specifies the amounting of credit that it needs; if the amounting is available, the method decrements the credit value and returns, else the thread waits until another frees its share.

4. return credit: the caller gives back its credit share and notifies the credit availability to all the waiting threads.

5. handle single mote: this method is used to perform the operation on a single mote. It is called by the threads created at step 4.b.1 and it implements the steps from 4.b.2 to 4.b.4.

6. set mote position: the information about the position of a mote, unlike the data obtained from the interfaces, comes directly from the users. This implies that all the interfaces that support the associated feature have to store the same value. The ”set mote position” method is thus made available for all the interfaces.

The following paragraphs describe the implemented interfaces.
MoteLs.
The interface called ”MoteLs” is in charge of the management of the USB-connected motes. The interface owes its name to the Perl script used to recognize the motes. The monitoring is continuously done by an internal dedicated thread. When the thread notices a mismatch between the state of the motes and the stored information, it modifies the local list and it reports the change to the list handler. Furthermore it forwards the XML data obtained by the list handler to the clients of the updating services. The interface makes available the methods to perform the operations of turning on and off the ports of the USB hubs that control the power supply of the motes. Other operative methods, like the reprogramming of the motes, can be easily added in future. The principal attributes that the interface defines are the following. The base attributes, as previously said, are inherited from the ”Interface” class.

1. polling interval: it sets how frequently the internal thread has to execute the monitoring of the motes.

2. known USB hubs: the daemon maintains a text file in which it stores the ”productId” and the ”vendorId” parameters of the USB hubs that implement the power management functionality. These ids are stored in a list of tuples.

3. mote-port dictionary: this dictionary is used to memorize the associations between the motes and the USB ports to which they are connected. The information about the known USB hubs is used to recognize if a mote can be remotely switched on and off or not. If between an ALIX and a connected mote there are some intermediaries, these could not provide this functionality. In this case the associated port is set up as the port of the last known hub. If the mote is not connected through a known hub, the interface will not support the management of the power supply for it.

The main methods implemented by the interface are the following.

1. execute motels: this method is used to execute a Perl script which searches through the registered devices if there are connected motes

and then it creates and returns the retrieved information. This script has been developed as a part of the old management software.

2. create new mote: each mote recognized from the script has to be registered in the list. This method retrieves further information about the mote, as the USB port to which is connected, and returns the "Mote" object with all the parameters set up. The tuple ("MoteLs", 0) is also added to the list of (interface, level) tuples of every field of the mote, except for the fields that can not be modified by the interface. For example, if the mote is not connected to the ALIX using a known USB hub, the feature named "usb power" will not be supported by the interface.

3. discovery: it calls the "execute motels" method, it creates the initial local list, it adds the motes to the global list and it notifies the server. Finally it activates the monitoring thread.

4. list polling: it activates the "list update" method every "polling interval" seconds.

5. list update: this method implements the central mechanism of the interface. In order to be more comprehensible, the procedure will be described through a list of steps.

   1 create the lists for the motes that will be added("add list"), removed ("remove list") and updated ("update id list" and "update feature list"); fill the "remove list" with all the old motes;

   2 call the "execute motels" method and obtain the current state of the motes;

   3 compare the old list with the new information and fill the lists; this step analyses every mote recognized in step 2: if the mote is already in list then remove it from the "remove list" and check its features and ids, else add it to the "add list"; notice that removing a mote from the "remove list" means that the mote will not be removed from the local list;

   4 since the "motels" script makes no difference between the switched-off motes and those not connected, every mote that is still in "remove list" has to be checked up: if its "usb power" feature has the value ”off” then it means that it had been switched off by a client before the monitoring, thus it has to be removed from the ”remove list”;

5 update the local list;

6 update the global list and get the XML data;

7 forward the XML data to the clients of the updating services.

6. usbp: it executes a bash script in order to switch on or off a port of an USB hub; the port number is retrieved from the mote-port dictionary.

7. do usb power on/off: they are used as operative methods by the threads created in the ”perform operation” method. They invoke ”usbp” to send the signal to the USB hubs and then they check if the system has registered the operation. Since the ”motels” script does not recognize the connected motes which have been switched off, the ”do usb power off” method modifies the information on the involved mote and returns, besides the result, the (mote, feature, new_value) tuple. The latter is used to update the global mote and to report the change to the clients of the updating services.

Provider.
This interface is used by the daemon to create the connections between the NCGs. The server, after the parsing of the configuration file, instantiates a ”Provider” interface for each child in the network hierarchy. This means that every child (also called provider from now on) is seen as a distinct interface which contributes to create the global list of motes. The daemon, in order to keep the list updated, makes a request on the ”Streaming Subscribe” resource of every provider. Every time it receives a chunk of data, this is parsed using a method of ”XMLCore”'s and the new information are exploited to update both the local and the global list. The requests on the local motes are forwarded to the provider, which in turn could forward them to its children. This mechanism continues until the requests do not reach the interfaces which interact directly with the physical motes. When this happens, the response follows the path in backward until it reaches the the initial client. Another interesting aspect is that for each step in the network hierarchy the levels related to the interfaces are incremented by one. This implies that, for what concerns the execution of the operations on the motes, the higher is

the distance of the provider in the hierarchy, the lower is its usage priority. The principal attributes of the ”Provider” class are the following.

1. polling interval: the internal mechanism uses a particular instruction, called ”select”, to monitor the presence of updating data. This instruction permits defining a timeout, after whose expiry it stops and returns. The ”polling interval” attribute is used to set up the timeout value, so that to exit from the select and control if the server has set the stop event.

2. provider id/ip: these attributes represent the identifier and the IP address of the NCG. Their values are specified in the configuration file.

3. connection and response: these attributes are used to store the ”HTTP-Connection” and the ”HTTPResponse” objects which permit managing the exchange of data between the daemon and the provider.

The main methods implemented by the interface are the following.

1. discovery: first of all is created the connection to the provider by requesting the ”streaming subscribe” resource. When the connection is established, the first received chunk contains the complete list of the motes of the provider. This becomes the local list of the interface. Every mote is then passed to the list handler and added to the global list. After having notified the server that the discovery phase is completed, the ”handle streaming connection” method is activated.

2. handle streaming connection: the main task of this method is to receive the chunks of data coming from the provider. This is done by using the ”select” instruction, which accepts in input a list of file objects (in this case the list is composed only by the connection socket) and as soon as one or more of them contain new data it returns the list of the ready-to-read ones. Once a chunk is available, this is read and passed to the ”list update” method as input. The ”select” instruction is able to recognize if the socket is closed by the provider, i.e. if the connection is dropped. This can happen if the provider crashes or its daemon is stopped. If the connection can not be restored, the interface empties the local list, reports the situation to the list handler and to

the clients and starts to request every few seconds a new streaming subscription to the provider. When the new connection is established, the interface refills the local list and returns performing its normal tasks. Another important issue which deserves to be deepened regards the reading of the chunks. Python 3 provides the ”http.client” library to manage the HTTP connections and responses, but there are no ways to read a HTTP Streaming response a chunk at a time with the offered methods. It has been therefore necessary to develop an ad-hoc method with which implement this functionality and patch the Python library. This method, called ”read_chunk”, can be perfectly integrated into the library, thus is not needed making any other modification to the original Python code.

3. list update: when a chunk of data is received from the provider it means that the list of the motes is changed. The ”list update” method firstly invokes the ”XMLCore” module to parse the chunk, then it creates a set of lists for the motes that will be added, removed and updated. Once the lists are filled with the information from the chunk and the local list is updated, the thread communicates the modifications to the list handler. Instead of creating a new chunk with the XML data obtained from the list handler's methods, the thread modifies the received one increasing by one the level associated to the interfaces, and then it forwards it to the clients of the updating services.

4. forward operation: this method permits forwarding an operation to a provider. Firstly it creates the XML payload by using the list of the ids of the involved motes, secondly it makes a request on the appropriate resource of the provider and it returns the ”HTTPConnection” object.

5. manage response: the connection created in the previous method is used to get the ”HTTPResponse” object which can be passed in input to the ”select” instruction. When the response is available it is parsed using the ”XMLCore” module. The retrieved data are used to fill the dictionary containing the results of the operation, where the keys are the motes and the values are (code, message) pairs.

6. do operation: it calls the ”forward operation” and the ”manage response” methods and then it returns the dictionary of the results.

7. do usb power on/off: these methods are used as operative methods by the threads created in the ”perform operation” method. They simply call the ”do operation” method and return the dictionary of the results.

8. set mote position: in order to store the position information of a mote at every level of the network hierarchy, it is necessary to forward the information to the provider directly connected with the mote. The ”Provider” interface overrides the ”set mote position” method of the ”Interface” class precisely to achieve this purpose.

User interface.
The architecture of the daemon permits integrating interfaces of various nature. ”MoteLs” and ”Provider”, although their mechanisms are very different, make use of an internal thread with which they control the status of the motes. The ”User Interface”, unlike them, leave the management of the motes to the users, which can change the local list as they prefer. This means that the users can define, by using the methods offered by the interface, the ids and the features of the motes. The interface is useful to simulate a WSN with many motes or when the physical motes are not available. Furthermore, when a fake mote is defined by a user it can be merged to a real mote with the ”merge motes” resource. This can be seen as the addition of information on a mote. Another way to get this result is to define a fake mote that shares an id with the real one. The list handler, when the interface adds the fake mote to the global list, unifies the information on the two motes. When the motes are no longer necessary the users can simply remove them from the list.

3.4 WebIoT integration

The users of the testbed can access to the various functionalities by using the WebIoT web application. In its booting phase, the application instantiates a set of ”MoteManager” objects. The references to the classes are stored in a database. Every manager creates a list of detected motes which will compose a part of the global one managed by the upper layer. The thesis project has implicated the creation of a ”MoteManager” extension, with which make possible interacting with the Python daemon. Figure 3.7 depicts the UML diagram of the classes of the developed solution.

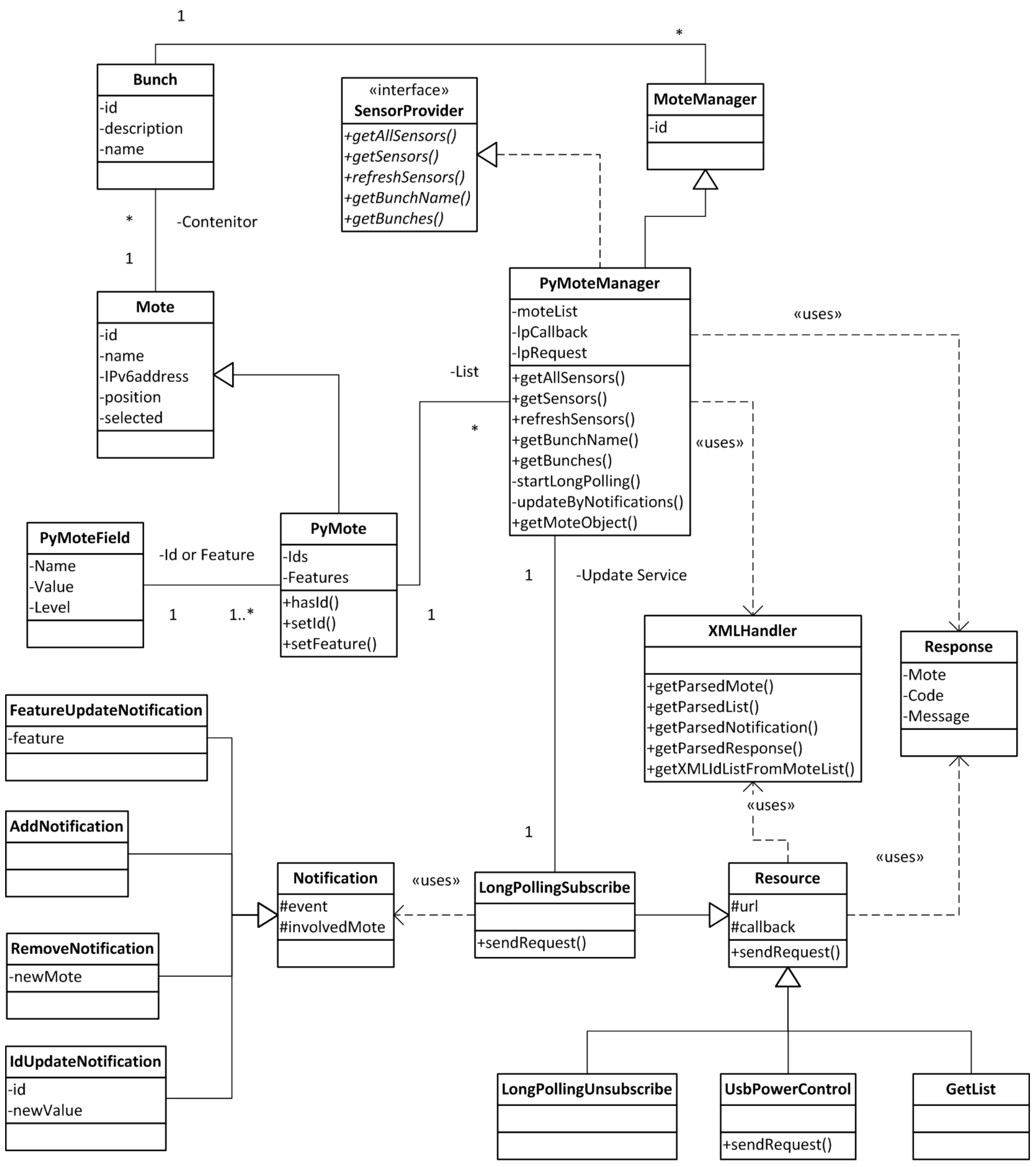


Fig. 3.7: Web Application - UML Diagram of the classes

The ”PyMote” class (”Py” stands for Python) extends the more general ”Mote” class, which abstracts the principal properties of the sensor nodes. Every mote is tied to a bunch, which is composed by a set of motes related each other. The relation can be, for example, their geographical position. The ids and the features of each mote are instances of the ”Field” class and are stored in two distinct lists. Every field has a name, a value and a level. The level equals to the one associated to the highest priority interface of the daemon which supports the field. The ”PyMoteManager” class extends ”MoteManager”, from which it gets the id attribute, and implements the ”SensorProvider” interface. The latter defines the base methods for the visualization and the update of the information about the motes. The principal attributes of ”PyMoteManager” include a list of ”Mote” objects, which is kept updated thanks to a subscription to the updating service provided by the daemon. The ”lpCallback” attribute is used to store the AsyncCallback object with which receive the updating data. The ”AsyncCallback” class represents one between the fundamental mechanism on which is based the GWT technology. When an asynchronous callback is created, in order to make an RPC (Remote Procedure Call) request to a server, the application will not expect to get the response but it will proceed in its execution. This means that it is possible to make non-blocking calls on the resources offered by the daemon. When the response becomes available the application executes one among the ”onSuccess” and ”onFailure” methods of the AsyncCallback object. The described mechanism is perfect to exploit the HTTP Long Polling technique, which is intrinsically asynchronous. The ”lpRequest” attribute has precisely this purpose, since it is used to store an instance of the ”LongPollingSubscribe” class. The mote manager, by requiring the ”long polling subscribe” resource to the NCG at the top of the network hierarchy, initially gets the complete list of the motes and then it starts to receive the updating data. This is done by calling the ”startLongPolling” method. Each message exchanged between the web application and the daemon is written in XML. The XML handler creates a bridge between the information contained in the messages and the classes of the application. Every time a chunk of data arrives from the daemon, the mote manager exploits the XML handler to parse it and obtain a list of Notification objects. The list is then used to update the available information on the motes. The application can require the resources offered by the daemon, like ”long polling subscribe” and

”usb power on/off”, by instantiating the corresponding class and using the ”sendRequest” method. The obtained response is received and read thanks to the callback and then transformed by the XML handler into a list of Response object, each of which stores the (code, message) pair for a mote.
At the moment the WebIoT application is still under development. One future improvement is surely represented by the possibility of exploiting all the functionalities provided by the daemon.

Chapter 4

Tests and Results

Abstract:
In this chapter are discussed the various tests which have been made to adjust the parameters of the daemon and to analyse the performance of the system.

4.1 Environment set up

The daemon has been installed on the WISE-WAI Server and on seven NCGs. In this way has been possible to manage up to 60 motes scattered throughout the department. The Python 3.1.3 interpreter has been compiled on a NCG and then copied on any other involved. The network hierarchy has been defined by setting up the configuration files of the daemons. The logical network topology which has been chosen is described in figure 4.1.
The figure 4.2 depicts a configuration more suitable for networks that are distributed in a wider geographical area, where it makes sense to create a sub-network for each building or delimited zone. In this case, the NCGs at the top of the sub-networks would be directly connected to server, while the others would be not aware of it. The choice to adopt the flat configuration described in figure 4.1 for the WISE-WAI testbed comes also from a consideration about the network latency. In this scenario, the daemon running on the WISE-WAI Server represents the point where all the information from the NCGs converge. Since the number of motes and ALIXes situated in the same room is very limited, the use of sub-networks would imply the creation of useless additional levels, which would introduce the exchange of further

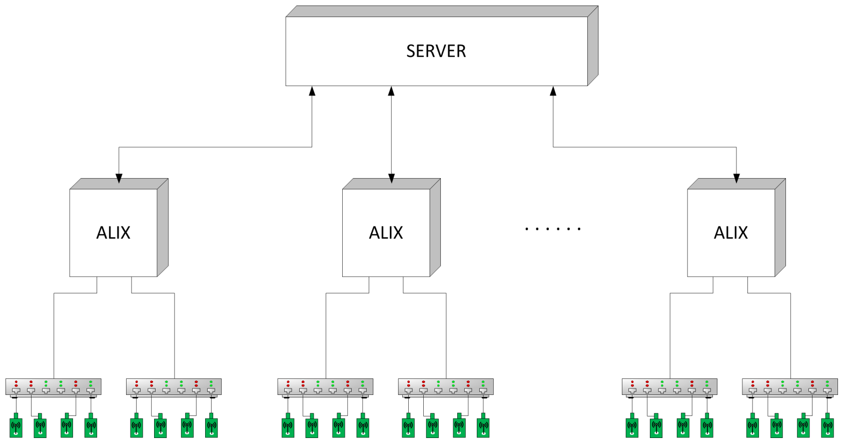


Fig. 4.1: Flat configuration of the logical network topology.

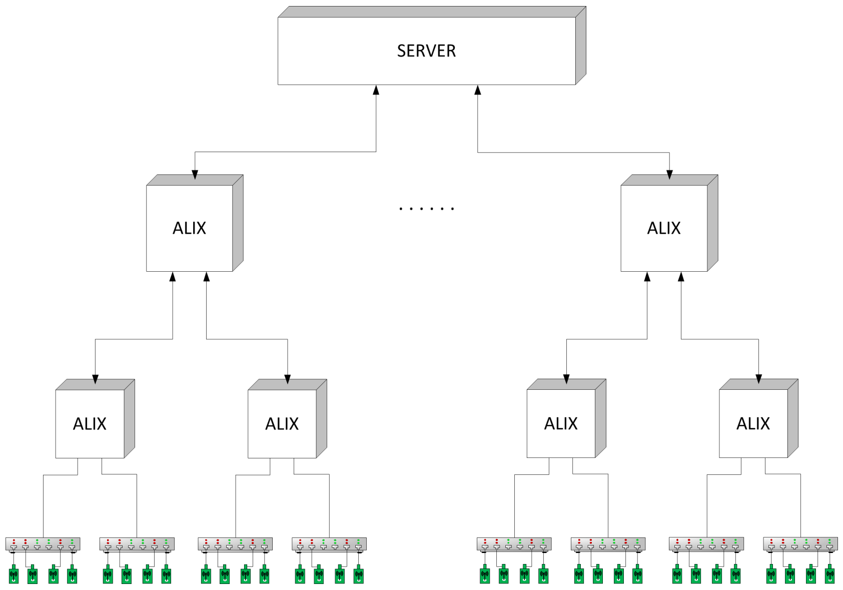


Fig. 4.2: Tree configuration of the logical network topology.

messages. Every eventual NCG used as root of a sub-tree would manage a list of motes whose size would not justify a similar configuration. The latter is in any case possible and properly working. In order to execute the tests has been created a set of Python scripts. Each script has been used to interact directly with the daemons. Their functioning will be described in the following sections.

4.2 Parallelism degree

The number of threads running at any given instant has to be limited to a known value. Incrementing without control the number of concurrent threads can lead to a degradation of the performance, since the management of the time slots and the context switch, although the threads all refer to the same process, is not at zero cost. Moreover, the use of semaphores and condition variables can introduce further overhead. When no request is served by the demon, the active threads are the following:

- the thread in charge of accepting the client requests;
- the internal threads of the interfaces;
- the internal threads of the updating service connections.

Every time a client request is received, the number of threads increases by one; if the request is on an active resource the corresponding operation can activate up to a new thread for each involved mote. Before to panic, there are two important facts to consider. The first is that the threads for most of the time are in a waiting state, in which they do not use the computational resources. The second one is more subtle. As discussed in Section 3.3.3, every interface defines a weight for each supported operation. The weight corresponds to the credit needed to perform the operation on a single mote. If the weight of an operation is set to zero by an interface, the operation will be executed on all the motes by the same thread. The "Provider" interface, since it is used to forward the requests between the NCGs, defines all the weights to zero. This means that number of threads can be increased for each request at most by the number of the directly connected motes. Since this number can however be high, the weights set by the interfaces have to be chosen accurately. The credit management is done within every interface by using a dedicated attribute. The initial amounting of credit is set to one. The weights can be chosen in the real interval [0,1]. Each operative thread has to get a quantity of credit equal to the operation weight before starting its execution. This implies that for each interface the number of threads that can process in parallel is limited to a precise value. When two or more operations have to be performed at the same time on the motes managed by a single interface, the available credit will be shared between them. In this way, the control on the parallelism degree can be extended simultaneously to all the active operations. The test described in the current section had the aim to find the best values for the weights of the operations, so that to achieve the best possible performance. The test has been performed on a single ALIX connected to 40 motes by using various USB hubs arranged in cascade. The requests on the resources of the daemon have been made by using a Python script. The script gets the list of the concerned motes from an XML file, which is forwarded to the daemon as the payload of the request. The script calculates the time between sending the request and receiving the response. The retrieval of the response is done by using the "select" instruction. In figure 4.3 is shown the graph of the execution times obtained by turning on and off all the 40 available motes and varying the weights of the associated operations (their values can be derived from the reciprocal of the parallelism degree). The time values refer to the execution

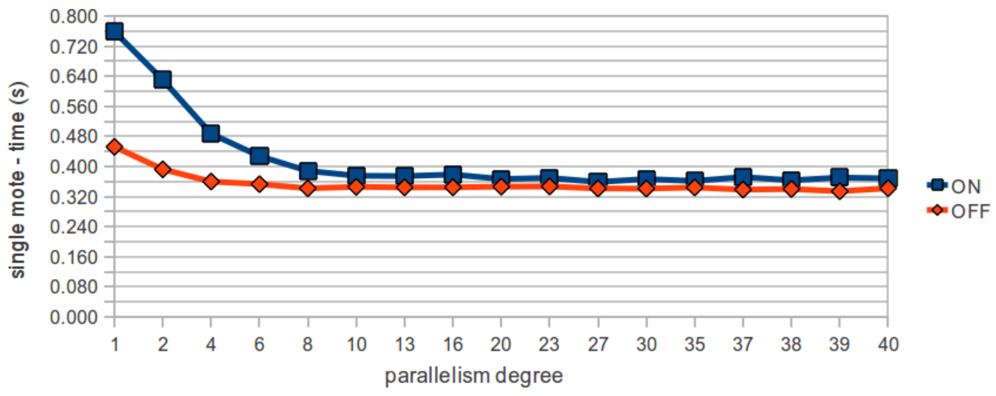


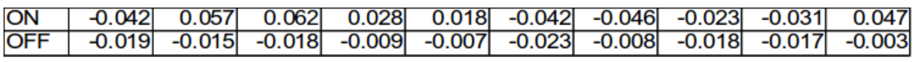
Fig. 4.3: Execution times obtained by varying the weights of the operations.

of the operations on a single mote. When the parallelism degree is equal to one the operative threads are completely serialized, while when the degree amounts to forty they are all activated at the same time. It is possible to see that the performance become stable once the parallelism degree reaches

the value of ten. Even if the number of parallel threads increases, the time required to switch on or off a mote does not decrease. The causes of these results can be found in the internal mechanisms of the hubs and of the Operating System. When a hub is powered on, its ports are activated one at a time. This means that the threads which try to turn on the motes are actually serialized. The behaviour of the hubs, however, is not enough to explain the situation. It can be supposed that a second serialization is introduced by the USB controller, which is not able to serve all the simultaneous requests and thus creates a bottleneck.
In order to optimize both the number of parallel threads and the performance, the weights of the operations of turning on and off the motes have been set to 0.1, which corresponds to a maximum number of parallel threads equal to ten.

4.3 Update delay

The test has been performed in order to observe, with respect to a request on an active resource, the time between the receipt of the response and the arrival of the update data. The predisposed Python script makes use of two distinct threads, one for sending the request and retrieving its response and the other for receiving and parsing the updating data. The latter thread requires to the daemon the ”Long Polling Subcribe” resource and maintains a counter for storing the number of satisfied motes. Every time a block of data arrives the thread stores the point in time, parses the data and increments the counter. When the counter value reaches the number of the involved motes the thread ends the cycle. The test has been performed by using a single ALIX with eight connected motes. In figure 4.4 are reported the results of ten measurements. The negative values refer to the situation



ON	-0.042	0.057	0.062	0.028	0.018	-0.042	-0.046	-0.023	-0.031	0.047
OFF	-0.019	-0.015	-0.018	-0.009	-0.007	-0.023	-0.008	-0.018	-0.017	-0.003

Fig. 4.4: Measured delays between the receiving of the response and the arrival of the updating data.

in which the updating data arrive before the response. It is possible to notice that for what concerns the turning off of the motes the values are always negative, while for the turning on the values are both positive and negative.

This is due to the behaviour of the ”MoteLs” interface. The turning on of the motes leaves to the monitoring thread the management of the updating data. During the operation execution the thread recognizes the turned-on motes and updates the local and the global lists. This brings to get a separation between the response's sending and the forwarding of the update data. The turning off, instead, implies the creation of a unique block of update data, which is sent to the clients by the resource thread. In this case, indeed, the monitoring thread is not able to distinguish the disconnected motes from the turned-off ones, thus the updating of the mote information has to be done by the resource thread before the sending of the response. The higher measured delay is equal to 62 ms. This value, compared to the average time required by the turning on of eight motes on a single ALIX (approximately 3.7 seconds), is lower than the 1.7 percent. The test's result shows that the system is able to update and reconfigure itself in a very short time. Furthermore, the clients of the updating services can receive the messages almost in real-time.

4.4 Performance evaluation

The following tests have been performed in order to analyse the performance of the system and its level of scalability. Every test has involved the operations of turning on and off the motes by requiring the resources offered by the daemons. The script used to execute the test is the same of the one employed during the parallelism analysis. In figures 4.5 and 4.6 are given the trends of the execution times in two different configurations. In the first case the daemon was running on a single ALIX, while in the second it was running on six different ALIXes. The ALIX engaged in the first test has been connected to 16 motes. It is possible to notice that in the left part of the graph the decreasing of the execution times is very marked. This behaviour indicates that when the number of the involved motes is low the parallelism has a great impact. The serialization introduced by the hubs and the USB controller starts to be significant when the number of involved motes is equal or greater than six. A similar trend is given also by the second test, where the motes are distributed among more ALIXes. Figures 4.7 and 4.8 show a comparison between the two configurations. The reported values refer to the total time of execution of the operations. The graphs

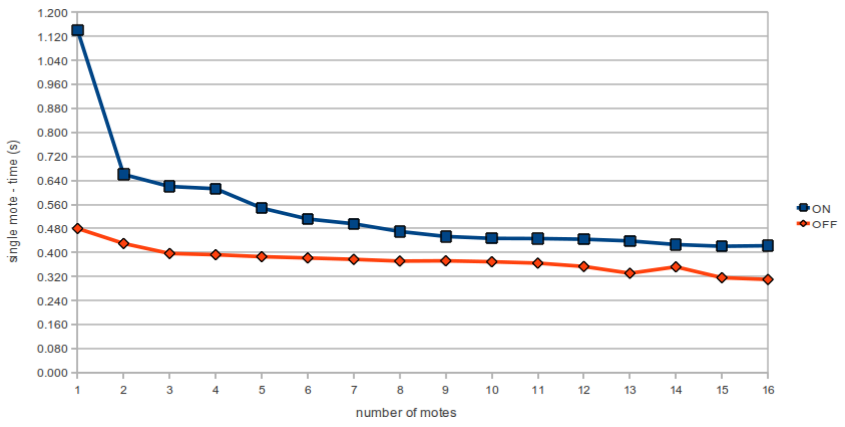


Fig. 4.5: Execution times obtained by varying the number of motes on a single ALIXes

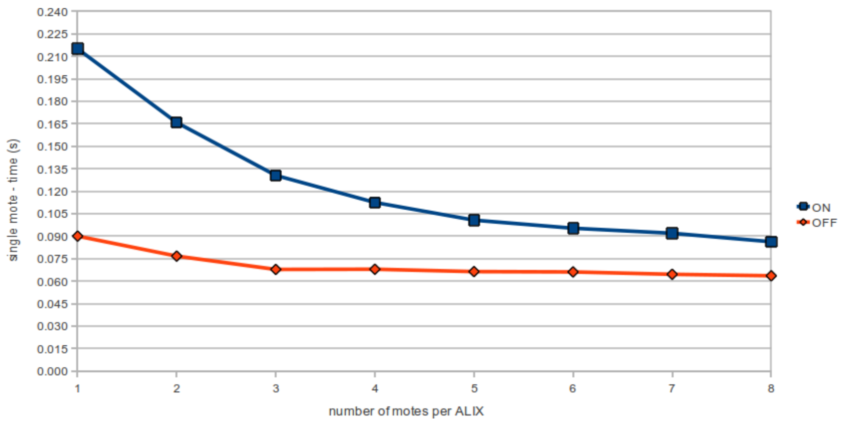


Fig. 4.6: Execution times obtained by varying the number of motes on six ALIXes

show that the two trends are very similar. For what concerns the turning off, the system is almost insensitive to the number of ALIXes employed. The trend is identical also for the turning on, except for an additive factor related to a different management of the updating data. These results are confirmed by the graph in figure 4.9. This test had the aim to analyse the scalability of the system by studying its behaviour when increasing the number of involved ALIXes. Each ALIX has been connected to eight motes. At every step of the test the turning on and off have been executed on all the motes connected to the interested ALIXes. The time values reported in the graph are calculated as the ratio between the total time spent and the fixed number of motes per ALIX (equal to eight). It is possible to observe

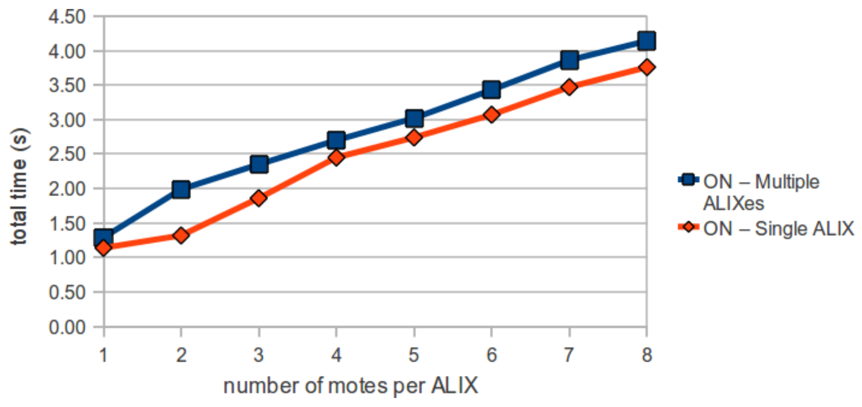


Fig. 4.7: Graph comparing the performance of turning ON the motes in the two configurations.

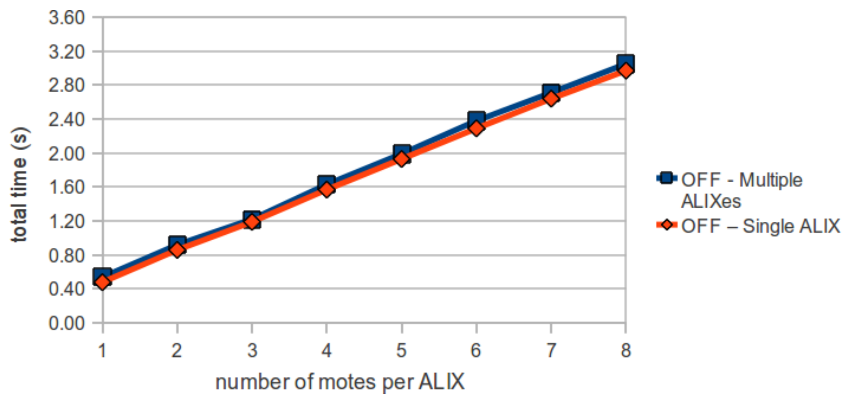


Fig. 4.8: Graph comparing the performance of turning OFF the motes in the two configurations.

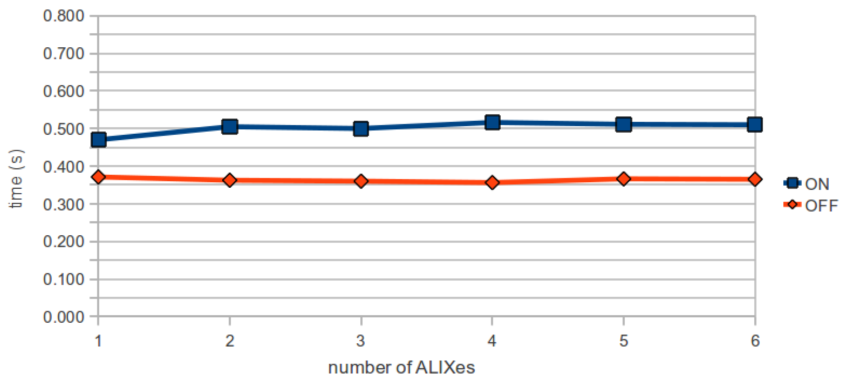


Fig. 4.9: Execution times obtained by varying the number of ALIXes

that the number of the engaged ALIXes does not affect on the performance. This demonstrates that the parallelism is exploited to its full capacity and the management of the results is fast and reliable. The tests have confirmed that the system architecture shows a high level of scalability, both for what concerns the internal functioning of the daemon and for the management of the network.

Chapter 5

Conclusion

The Internet of Things is a technological revolution that represents the future of computing and communications, and its development needs the support from some innovative technologies. The principal aim of the IoT is to connect physical objects to the web. New smart things should be able to process data, to configure, maintain and repair themselves, to make independent decision, to interact with others by exchanging information. The Wireless Sensor Networks constitute a direct application of the IoT paradigm. Every sensor node is provided with an IPv6 address, so it is reachable through web connections. The nodes can interact each other so that to share information and execute distributed applications. The nodes can be also connected to elaborative units called sink nodes. These tiny computers are used to provide the sensor nodes with power supply, to monitor their state and to accept external requests with which intervene on the WSN devices. In order to develop and test applications and protocols for the WSNs, the SigNET group of the University of Padova has set up a large testbed. The thesis project has consisted on the design and the development of a software system with which make possible the high-level management of this testbed and, more in general, of a WSN. The heart of the project is focused on the sink nodes. The developed software, which implements the functionality of Web Server, runs on every sink node as an internal daemon. The software is based on the REST architectural style, so it has been designed to offer web resources to the clients. The daemons are able to keep updated a local list of connected motes, which permits storing all the useful information about the motes and their states. The daemons are able to share this list with the other units, to accept requests on the available resources, to forward the requests through the network and, last but not least, to provide clients an updating service. The software architecture is based on the concept of modularity. The internal data structures are completely configurable according to the kind of sensor nodes that compose the WSN. The various kinds of interactions with the nodes are provided by a set of interface modules. Each module monitors the state of a set of nodes and implements the operations to intervene on their states. These operations are performed as result of client requests on the available REST resources. The interface modules and the resources can be added and removed as needed. This kind of architecture allows to easily add new functionalities to the system without change the core structure. Furthermore, the implementation and the integration of network-specific modules makes the system portable and easily extensible. The software is completely configurable also for what concerns the logical network topology. As described in Chapter 4, the sink nodes can be arranged in a flat configuration or in multiple layers. This permits to adapt the software according to the size and the features of the WSN. The use of XML for the data interchanging represents another choice that aims to make the software usable and accessible. The dynamic nature of the software ensures a fast and continuous updating of the information about the sensor nodes. The monitoring task, which is performed by the interfaces modules, exploits an implementation of the Bidirectional HTTP techniques in order to report the changes to the clients of the updating services by reducing as more as possible the network traffic. The developed software also shows a high level of scalability. The tests have demonstrated that the performance of the system remain constant with the increasing of its size. The fully exploitation of the parallelism, granted locally by using multithreading and globally by redistributing the client requests between the sink nodes, makes the system suitable both to small and large networks. As described in Chapter 3, the project has also involved the development of a part of the WebIoT application. The users in this way are able to interact with the WSN without being aware of the implementation details. The possible future improvements of the developed software concern the extension of the functionalities offered by the daemon, by creating new interface modules and resources. Even the WebIoT application can be integrated with new modules. In this way the users will be able to interact with the system in all its possibilities.

Bibliography

[1] C. Buratti et al., *Sensor Networks with IEEE 802.15.4 Systems*, Signal and Communication Technology, DOI: 10.1007/978-3-642-17490-2_1, Springer-Verlag Berlin Heidelberg, 2011.

[2] Vlado Handziski et al., *TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Networks*, in Proc. Of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, (RealMAN 2006), Florence, Italy, May 2006.

[3] WISE-WAI project web site.
Available online: http://cariparo.dei.unipd.it.

[4] Lu Tan et al., *Future Internet: The Internet of Things*, 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE).

[5] K. Finkenzeller, *RFID Handbook, 3rd Ed.*, Wiley, 2010, ISBN: 978-0-470-69506-7.

[6] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi, *From todayâĂŹs INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view*, IEEE Wireless Communications, vol. 17, no. 6, pp. 44-51, Dec. 2010.

[7] Angelo P. Castellani et al., *Architecture and Protocols for the Internet of Things: A Case Study*, In Proc. of the 1st IEEE IntâĂŹl. Wksp. Web of Things (WoT 2010 at IEEE PER- COM), pp. 678-83.

[8] Roy T. Fielding and Richard N. Taylor, *Principled design of the modern web architecture*, ACM Trans. Inter. Tech., 2(2):115-150, 2002.

[9] Michael Jakl, *REST Representational State Transfer*, CiteSeerX, http://citeseerx.ist.psu.edu, doi=10.1.1.97.7334

[10] T. Berners-lee, R. Fielding, and L. Masinter., *Uniform resource identifiers (URI): generic syntax*, Technical Report Internet RFC 2396, IETF, 1998.

[11] W3C, XML base, Technical report, W3C, 06 2001.

[12] P. Casari et al., *The "Wireless Sensor Networks for City-Wide Ambient Intelligence (WISE-WAI)" Project*, www.mdpi.com/journal/sensors, doi:10.3390/s90604056.

[13] http://www.alix-board.de.

[14] http://linux.voyage.hk/

[15] CrossBow Technology. TelosB sensor node. http://www.xbow.com.

[16] Google Web Toolkit , http://code.google.com/webtoolkit/.

[17] J. Hill et al., *System architecture directions for networked sensors*, In Proc. of the 9th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 93-104. ACM Press, 2000.

[18] G. Werner-Allen et al., *MoteLab: a wireless sensor network testbed*, in ISPN, 2005.

[19] S. Loreto et al., *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*, RFC 6202, April 2011.

[20] http://www.python.org

[21] B. W. Kernighan, D. M. Ritchie., *The C Programming Language (2nd ed.)*, Prentice Hall, ISBN 0-13-110362-8, 1988.

Ringrazio la mia famiglia, in particolare i miei genitori e i miei nonni, per avermi donato libertà e giusti consigli.
Ringrazio Veronica, per aver reso questo mondo un posto bello dove vivere.
Ringrazio tutti gli amici che ho la fortuna di avere accanto, in questi anni di gioe, fatiche, speranze e battaglie non mi sono mai sentito solo.
Ringrazio i ragazzi del laboratorio SigNET per il supporto durante questi mesi di tesi, in particolare un sentito grazie va ad Angelo e a Moreno che non si sono mai stufati di rispondere alle mie domande (o almeno credo).

Infine ringrazio Dio, semplicemente perché sono qui, vivo.