
A Lightweight Resource-Oriented Application Framework for Wireless Sensor Networks

Master's Thesis

Author : Andreas Kamilaris
camel9@gmail.com

Supervisor : Vlad Trifa
Professor : Friedemann Mattern

Institute for Pervasive Computing
Department of Computer Science
ETH Zürich

April 2009

Abstract

With current advancements in technology, an increasing number of embedded devices are being deployed around the world to solve dedicated, specific tasks like environmental monitoring or automating control. These resource-constrained devices operate in specialized, low-level physical links.

We investigate the use of smart gateways to provide access on the Internet for these devices. These gateways are responsible for discovering heterogeneous sensor devices in their neighborhood, extracting automatically as much information as possible concerning their characteristics and functionality and making them available as Web resources that can be accessed in an uniform way, independently of their low-level specific characteristics or the different communication protocols they use.

We also examine ways of accelerating performance and augmenting functionality of these resource-constrained devices in order to solve traditional problems that always exist in the unpredictable environment of sensor networks and concern energy constraints, reliability issues, transmission conflicts and device failures.

Our gateway links physical devices with existing Web technology, bringing sensor technology one step further, towards the way of full integration with the Web, in order to enable the vision of the *Web of Things*.

Table of contents

1	Introduction	1
1.1	Requirements	2
1.2	Thesis Overview	2
1.3	Contribution	3
2	Related Work	5
2.1	Existing Solutions	5
2.2	Solution Comparison	6
2.3	Reasoning about gateways	7
3	Architectural Styles for distributed Applications	9
3.1	Service Oriented Architectures	9
3.1.1	Big Web Services	9
3.2	REST and Resource Oriented Architectures	10
3.3	Discussion	12
4	Towards the Web of Things	15
4.1	Sensor Technology	15
4.1.1	Tmote Sky Sensors	15
4.2	Sensor Operating Systems	16
4.2.1	TinyOS	16
4.2.2	Contiki Operating System	17
4.3	Web Technology	18
4.3.1	HTTP	18
4.3.2	Multipurpose Internet Mail Extensions	19
5	REST-oriented Patterns	21
5.1	Device Discovery	21
5.2	Device Description	23
5.3	Service Description	23
5.4	Device Operation	24
5.4.1	Request/Response Model	25
5.4.2	Event-Based Model	25
5.4.3	Data Streaming Model	26

6	Gateways for Augmenting RESTful Devices	27
6.1	Gateway Architecture	27
6.2	Mechanisms Involved	30
6.2.1	Retransmissions	30
6.2.2	Caching	30
6.2.3	Congestion Avoidance	30
6.2.4	Failure Recognition	31
6.2.5	Failure Masking	32
6.2.6	Statistics	32
6.2.7	Adaptive Streaming	32
6.3	Synchronous/Asynchronous Translation	33
7	REST-enabled Sensor Devices	35
7.1	Sensor Interaction	35
7.1.1	Message Templates	36
7.1.2	Operational Phases	36
7.2	Sensor Deployment	39
7.3	Error Handling	39
7.3.1	Same Identity	40
7.3.2	Sensor Interference	40
7.3.3	Transmission Conflicts	40
7.3.4	Device Registration Error	40
7.3.5	Request/Response Message Issues	40
7.3.6	Gateway Selection	41
8	Evaluation	43
8.1	Experimental Setup	43
8.2	Experiments	43
8.2.1	Device Discovery Phase	43
8.2.2	Gateway's Performance	46
8.2.3	Gateway Vs Proxy	47
8.2.4	tinyOS Vs Contiki sensors	49
8.3	General Comments	50
8.3.1	Bottlenecks	51
9	Discussion and Future Work	53
9.1	Future Work	53
9.2	Conclusion	55
9.3	Acknowledgments	55
	Appendices	56
A	Software Installation on sensor Devices	57
A.1	TinyOS Installation	57
A.2	Contiki Installation	58
B	Source Code	59
B.1	Asynchronous/Synchronous Execution Synchronization	59
	Bibliography	62

Introduction

Nowadays, we witness a technological advancement in computing hardware that enables tiny devices perform quite complex computing tasks. Extremely small sensor devices provide advanced sensing and networking capabilities.

In parallel, a lot of research has directed low-level software that enables these resource-constrained devices to form sensor networks with different topologies, measure different quantities like temperature or humidity, transmit data to sinks and perform specific tasks like monitoring some area or solving a particular problem like traffic control, mainly after a static deployment in the targeted environment.

At the same time, many operating systems targeting these types of devices have been developed, which claim of increasing their performance. Through dedicated libraries and functions a programmer, with little effort, can exploit the sensing functionalities of these sensor nodes.

We believe that sensor networks should start being observed from a global view, where each individual device becomes a citizen of its own infrastructure, with dedicated requirements and specific tasks. In a real-life scenario today, many heterogeneous devices operating with different low-level protocols and layers appear in common places, each one having different functionality, trying to solve particular problems.

Hence, problem solving in embedded distributed systems has moved at a higher level, at the level of discovering unknown, heterogeneous devices, understanding their capabilities in a uniform way and interacting efficiently with them, possibly to solve a higher goal that involves combined use of these devices.

In other words, we define as necessity the design of a mediator service such as a gateway, which would provide an application framework that could handle and represent effectively embedded devices, interact with them in a standardized way in order to discover them, learn their abilities and exploit the services they offer. This framework, encapsulated in a more powerful device inside the already-existing network infrastructure (such as a router machine or a normal PC), could constitute a bridge, which would provide access to the Web for these embedded devices.

Concerning uniformity and standardization in the way of Web-enabling these sensor devices, we are inspired from the World Wide Web, which after all, operates in a high scalable and efficient way. The questions we must answer are mainly why is the Web so prevalent and ubiquitous and what makes it scale so well. Today's Internet was designed with simplicity, through which all the protocols and technologies that govern the Web were implemented. We intend to follow this simplicity by integrating in our model an architecture style, which is highly parallelized by Internet's simple and efficient operation, namely REpresentational State Transfer (REST).

Furthermore, as a consequence of the use of this application framework, additional needs would be created, that are unable to be solved by today's solutions. We refer to those requirements in detail in Section 1.1.

Our long-term intention, is to use this referenced gateway system, as the intermediate vehicle that will drive us to the *Web of Things*, where every smart object is accessible from the Web.

1.1 Requirements

By developing a gateway that represents embedded devices, we must also consider augmenting functionality of them. Dedicated mechanisms, integrated on the gateway software should be developed that could accelerate performance.

A number of problems that appear in constrained environments where heterogeneous devices exist must be faced, such as device discovery, uniform data description and modeled interaction. An architectural style must be followed that provides standardization and interoperability during interaction with these resource-constrained devices.

Through our gateway's infrastructure, we aim in providing a general application framework that would constitute a new direction for future generations of WSN applications. The general requirements for such a framework are:

- data must be easily exported into Web applications in simple, easy to be understood data structures
- full access to devices, that would become a cloud where devices can be individually invoked and addressed, where each of their sensing operations can be also individually accessed in a standardized way
- concurrent Web client support
- particularities of WSN should get abstracted just like TCP/IP, for example masking failures, dealing with not always connected devices, operating smoothly when multiple unknown devices are around and providing some form of Quality of Service

We plan in developing an open and scalable sensor network infrastructure that supports many types of interaction and is capable of accomplishing the most important, in general, tasks that a WSN must provide. There are three large classes of WSN deployments that need to be supported by our framework. Random access interaction (request-response model), continuous monitoring (devices stream data at regular intervals) and event-based systems (events are sent sporadically).

1.2 Thesis Overview

After introducing the reader to the problem we try to solve, the goals we want to achieve and the general picture, we try to identify in Section 2, relevant projects that exist in scientific research and we consider whether we offer something innovative.

We present in Section 3 the architectural styles that dominate the modern Web-based distributed systems and we make a selection according to our requirements. In Section 4 we present the technologies we will use, in our way of implementing the concept of the *Web of Things*. Following the architectural style we decided, namely REST, we define in Section 5, the different low-level interaction patterns to access embedded devices that support REST.

After these theoretical issues, we begin explaining the practical stuff. In Section 6, an overall view of the gateway as we designed and implemented is available and in Section 7 the RESTful software for resource-constrained devices we developed is presented.

Then we continue with an evaluation of our model in Section 8, where we describe a number of experiments we prepared in order to test the performance of our system. Last, in Section 9 some conclusions are extracted generally about the paper and some future work that would enhance the functionality of our gateway is listed.

1.3 Contribution

Our gateway model, offers an efficient approach for solving the problem of Web-enabling heterogeneous, embedded devices. Interaction patterns, based on REST architectural principles, provide uniformity and interoperability in a high-level, proposing solutions to challenges such as device discovery and service description. Through its unique design, our gateway is transformed into an application framework where powerful, flexible applications can be developed with little effort and where multi-user concurrency is provided. The mechanisms we integrated in gateway's structure, mask away failure issues and transmission conflicts, providing robustness, reliability and advanced performance. Hence, an application programmer does not have to worry about these issues. These "smart" gateways, by design, are capable of creating in the future a backbone structure for the *Web of Things*.

Related Work

Our work becomes attractive, only in case we possess something innovative to offer to scientific community. We present in this chapter a listing of the existing related projects, which are based on the foundational concepts on which we are motivated to create our own system. At the end we make a short comparison between these implementations and our own idea, to consider whether we actually propose something new in research in the field of distributed systems or we offer just another common project, created simply for the needs of gaining a master degree.

2.1 Existing Solutions

The idea of linking physical objects with the Web is certainly not new, and early approaches used physical tokens (such as barcodes or RFID tags) to retrieve information about objects they were attached to [29, 21].

In the Cooltown project [20], each thing, place, and person had an associated Web page with information about them. XML and SOAP was used for the communication and WSDL and UDDI for the discovery and device advertisements.

Shaman was an early gateway system that enabled low-power devices to be part of wider networks [30]. An extendable, Java-based service gateway for networked sensor systems was developed, to integrate small network-attached sensor-actuator modules (SAMs) into high-level networking communities. It was a pioneer attempt towards creating a uniform, simple interface for interaction with heterogeneous, sensing devices.

Many other technologies for building distributed applications on top of heterogeneous devices have been proposed (CORBA, JINI, or RMI). JXTA [32] is a set of open protocols for allowing devices to collaborate in a peer-to-peer fashion. JXTA was among the first real attempts to bridge physical objects in the world with the Internet. More recently, Web services, have also been used to interconnect devices on top of standard Web protocols [26]. Most of these approaches are based on tightly coupled solutions, where each element had full knowledge about the other peers and the functions they offered. Unfortunately, these approaches are not sufficient to deal with the constraints and requirements of mobile embedded devices, in particular for ad-hoc interaction with new devices that have unknown properties.

Several systems for integration of sensor systems with the Internet have been proposed [12, 3, 18]. SenseWeb project [19] is a platform for people to share their sensory readings using Web services to transmit data on a central server. Pachube [17] offers a similar community Web site for people to share their sensor readings and uses more open data formats. Unfortunately, these approaches are based on a centralized repository and devices need to be registered before they can publish data, thus are not sufficiently scalable. Prehofer [25, 33] recently proposed a

web-based middleware that can be related to our approach, however Internet is used only as a transport protocol and references to use a fully Web-like approach are not mentioned. An interesting approach to use the web architecture and the semantic web technologies can be found in [34]. The approach found in [31] is the first to our knowledge to take a very similar approach to ours, but focuses mainly on the discovery of devices, unfortunately a systematic approach and system evaluation is lacking.

An interesting, large project being currently developed in Nokia laboratories and can be parallelized with our work is Nokia Home Control Center¹. It targets building a technology-neutral smart home that can be controlled with a mobile phone, using a unified user interface. It focuses in supporting the most common smart home technologies and it allows third parties to develop their own solutions and services on top of the platform, expanding the system to support new services and smart home technologies. The project is still under refinement and we wait to see the final product.

Arch Rock Primer Pack/IP² is a commercial solution for a standards-based, out-of-the-box wireless sensor network (WSN) application development and deployment platform, based on service-oriented architecture (SOA), Internet Protocol (IP)-based networking, and secure, reliable low-power mesh networking. It is considered by the writer a tightly coupled system with a comparatively high price.

Reuse and composition of documents and streams on the Web have recently gained broad attention. pREST [6] describes a protocol how resources can be interlinked with REST principles in mind. Everything in pREST is modeled as a resource. Producers and consumers that are pREST-enabled can be linked together by so called *endpoints* to exchange sensor-data (eg. camera sending an image to the Web-server that upon receipt publishes the image).

With advances in computing technology, tiny Web servers could be embedded in most devices[4]. The idea of each device having its own Web page is appealing because device pages can be indexed, searched, and accessed by search engines, and this directly from a Web browser.

The real challenge, nowadays, is to force sensing devices, to be accessible directly on the Web, by understanding the HTTP protocol. It is likely that in the near future, most WSN platforms will have native support for TCP/IP connectivity, thus embedded HTTP servers will be commonly available on most devices. In this case, any device and their properties will be URI-addressable and would understand any Web request. This approach is highly desirable because there is no need to translate HTTP requests from Web clients into the proprietary APIs for the different devices. Recently, a small footprint Web server, was implemented on Contiki [10], that is capable of processing HTTP requests and serving a Web page with real-time sensor data. A Web server was also recently integrated on Sun SPOT devices [16], in a way that services offered by the device can be accessed through a RESTful interface using any Web browser.

2.2 Solution Comparison

Most of these existing, Web-based approaches use HTTP only to transport data between devices, whereas HTTP is in fact an application protocol. Projects that specifically focus on re-using the founding principles of the Web as an application protocol are still lacking. Creation of devices that are Web-enabled *by design* would facilitate the integration of physical devices with other content on the Web. As pointed out in [37], in that case there would be no need for any additional API or descriptions of resources/functions.

¹<http://http://smarthomepartnering.com/cms/>

²<http://www.archrock.com/product/>

The majority of the systems we examined employ a rather static infrastructure, making themselves inappropriate for dynamic, distributed, mobile scenarios. To face device heterogeneity, many approaches introduced heavy protocols and platforms, which should not be the case in resource-constrained environments. In almost all cases tight-coupled, closed systems were developed, where concepts like interoperability and uniformity were totally absent. In other implementations, we observed approaches, based on a centralized repository, where devices needed to be registered before they could publish data, thus they were not sufficiently scalable.

TinyREST [22], is the only promising idea with its big picture fitting our view so well. It also proposes a gateway that directly embeds devices as resources into the Web. The RESTful interface allows clients to send `POST` and `GET` requests directly to the devices via URI (eg `http://gwIP/sensor1/light`). Our approach goes one step further from this work, transforming TinyREST's gateway into a powerful system with integrated mechanisms, that augments functionality and increases performance of resource-constrained devices.

Our system differs from all these implementations as it tries to present itself as a general framework, where flexibility in dealing with sensor networks is provided. We believe to be the only project that manages to model all the aspects that concern challenges in embedded, heterogeneous, distributed systems, such as device discovery, device description and interaction between a mediator (in our case the gateway) and a sensor network, in a uniform way, highly motivated from simple, concrete standards that govern the Web today, through REST architectural style. Furthermore, the multi-user support, provided by our system as well as its performance, by means of the mechanisms implemented inside the gateway (Section 6.2), highlight our approach, as a unique research direction.

2.3 Reasoning about gateways

At last, the reader could argue, why a use of a gateway since HTTP-enabled software on sensor devices is currently being developed. After all, the reader could think, gateways have been a research subject for the last decade and it is not something new.

We declare that, still, serving full HTML pages directly from devices does not make much sense, unless users want to access a single device in particular. Even in that case, there is a serious degradation in performance. Furthermore, when the overall goal is to interact with a set of devices and use the collected or aggregated data from the set as a whole, then devices can serve much simpler, machine-readable formats, such as JSON or XML. In that case the data collected from a multitude of devices can be aggregated at intermediary nodes in the network such as gateways. Gateways can offer a unique possibility of processing the data from disparate devices locally and offer a high-level, interactive user interface (UI) that allows end users to use the WSN from the gateway without actually interacting with the devices directly.

In addition, our system claims to provide full compatibility with directly HTTP-enabled devices. Nevertheless, with current developments concerning enabling HTTP on devices, dedicated daemons, working as proxies are necessary to operate on mediator devices. So since these mediator devices have generally larger computing capabilities, why not using a gateway instead of a proxy, which would accelerate significantly performance, providing reliability and QoS? In Section 8.2.3, a comparison is performed between these two possibilities.

To sum up, we argue that there exist distributed problems that can not be solved without the use of an intermediate, intelligent device. This mainly concerns dynamic indexing of mobile devices, which is not possible without a mediator when new devices (dis)appear continuously. Any attempt to seamlessly integrate devices into the Web includes challenges such as device

discovery, device and service description as well as multi-user concurrent support. These challenges, considering also the capabilities of the sensor devices that exist in the market today, require the employment of a "smart" gateway.

Architectural Styles for distributed Applications

Since we require from our system to face challenges such as standardization and interoperability in the heterogeneous nature of today's sensor network deployments, the need for choosing an architectural style, basen on which we would develop our patterns, is necessitated. In this chapter, we examine the core principles that apply in modern Web-based distributed systems and we study their applicability in the field of sensor networks. Basically, we identified two architectural styles, that dominate in interactions where heterogeneous systems are involved through the Web, namely *Service Oriented Architectures* through *Big Web Services* and *Resource Oriented Architectures* through *REST*. In the following sections, we examine both of these approaches and we make a selection of the most appropriate based on our special needs.

3.1 Service Oriented Architectures

Service Oriented Architecture (SOA), can be technically defined as an application architecture within which all functions are defined as independent services with well-defined invocable interfaces, which can be called in defined sequences to form business processes. These services are independent of each other, heterogeneous and distributed. The interaction is message-based rather than through direct calls (CORBA¹, RPC², RMI³). Large enterprise applications are easily built, by relying on the infrastructure to take care of the integration issues (protocols to use, intermediate processing, routing and distribution, data transformation). SOA is particularly suited for Business-to-Business integration scenarios.

3.1.1 Big Web Services

The vehicle for the realization of SOA listens to the name *Big Web Services* (they are also called *WS-**). Their difference with conventional middleware is mainly related to the standardization efforts at the W3C that try to guarantee platform independence (hardware, operating system), reuse of existing networking infrastructure (mainly through the ubiquitous use of HTTP), programming language neutrality (.NET can talk with Java and vice versa) and portability across middleware tools of different vendors.

Big Web Services are loosely coupled components that foster software reuse, trying to be composable in order to that be adopted incrementally.

Their design has a remarkable client/server flavor and it is based on three elements:

¹<http://en.wikipedia.org/wiki/Corba>

²http://en.wikipedia.org/wiki/Remote_procedure_call

³http://en.wikipedia.org/wiki/Java_remote_method_invocation

1. Service requester. The potential user of a service (the client)
2. Service provider. The entity that implements the service and offers to carry it out on behalf of the requester (the server)
3. Service registry. A place where available services are listed and that allows providers to advertise their services and requesters to lookup and query for services

Mainly, their realization is based on two key concepts:

- architecture of existing synchronous middleware platforms
- current specifications of SOAP, WSDL and UDDI⁴.

SOAP. Simple Object Access Protocol [15] deals with making the service invocation part of the language in a more or less transparent manner. Basically, it is an XML language defining the structure and the format of SOAP messages (*envelopes*). A SOAP message consists of a header (containing information for some QoS) and a body (containing the payload). SOAP messages are the basic building blocks of the WS-*. They are conceived as the minimal possible infrastructure necessary to perform RPC through the Internet. To translate a RPC into a SOAP document, the call is first serialized (marshalled) into XML (eg. using JAXB⁵). Before storing the XML into the SOAP body, WS-* allows a series of transformations to be applied to the payload, ranging from encryption to reliability contracts.

WSDL. Web Services Description Language [5] tackles the problem of exchanging data between machines that might use different representations for different data types. This involves data type formats (e.g., byte orders in different architectures) and data structures (need to be flattened and then reconstructed). Basically, it is a statically defined, based on XML, standardized interface description language which is machine readable, can be automatically derived from existing APIs and is binding independent (it allows run time choice).

UDDI. Universal Description and Discovery protocol is responsible of helping a client find the service he actually wants, among a potentially large collection of services and servers. The goal is that the client does not necessarily need to know where the server resides or even which server provides the service. It can be seen as a platform-independent, XML-based registry for businesses worldwide to list themselves on the Internet. It presents itself as the infrastructure for Web services, meaning the same role as a name and directory service (i.e., binder in RPC) but applied to Web services. However, nowadays it is mostly used in constrained environments (internally within a company or among a predefined set of business partners).

3.2 REST and Resource Oriented Architectures

The notion of REpresentational State-Transfer (REST) has been conceptualized in Roy Fielding's PhD thesis [11]. Rather than being a technology or standard, REST is an architectural style which basically explains how to use HTTP as an application protocol. Indeed, unlike most integration technologies using the Web (e.g. WS-* Web Services), REST advocates in providing services directly based on the HTTP protocol itself. An API (Application Programming Interface) fulfilling the REST architectural style, is said to be RESTful.

⁴http://http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration

⁵<http://jaxb.dev.java.net/>

Beyond the basic design criterias provided in Roy Fielding's thesis, the REST community has been working on refining the notions to create Resource Oriented Architectures (ROA) to provide service on the Web. Traditionally, Resource Oriented Architectures are used to connect together virtual services on the Web. Because of the underlying simplicity of this architecture, we believe that they could as well be adapted, in order to interconnect the physical world and in particular wireless sensor networks.

Basically a Resource Oriented Architecture is about four concepts [27]:

1. Resources. A resource in a ROA is anything important enough to be referenced and linked by other resources. In more service-oriented terms, a resource is an entity that provides (or could provide) services. In the case of WSNs, the network as a whole is a resource and each individual node can be considered as a resource. Furthermore, the sensors (e.g. temperature, humidity, etc.) are resources as well.
2. Their Names. A resource has to be addressable, i.e. it needs to have URI (Uniform Resource Identifier) that uniquely identifies it (e.g. <http://.../sensors/dataCenterSensor/temperature>). This concept is known as the requirement for addressability of RESTful APIs.
3. The links between them. Thanks to the hyperlinking structure of the Web, resources can be related and connected to one another. As an example the resource [dataCenterSensor](#) links to its underlying resource [temperature](#). This is the requirement for connectedness of RESTful APIs.
4. And their Representations. Resources can be represented using various formats. The most common one for resources on the Web is (X)HTML. Its intrinsic support for browsing an human readability makes it a nice candidate, however REST does not limit representations to this one only. In particular when machine readability is required, XML and JSON are often chosen. Note that JSON is as a data interchange format which stands as a lightweight alternative to the sometimes too verbose XML ⁶.

REST advocates the use of a uniform interface. Unlike WS-* Web Services where an arbitrary number of operations can be performed on objects, resources can only be manipulated by the methods specified in the HTTP standard (after version 0.9). Amongst these verbs, four are most commonly used:

- GET is used to retrieve a representation of a resource. Concretely, sending a GET request along with the URI <http://.../nodes/sunspot1/sensors/light.json> would return the light level currently observed by [sunspot1](#) in the JSON format. GET is both an idempotent and safe operation. This means that no matter how many times you apply the operation, the result is always the same and that such a request does not change the state of the resource it is called on.
- PUT represents an insert or update. As an example, it could be used to change the state of an actuator. Sending a PUT request to <http://.../nodes/sunspot1/actuators/led/1> with the JSON payload `on`, turns led 1 on. PUT is also idempotent because sending the same PUT message more than once has no affect on the underlying resource since its state will remain the same as after the first request.
- DELETE is used to remove resources. It is idempotent as well.

⁶<http://www.json.org>

- POST is the only non-idempotent and unsafe operation of HTTP. It is a method where the constraints are relaxed, to give some flexibility to the user. In a RESTful system, POST usually models a factory service. Where with PUT you know exactly which object you are creating, with POST you are relying on a factory service to create the object for you.

Furthermore, REST implies a stateless communication between resources, where the interacting resources do not maintain a local state (eg. cookie) during a communication session. Thanks to their simplicity, the use of an uniform interface and the wide availability of HTTP libraries and clients, RESTful services are truly loosely-coupled [24]. This concretely means that services based on RESTful APIs can be re-used and re-combined in a quite straightforward manner.

The REST principles led to the idea of *Resource Oriented Architectures* [28]. ROA carefully applies the REST principles to the whole design process and the development of the application. It further expects from the system to include two additional concepts. *Addressability*, which means that every resource should be accessible through exactly one globally unique identifier, that can be stored, transmitted, bookmarked and indexed by search engines and *connectedness*, which requires resources to be linked to as many other resources possible, leading to a well-connected and easy browseable mesh of hypertext documents suitable for human interaction.

3.3 Discussion

In this chapter we illustrated the two main architectural styles that dominate in web-based integration and interaction, in distributed systems. In this last section, we discuss our choice for the architectural style used in our system.

Big Web Services follow a complex set of related standards based on XML. They are ideal for enterprise computing scenarios (business to business) with involvement of computationally powerful machines. Structured interface contracts between consumer and producer through WSDL, help in building reliable software, promoting integration and reuse.

However, they do not maintain a uniform interface, but service description based on WSDL has a rather static structure. Whenever a service provider changes its interface, all clients have to recompile their stubs as well. This drawback makes WSDL hard to use in mobile environments, where you have to integrate physical devices where their interface is not known in advance or where the interface needs to evolve over time. In addition, UDDI can not operate in a highly unpredictable environment such as a sensor network, since it needs stable references for the records it maintains. At last, use of XML-based SOAP messaging, creates big computational requirements, mainly in complex XML transformations, that mobile and sensor devices are not able to accept due to their limited resources in battery, memory and processing power.

On the other hand, RESTful Web Services do not require any interface declarations. Their efforts to reuse the principles that designed the World Wide Web and be fully operational and compatible with Web's standards, provides ease of use, flexibility and simplicity at RESTful interactions. Concepts like resource identification through URIs, uniform interfaces for all resources and self-descriptive messages provide a framework where heterogeneous, resource-constrained devices can operate together. Computation is shifted away from a centralized infrastructure into a distributed, loosely coupled network of resources.

Furthermore, the flexibility of using multiple resource representation formats helps the designer of the system choose between a variety of possibilities. For example, smart phones can still use the XML standard for interoperability while sensor devices can include JSON for their

interactions. At last, in the area of sensor networks in general, operations that need to be accomplished are relatively simple, so they can successfully and with little effort be designed, with a RESTful style.

REST's biggest weakness remains the fact that it has not effectively answered yet, the problem of interface description. This causes many problems during application integration efforts. Web Application Description Language (WADL) [23], makes positive steps towards the direction of solving that problem, but there are still a lot to be made, concerning mainly a standardized structure and interface format.

REST architectural style is our selection, based on which we decided to build our system. After all, REST has slowly-slowly started to be leveraged today, by all major Web 2.0 applications. We are convinced that the scalability REST offers and the perceived ease of adoption, due to its light infrastructure requirements (need of just a Web browser) will be proved useful tools that will accompany our efforts.

Towards the Web of Things

In this chapter, we will present the technologies we will use, in our way of implementing the concept of *Web of Things*. We will describe the sensor devices we will employ through our work and the sensor operating systems, on which we decided to develop parts of our software. Furthermore, since we decided to follow REST architectural style, we must comply with the technologies that rule the Web today. For that reason, we will also illustrate *HTTP* protocol as well as *Multipurpose Internet Mail Extensions* (MIME) types, which we will integrate in our work.

4.1 Sensor Technology

Here we will give some information concerning the sensor device types we decided to use during our work. We were basically interested in selecting a device type which would operate wirelessly, in a low-level physical link and which would be capable of performing some useful functionality, for example measuring some quantity like temperature or humidity, or monitoring some territory for noise or movement. Our desire was, in addition, to choose a device type which operates in a resource-constrained environment, since one of our overall goals is to augment the functionalities of these devices and increase their capabilities through the use of a gateway. Based on these thoughts, we selected *Tmote Sky*¹ sensor devices in order to design, in cooperation with them, our system and execute on them our experiments.

4.1.1 Tmote Sky Sensors

Tmote Sky, is claimed to be the next-generation mote platform for extremely low power, high data-rate, sensor network applications designed with the dual goal of fault tolerance and development ease. It boasts a large on-chip RAM size (10kB), an IEEE802.15.4 radio and an integrated on-board antenna providing range up to 125 meter.

It also offers a number of integrated peripherals, including a 12-bit ADC and DAC, Timer, I2C, SPI and UART bus protocols, as well as a performance boosting DMA controller. Tmote Sky offers a robust solution with hardware protected external flash (1Mb in size) and applications may be wirelessly programmed to the Tmote Sky module. In the event of a malfunctioning program, the module loads a protected image from flash. Toward development ease, Tmote Sky provides an easy-to-use USB protocol for programming, debugging and data collection.

¹<http://http://www.sentilla.com/moteiv-transition.html>

Furthermore, it provides integrated humidity, temperature and light radiation sensors and it claims of offering seamless vertical integration between the hardware and the TinyOS operating system. The reader can graphically see such a sensor device in the left of Figure 4.1. In the right of the same Figure, the internal design of Tmote Sky is graphically presented.

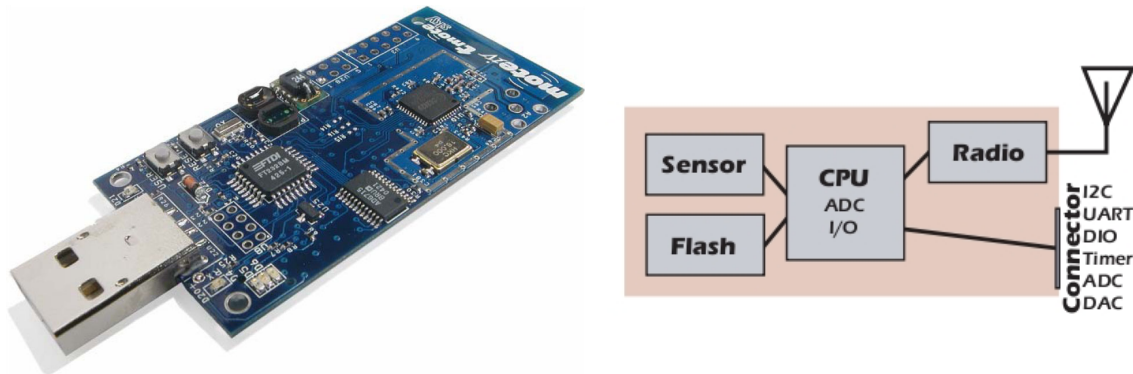


Figure 4.1: A Tmote Sky physical sensor device and its internal design.

4.2 Sensor Operating Systems

After we concluded in which sensor device platform we would use, it is time to select the operating system, based on which we would develop our RESTful software. The only requirement we posed, was that the desired operating system had to be specifically designed for embedded devices. Our choice was rather easy, since there exist not so many operating systems for sensor devices available for free use in the market. TinyOS [2] was our first priority, since it has been developed for a couple of years and it claims a stable performance with many available tools available that help in implementation. Furthermore, Tmote Sky's manufacturers promote the use of their devices in combination with TinyOS.

As a second step we decided to introduce a second operating system in our platform, to test our system in a real heterogeneous environment. We chose Contiki [8] for that reason, since it is the product of significant research, produced by Adam Dunkels's² group. Furthermore, Contiki gains more and more acceptance in sensor networks area. There is already a large forum in the Web³, where developers exchange their opinions and experiences, based on their activity on this operating system.

In the following subsections, we present these two operating systems in detail.

4.2.1 TinyOS

TinyOS is a free and open source, component-based operating system and platform, targeting wireless sensor networks. It is an embedded operating system, written in the nesC programming language as a set of cooperating tasks and processes. It basically started as a collaboration between the University of California, Berkeley in co-operation with Intel Research, and has since grown to a be an international consortium, the TinyOS Alliance.

TinyOS applications are written in nesC, which is actually a dialect of the C programming language, optimized for the memory limitations of sensor networks. TinyOS programs are built

²<http://http://www.sics.se/~adam/>

³<http://https://lists.sourceforge.net/lists/listinfo/contiki-developers>

out of software components, which are connected to each other using interfaces. Interfaces contain definitions of commands and events. Components must implement the events they use and the commands they provide. TinyOS provides interfaces and components for common hardware abstractions such as packet communication, routing, sensing, actuation and storage. A general idea of the programming model can be seen in Figure 4.2.

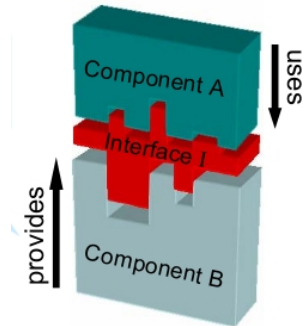


Figure 4.2: General programming Model used in TinyOS.

A TinyOS component can post a task, which the OS will schedule to run later. Tasks are non-preemptive and run in FIFO order. Operation is completely non-blocking, there exists only a single stack. Therefore, all I/O operations that last longer than a few hundred microseconds are asynchronous and have a callback. TinyOS uses nesC's features to link these callbacks, called events, statically.

In case a programmer writes a single application, TinyOS links statically program code with operating system's code and compiles it into a small binary, which is then uploaded to the real sensor device.

4.2.2 Contiki Operating System

Contiki is an open source, highly portable, multi-tasking operating system for memory-efficient networked embedded systems and wireless sensor networks. It is mainly designed for microcontrollers with small amounts of memory. A typical Contiki configuration is 2 kilobytes of RAM and 40 kilobytes of ROM.

Contiki's key advantage that lead to its popularity in the area of sensor networks, is that it provides IP communication, both for IPv4 and IPv6 [10]. Actually, it provides the flexibility for the programmer to choose between full IP networking and low-power radio communication mechanisms. For communication within a wireless sensor network, Contiki uses the RIME [7] low-power radio networking stack. The RIME stack implements sensor network protocols ranging from reliable data collection and best-effort network flooding to multi-hop bulk data transfer and data dissemination. IP packets are tunnelled over multi-hop routing via the RIME stack.

Contiki is written in the C programming language and consists of an event-driven kernel, on top of which application programs can be dynamically loaded and unloaded at run time. Contiki processes use lightweight protothreads [9] that provide a linear, thread-like programming style on top of the event-driven kernel. In addition to protothreads, Contiki also supports per-process optional multithreading and interprocess communication using message passing. Contiki provides three types of memory management: regular malloc(), memory block allocation, and a managed memory allocator.

Interaction with a network of Contiki sensors can be achieved with a Web browser, a text-based shell interface or dedicated software that stores and displays collected sensor data. The text-based shell interface is inspired by the Unix command shell but provides special commands for sensor network interaction and sensing.

To provide a long sensor network lifetime, it is crucial to control and reduce the power consumption of each sensor node. Contiki provides a software-based power profiling mechanism that keeps track of the energy expenditure of each sensor node. Being software-based, the mechanism allows power profiling at the network scale without any additional hardware. Contiki's power profiling mechanism is used both as a research tool for experimental evaluation of sensor network protocols and as a way to estimate the lifetime of a network of sensors.

4.3 Web Technology

Here we present the main web technologies that appear in our work, namely HTTP and MIME types.

4.3.1 HTTP

HyperText Transfer Protocol (HTTP) [14] is a stateless, application-level protocol that is used extensively in today's Internet. HTTP follows a request/response standard of a client and a server. A client is the end-user, the server is the web site. The client makes an HTTP request, mainly by using a web browser and the responding server stores or creates resources such as HTML files and images.

A request message consists of a request line, some headers and an optional message body. HTTP defines, in total, eight request methods indicating the desired action to be performed on the identified resource. These methods are:

- **HEAD.** Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.
- **GET.** Requests a representation of the specified resource.
- **POST.** Submits data to be processed (e.g., from an HTML form) to the identified resource. The data is included in the body of the request. This may result in the creation of a new resource or the updates of existing resources or both.
- **PUT.** Uploads a representation of the specified resource.
- **DELETE.** Deletes the specified resource.
- **TRACE.** Echoes back the received request, so that a client can see what intermediate servers are adding or changing in the request.
- **OPTIONS.** Returns the HTTP methods that the server supports for specified URL.
- **CONNECT.** Converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.

In HTTP/1.0 and since, the first line of the HTTP response is called the status line and includes a numeric status code (such as "404") and a textual reason phrase (such as "Not Found"). In HTTP/0.9 and 1.0, the connection is closed after a single request/response pair. In HTTP/1.1 a keep-alive-mechanism was introduced, where a connection could be reused for more than one request.

A key feature of the HTTP protocol is its capability to negotiate the format of the information exchanged between client and server using meta-data. The client indicates one or several desired format(s) as MIME types (Section 4.3.2) together with language preferences, and the server will try to answer the request by finding the resource that matches the request the best.

4.3.2 Multipurpose Internet Mail Extensions

Multipurpose Internet Mail Extensions (MIME) [13], is a standard that describes the structure of messages in the Internet. Originally, MIME was invented to embody any kind of data into emails (eg. images, video, audio, etc.), but today is applied to other areas as well (eg. HTTP, search meta-data, etc.).

MIME categorizes format and encoding into types called *Internet Media Type* or simply MIME type. The Internet Assigned Numbers Authority (IANA) maintains a list of all the official MIME types⁴ (applications are allowed to define their own types). Types contain subtypes to further organize and classify. A MIME type for an image encoded as *png* is expressed with *image/png*, an audio in *mp3* with *audio/mp3*. Within a MIME document, the MIME type is specified with the keyword *Content-type*.

Documents that use MIME have a *header* and a *body*. The body contains the payload (the encoded document(s)) and the header maintains instructions how the payload has to be interpreted.

⁴<http://www.iana.org/assignments/media-types/>

REST-oriented Patterns

In this chapter, we describe the different low-level interaction patterns to access devices that support REST. These interaction patterns are necessary for a gateway to be informed about device's existence, gain general knowledge about the device and collect as much information as possible concerning the services it offers in order to successfully represent the device and its resources on the Internet, in an easy to be understood, for the Internet users, way.

These interaction patterns, could be applied, for achieving interoperability and accelerating performance between devices and a gateway. The patterns and approaches we use, are a product of many experiments and they should be seen by the reader mainly as indications or suggestions concerning possible implementations on (new) low-level protocols or technologies that appear in the market, in order for them to be supported by a gateway, which also follows the RESTful architectural style. They were considered having REST architectural principles in mind and they are designed carefully in order to achieve optimized REST functionality on resource-constrained devices.

5.1 Device Discovery

As soon as a device is somewhere deployed and is equipped with an electrical source, begins to search for a gateway to declare its existence. The reason we require for the device to search for the gateway (and not vice-versa) is to avoid continuous broadcast messages from the gateway which are, in their large majority, useless and will increase effectively power consumption. By means of this approach only the necessary packets are exchanged between the two sides and we avoid unnecessary broadcast messages.

Figure 5.1 shows the interaction pattern which is followed in the *Discovery Phase*. The messages exchanged have the following semantics:

1. A device, in specified time intervals, broadcast a message, which indicates its existence to a gateway. In that message, information about reaching the device is included.
2. A gateway, after it receives the broadcast message, answers with a message indicating its characteristics, as well as necessary information in order to be contacted.
3. Device realizes the existence of the gateway in its vicinity and responds with a message indicating its general device characteristics.
4. The gateway, after receiving device description data, answers with an Acknowledgment message.

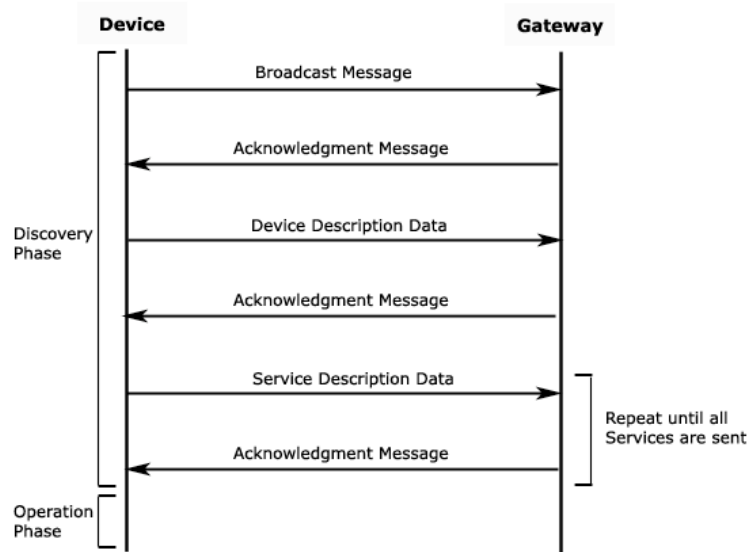


Figure 5.1: Communication Pattern followed in *Device Discovery Phase* between a gateway and a device.

5. The device begins the procedure of sending a number of messages, each of them containing a description of a particular service the device offers.
6. For each service description message received, gateway returns an Acknowledgment, indicating reception of the service data.
7. The previous two steps are repeated until device has no more service description data to send and all messages are acknowledged by the gateway.

It must be noted that, in case of a loss of an Acknowledgment message, somewhere during the *Device Discovery* phase, device re-sends the corresponding description message again, after a predefined time interval. In case of no acknowledgments, this procedure is repeated for a number of iterations and then device returns to its initial state, searching again for a gateway.

During the transmission of the service description messages, it is important that the gateway acknowledges every message, by indicating exactly which service it acknowledges in every response. This should be done in order to be avoided loss of service description messages due to delayed acknowledgments from the gateway, which targeted a previous service description message. More advanced, to save unnecessary retransmissions, gateway could straight respond with an indication of the latest service which it received. Since the service description transmission procedure in the devices is sequential, this causes no problems.

All the values of time intervals as well as the number of retries used in retransmission scenarios from the device, depend heavily from the type of the device used and the software installed on it. Therefore, they should be considered by the programmer of the device, after a logical number of experiments and tests. In case the device message or some of the service description messages are bigger than the default size supported by device's operating system, fragmentation can be used where the initial message is split in a number of smaller packets, which are sent sequentially.

After the gateway acknowledges the last service description message, device enters *Operation Phase*, where it permits the gateway manipulate itself by sending requests to it. From

now on, gateway is authenticated by the device. Gateway, on the other hand, after receiving all description messages from the device, creates all the resources necessary to support and represent it through its presentation abilities.

5.2 Device Description

Since we decided to follow the REST model for our interactions, some guidelines must be given, concerning the data which device will use to describe itself (device description data) as well as the services and the functionality it offers (service description data). Our main goal is to find a trade-off between sending as much information as possible (in order to help the users easily understand the capabilities of the device they are manipulating) and maintaining performance and low energy consumption at the same time. Of course, the amount of useful information varies from device to device. More information can be provided by a Smart Phone and much less by a sensor device. Also, energy capabilities of these two examples vary significantly.

It is important to note that we encourage the use of coding for describing the data in order to achieve lower energy consumption during devices' operation and consequently to extend their life-time. This would of course require the use of dedicated parsers inside the gateway system to decode properly the data. Anyway, this is a matter of the programmer of a specific device type, in case the energy constraints of it don't let it fully operate according to our guidelines.

As far as Device Description data is concerned, the most important information we require from a device is its unique device ID, its device name as well as its type. Additional information like location, specific model, manufacturer, date manufactured etc. are welcomed but not necessary for the device's operation. As an example, a very simple but complete message, sent from a TinyOS sensor device to describe itself can be:

```
"7 M4AOCFAO TmoteSky."
```

This sensor's unique ID used to program it is γ , its name is *M4AOCFAO* and its type is *TmoteSky*.

5.3 Service Description

Concerning Service Description data, many more constraints must be defined, in order to adapt and to enable REST protocols to be applied. A programmer of a device must be careful in following these disciplines to describe the services his device provides, in order to accelerate its performance. We will describe how this can be done in Section 6.2.

At first, each service is mainly seen as a resource. It has a Uniform Resource Identifier (URI) which must be chosen carefully to represent precisely the service. Resources can be named after the physical property for which they provide functionality, for example a resource measuring warmness of the environment should be named *Temperature*, a resource which is capable of turning on/off LEDs should be named *Light* etc.

Considering the fact that resources will be presented in the Internet and manipulated from Internet users, a universal way must be found for their return types to be defined. The answer to this question lies again in REST approach and is called MIME types. A Resource measuring a quantity in integer or double values can provide a *text/plain* MIME type and a resource capable of presenting images in a digital photo display may require *image/jpeg*.

It is still necessary for Internet clients, since they are forced to use REST to know with which verbs they can interact with the devices to use their services. We decided for that reason to introduce capabilities, which each service can offer. These capabilities are mainly the REST

verbs that are used in Internet: *GET*, *POST*, *PUT* and *DELETE*, plus some other special verbs which are encapsulated inside these four main verbs to provide additional functionality, necessary in the resource-constrained environment where we act. These four more verbs are *START*, to start the service listening to requests, *STOP*, to stop the service from operating, *RESTART*, to start the service from the beginning and at last *EVENT*, to indicate that a particular service is capable of producing some form of streaming (depending on the quantity it measures). In particular *EVENT* keyword is added at the end of the URI concerning device and service being invoked and it is encapsulated inside the verb *POST* with the universal parameters *interval*, which specifies the delay in seconds for every new streaming packet and *iterations*, which specifies the number of streaming packets sent. The approach we use of introducing some additional verbs to gain some advanced functionality can be defined as an *augmented REST design for resource-constrained devices*.

Furthermore, we require from each service description message to contain the parameter names as well as the corresponding parameter types that need to be defined, so someone can easily and successfully invoke the service.

Finally, a textual description part is expected, which will describe the resource's operation in a few words.

The reader can logically wonder how an Internet client, merely from this information can understand thoroughly the resource's semantics and use the resource properly. Basically, this is one of the reasons we introduce REST for our interactions, to make the procedure as universal as possible. In addition, since we mainly deal with devices that have only limited physical capabilities, we expect the services they offer to be basic, for example measuring some quantity or turning on/off some electrical source.

At last, we present two simple examples to show how this resource description can be done. At first a resource measuring temperature in Celsius degrees, having streaming capabilities, could be defined in a simple service description message as:

```
"text/plain Temperature 'measure in Celsius' 0 GET, EVENT".
```

The number 0 is an indication that no parameters are needed to invoke the service.

An other example can be a resource, capable of turning on some LEDs with colors green, red and blue. This resource can be described as:

```
"text/plain Light 'set RGB LEDs' 1 char color POST".
```

In that case we have 1 parameter, namely *color*, which takes a character as a value and through POSTing from Internet, a Web user can turn on a LEDs specified by the character used, 'G' for green, 'R' for red and 'B' for Blue.

5.4 Device Operation

As we described in Section 5.1, after a device sends all its device and service description data, can start its operational phase. In *Operation Phase*, the device becomes a server that listens for incoming requests from Web users, wishing to use the services offered by the device. In this mode, three types of operations are possible, and are explained in the next sections. We believe that by categorizing these operation types into these three categories, we claim to touch all the main interaction demands required from a sensor network. Through our categorization and the way we handle each case, we want to propose an international way of trading these operations, which way could provide interoperability in the future between many types of heterogeneous devices.

5.4.1 Request/Response Model

At the application level, the basic interaction pattern with a device is Request/Response, similarly to the client/server model of the Web. In this model, a client sends a request to a device and expects an answer from it. This can be either to retrieve the value of a sensor or of a variable (for example the reading of a light or temperature sensor), or to send a command to the devices (turn on LEDs or reset a counter).

We decided that every request posed to a device should have the following format:

```
Request {
    deviceID;
    serviceName;
    command;
    List (parameters,values);
}
```

The *deviceID* is the unique ID of the device, *serviceName* is the name of the resource being invoked, while *command* is the REST verb used. An optional list of parameters and their corresponding values is possible, to fully specify the possible parameters for the request.

The response created by the device has a similar format and correlates strongly with the request:

```
Response {
    deviceID;
    serviceName;
    command;
    List<Object> result;
}
```

The only difference here is the *result*, which is the return type of the service after it performed its intended operation. Since we know in advance the MIME type of the service's return type, we can easily cast the type to the correct type and get the result in the desired format.

Request and Response data structures can be easily parsed in a dedicated, easy to be understood form, highly dependable from the device type used, in order to create request and response messages that are exchanged between the gateway and the device.

5.4.2 Event-Based Model

In many applications, Request/Response model is not sufficient, for example when devices must monitor the environment for a particular phenomenon to occur. In the relatively not very frequent case when this phenomenon occurs, they must immediately inform the network or the sink about it. In this case, a simple rule on the device can be periodically evaluated and the device will send data only if the rule is fired. As HTTP was designed as a client-server protocol, where the client must pull the data it needs, a solution is needed for pushing data over HTTP. A solution we propose is that devices mark the message they send as an *event-occured* message and the gateway is responsible to handle it according to its type or significance, possibly forwarding the events only to Internet clients already interested in these type of events (through clients' subscription/notification model which is handled at the gateway system).

5.4.3 Data Streaming Model

Most sensor network applications, have as a purpose, data collection from all the nodes. At specific time intervals, all devices send their data to a sink. This is exactly the same process as for events described above, but the difference here is that events are sent periodically when a timer is fired instead of a rule checked.

Our purpose is to standardize this procedure, clearly by following REST principles. In particular, we demand `EVENT` keyword to be added at the end of the URI concerning device and service being invoked and to be encapsulated inside the verb `POST` with the universal parameters *interval*, which specifies the delay in seconds for every new streaming packet and *iterations*, which specifies the number of streaming packets sent. Basically it is a modification of the Request/Response model, which causes the device to start a streaming for the time delay and iterations number specified in the parameters of the request.

Gateways for Augmenting RESTful Devices

In this section, we describe how we designed and implemented the gateway. It was designed so that it complies with all the requirements we posed in Chapter 1. It was developed, not only with the goal of supporting devices, communicating and interacting with them using the communication patterns we presented in Section 5, but also with the goal of augmenting their performance and their functionality, by using some techniques we will shortly present. The implementation was mainly done in Java programming language, because of the versatility and portability of the language which allows the gateway to run on virtually any device that has a Java virtual machine.

6.1 Gateway Architecture

In Figure 6.1, we illustrate the architecture which was followed to implement our gateway. The gateway follows a modular architecture and is composed of three principal layers. In general, *Device Layer* is responsible for interaction and management of devices, *Control Layer* is the central processing unit of the system and *Presentation Layer* has the duty of presenting the available devices and their corresponding services to the Internet users, through a *Web Server* interface.

Device Layer is the layer mainly responsible for the communication with the devices. *Device/Service Discovery/Communication Module* has the duty of interacting with the devices and supporting the communication patterns, we pre-described. It is also responsible for sending and receiving messages to/from the device during device's *Operation Phase*. It deals mainly with low-level communication, with devices having specific types only if these device types are supported by the module through dedicated drivers which lie inside it. We defined these driver interfaces to be generally simple to be implemented, as we would like to welcome programmers write drivers for new device types that appear in the market. Currently, our gateway system has support for two sensor operating systems, Contiki and TinyOS. We can graphically see how *Device/Service Discovery/Communication Module* is developed in Figure 6.2. As soon as that module discovers a new device, a new thread is automatically created, dedicated in the task of representing the newly discovered device. Using threads for devices simplifies their management, accelerates their performance and keeps the implementation logic as simple as possible. From now on, device operates in its own thread environment and behaves independently of its underlined hardware specifications and specific physical characteristics. All device threads have the same behavior and functionality, although they can in reality, represent totally different technological appliances.

Inside every device's thread, there are stored information concerning device description as

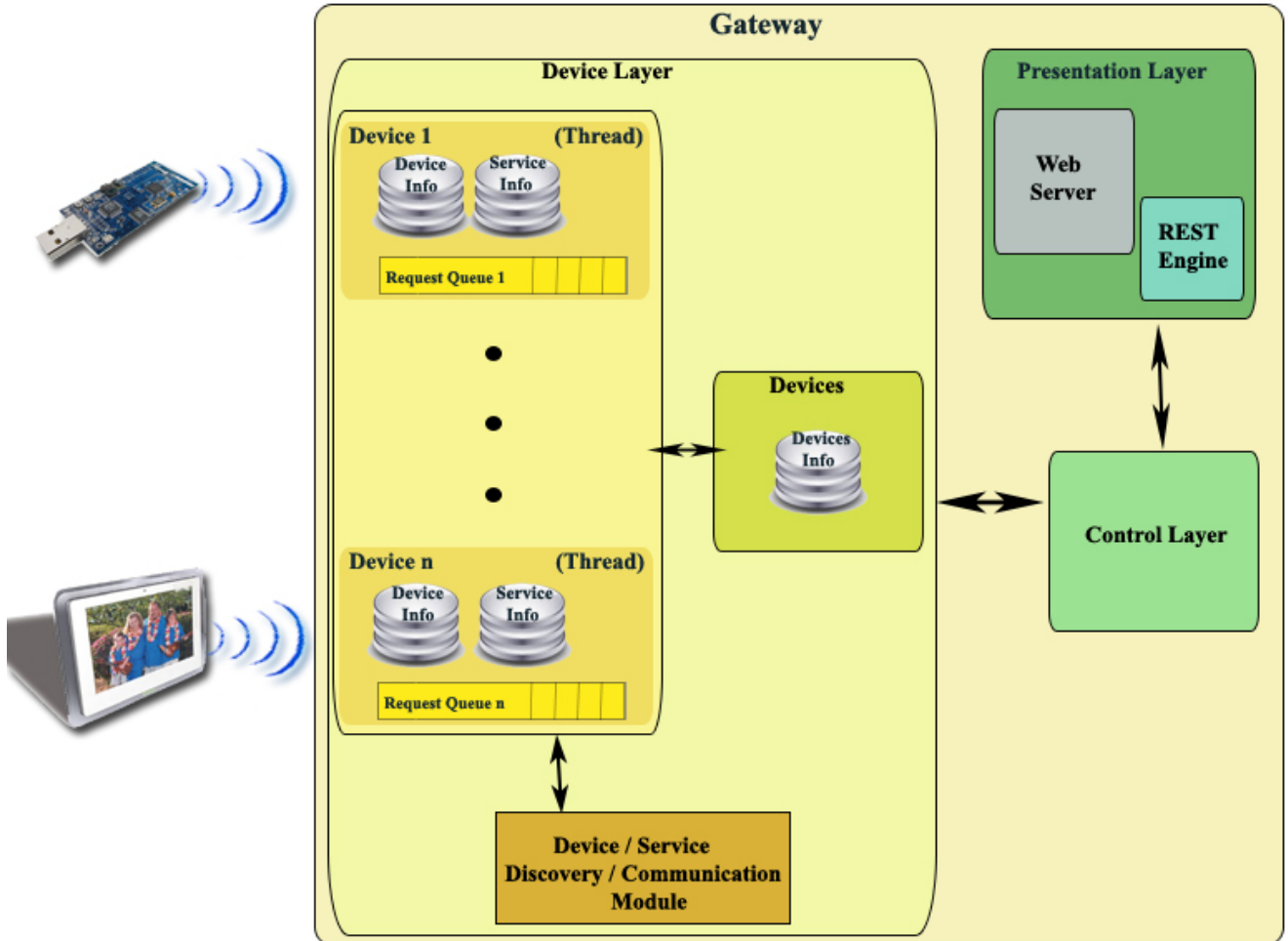


Figure 6.1: Gateway Architecture.

well as a list of the services supported by the device, along with their service description data. Because of the unstable environment of resource-constrained devices, we decided to introduce a *Request Queue* for every device, that would help in transmitting the different requests to the real physical device with some guarantees. Every new request made for a service which is supported by the device, should be temporarily stored inside that Request Queue. Requests are stored in a FIFO manner and each time a specified time interval passes (which depends on the device type and its ability to respond in a time limit), a new request is returned from the queue and is transmitted for execution at the (real) device. What actually happens during a request transmission is that the thread, calls the corresponding driver which supports this type of device and which lies inside Device/Service Discovery/Communication Module, as we already mentioned. After the response arrives, that module forwards the response to the appropriate device's thread to consume it. Since we must also deal with possible failure and transmission conflict issues, requests, after being sent should not be removed from the system, but remain inside the queue until the request is satisfied. Only then the system is free to permanently remove them. If the response does not arrive in a predefined time interval, the request is refetched from the queue and retransmitted. After a number of unsuccessful attempts to send

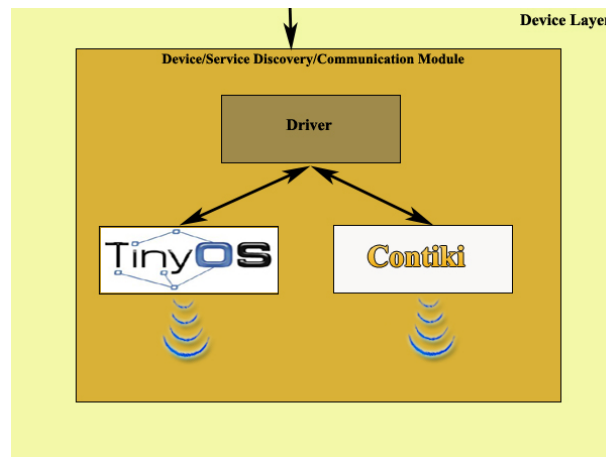


Figure 6.2: Device/Service Discovery/Communication Module Design. It has currently support for Contiki- and TinyOS-enabled sensors.

a request, the device is considered as failed and it is removed from the system.

Devices Module, located also inside Device Layer, is responsible for the interaction between the upper layers and the device threads. It maintains a list of all the devices which are binded to the system, as well as all the information covering the services they offer. Higher layers are directly informed through querying that module about all the necessary information concerning the devices connected to the system. In other words, it offers an abstract way of reaching the devices by means of a simple interface. It was designed with the goal in mind, future applications developed independently in higher layers, to be able to communicate between them and understand each other.

Control Layer is the soul of the system. It runs on the background all the time, maintains system's threads, initializes all the gateway's modules and checks that everything operates according to its specifications.

The layer that interacts with Internet users is the *Presentation Layer*. This layer is separated from the rest of the system since it is the most extendable part of it. Currently, there is a *Web Server* implemented, which gives access to Internet users manipulate devices through the services they offer. A *REST Engine* provides REST support to the Web Server, so the users can use REST architectural style to interact indirectly with the devices through their web browser. The Presentation Layer was completely developed by Samuel Wieland, during his master thesis [36].

The flexibility of our architecture facilitates the development of new applications inside Presentation Layer, which can take advantage of the gateway's structure to access and gain information from the devices supported by the system and create their own dedicated interfaces. For example, with a few lines of code, a Database could be created and used, where all the data received from the devices would be automatically saved and where users could select the data they are interested in, from that repository easily and fast. In a little more advanced case, local mashups of data can be implemented that offer an abstract view of the WSN functionality, as a composition of functionalities of individual devices supported by the gateway.

6.2 Mechanisms Involved

By means of the architecture we designed it was straight-forward the integration of some additional mechanisms, which could accelerate the performance of our system and increase its functionality. In addition, since we deal with devices that offer limited capabilities, it would be desirable to find some ways, through the gateway, to unload some of the burden which devices had to produce to stay in line with our requirements.

We present here the mechanisms we intergrated in our gateway, which can prove the endless possibilites of using a gateway system, as we propose, to increase efficiency and augment functionality of resource-constrained devices.

6.2.1 Retransmissions

Gateway operates in a distributed environment with high possibilities of transmission errors. These errors do not necessarily imply a device's failure. In the most common case, they happen because of a transmission conflict between two or more devices concurrently transmitting or because of packet loss due to interference from the environment. In these cases, we shouldn't simply conclude that a device has failed and delete it from the system, but in a first step we should retransmit our request to eliminate the possibility of a conflict. Only in the case when a number of requests remain unanswered we should deal with the case of device failure. The tactic of retransmitting the requests after specified time intervals in case of no answer, increases robustness and reliability of our system, since it eliminates errors that can naturally happen for the reasons discussed. Value of time interval (for sequential request transmission in Request Queue) must be defined with care, on one hand to avoid further conflicts (concurrent transmissions from gateway and device) and on the other to increase system's responsiveness to Internet clients, in case of some message's delivery failure.

6.2.2 Caching

Use of caching could be proved very useful in order to reduce the requests made to the devices. In case two requests from different Internet clients are made to the same device, requesting the same service to be executed, with the result not altering the state of the device (mainly in cases of GET requests, for example to get the current temperature measured by the specified sensing device), a caching mechanism could be employed to avoid asking the device twice for the same data with a consequence of wasting energy for unnecessary transmissions. A time interval must be specified which will define as reliable the last measurement, if the request was made inside that time boundary. Otherwise, the normal procedure will be executed and the sensing device will be queried for the service to be invoked. Care must be taken at the definition of the time interval. A trade-off must be taken in order to increase performance on one hand, but to retain reliability of the measurement on the other. Big intervals could make caching mechanism really powerful, but at the same time they could send to the user wrong measurements according to changes in resources' current true values.

We believe that use of caching would increase dramatically Internet clients' request response times and would reduce energy consumption at the devices.

6.2.3 Congestion Avoidance

Care must be taken when multiple Internet clients request services which reside in the same device. Their requests, can not be concurrently sent to the device, but they should be executed

one after the other, leaving time for the device to answer each request. Most services deal with some embedded sensors which measure some quantity. Time must be given to the sensors to take their measurements and to the radio system of the sensing devices to transmit the results of the measurements back to the gateway. To achieve fairness, the requests are transmitted to the devices in a FIFO manner. Requests that are pending to be sent, must remain stored in some data structure, ready to be sent when an amount of time passes. That amount of time should not be smaller than the regular amount of time needed by a device to answer a request and depends on device type. For example, devices with larger computing capabilities like mobile phones need less time interval between request transmissions, than resource-constrained devices like sensor nodes. By storing requests to be sent in later time when the device is not busy, we assure that no requests are lost because of congestion and we unload the device from keep answering to requests all the time. Additionally, we make more efficient use of the caching mechanism, which was previously described.

6.2.4 Failure Recognition

This scenario happens when a device becomes totally unreachable from the gateway. This is the case when the battery of the device is empty, which results in a failure of the device or when an obstacle prevents the messages from reaching their destination. It can also happen in the rare case where the device (due to temporary loss of communication with the current gateway) joins an other gateway and does not respond any more to requests made from the current gateway.

A mechanism must be developed that deals with this kind of errors and identifies immediately the absence of the device and the further lack of communication and interaction. It must be mentioned that this procedure holds only during devices Operation Phase. This identification should take place as fast as possible in order to remove the device's information from the gateway's data structures and to stop presenting this device to the Internet clients in Presentation Layer, which would lead to unnecessary faults and user inconvenience, due to the device's failure.

The countermeasure we introduced to handle this issue is based on regular *Aliveness Checks*, made from the gateway to the device. Aliveness Checks are actually very light-weight requests posed to the device to determine that it is still alive and operates normally. A time interval must also be specified in this case, which will inform the gateway to check for device's aliveness. This interval must be specified with care: long intervals would reduce failure identification ability and subsequently reliability of the system while small intervals would increase energy consumption. Also the value of the time interval depends heavily on device type. In a case of a device with big energy constraints, we should "sacrifice" some checks, to maintain the battery for a longer run.

A smart idea to assure, during gateway's operation, that a device is alive is when a normal service request is made to the device and a response is received. We are sure then, that the device operates normally and we don't need to make an additional check to test it (so we reset the aliveness timer). On the other hand, when a response is not received in a specified amount of time, we should be alerted and immediately check the possibility of a device failure.

In any case, if the gateway does not receive any answer from the device after a number of unsuccessful transmissions (which number depends on device type and on the level of reliability we seek of our system) for a request, then it determines that device has failed operating for some reason and it removes it from its list of devices.

6.2.5 Failure Masking

This mechanism shows the real potential of our design. In the not rare scenario where a device fails, with some requests pending in its Request Queue, the gateway, instead of immediately producing an error message to the corresponding Internet client who made the request, searches its data structures for another device with capability of providing the same service. It can be observed here how important it is to face services as resources. Using URIs to name services offers the possibility to easily find a similar service with the one originally requested. In the case another device is found, it is invoked, the response is produced and forwarded to the corresponding Internet user, without him noticing device's failure. The error is in that scenario, effectively masked. Device type in this scenario makes no difference, since the mechanism works in a higher level. For example, a TinyOS sensor failure could be compensated by a Contiki sensor device, offering the same or very similar functionality.

In case an other device with a similar service is not available, the gateway tries to find an older, cached value from the failed device's caching mechanism to compensate for the failure and send it to the Internet client who originally made the request (with a higher probability of error between older measurement and current real value of the quantity supposed to be measured). In that case, of course, the system warns the Internet client about error possibility of the measurement.

6.2.6 Statistics

After we make use of a service offered by a device a considerable amount of time, after that service is invoked from Internet clients a number of times, we have in our possession a series of measurements concerning values of the quantity measured by the service. We can take advantage of these measurements to extract useful information that can effectively represent that resource's activity and can give a detailed report to the interested Internet clients, who could, after being presented by this information, avoid re-querying the service to get additional data but remain satisfied with the data shown to them. This information could be a mean value of the measurements or more advanced a graph representing values measured as well as the timestamps when these measurements were taken. New measurements could involve differences from the mean value and standard deviation information could be easily calculated.

These statistics could also make the eventing mechanism more efficient, by adaptively identifying an event (for example a dramatic change in the current measurement in reference to the mean value of all the previous measurements could be effectively recognised).

6.2.7 Adaptive Streaming

One of the requirements we posed for the gateway is support for multiple Internet clients and reliable handling of their possibly concurrent requests. Special care had to be taken when multiple streaming requests were concurrently posed for a specific service (with streaming capabilities), offered by a specific device. In order to deal with this problem, we used a simple "adapting" approach of the requests to satisfy all the interested clients, based on simple mathematics. If a device was already sending some streaming for a particular service (measuring some particular quantity) and an other request arrives demanding some streaming from the very same service, we force the device to follow the request with the lower time interval demand between sequential streaming packets, but we also adapt the number of total streaming packets to be sent to adapt with the maximum (of both requests) total amount of time. So we have both clients satisfied. This simple approach can be followed when more users make

concurrent requests for the particular service.

We can in detail see how this procedure works in Listing 6.1. Basically in this method, the new request is adapted in case there is already a streaming procedure triggered for that service on that device and the adapted request is sent to the device, in order to satisfy both Internet clients at the same time.

```
1 public Request synchronizeEventingRequests(Request newRequest){
2     newInterval    = newRequest.getDelayInterval();
3     newIterations  = newRequest.getNumberOfIterations();
4
5     if(newInterval > currInterval){
6         intervalDiff = newInterval/currInterval;
7         currIterations = max(intervalDiff*newIterations),
8             currIterations);
9     }
10    else if(newInterval == currInterval){
11        currIterations = max((newIterations, currIterations);
12    }
13    else{
14        intervalDifference = currInterval/newInterval;
15        currIterations = max((intervalDiff*currIterations),
16            newIterations);
17        currInterval = newInterval;
18    }
19
20    // create the new adapted Request to be sent to Smart Device
21    List<Object> adaptedValues = new LinkedList<Object>();
22    adaptedValues.add(currInterval);
23    adaptedValues.add(currIterations);
24    Request adaptedRequest = adaptRequestParameters(adaptedValues);
25    return adaptedRequest;
26 }
```

Listing 6.1: "Code for the concurrent streaming request adaptation procedure."

We must specify here what "event checking" means in our gateway, according to the logic we developed our system with. We believe that every different person has a different concept of an event in his mind. Hence, it should be a matter of the Presentation Layer to deal with event filtering. In other words, we expect from applications developed in Presentation Layer, to take care of this issue.

We merely developed in Device Layer a simple forwarding mechanism, which forwards all the measurement values of the "candidate events" to the higher layers. We try also to identify events, even when no streaming is requested. For example, we check every GET response arriving at a device's thread for possible events, basically by forwarding also these values to the upper layers).

6.3 Synchronous/Asynchronous Translation

One of the major criticism against the use of the REST architectural style in the area of sensor networks spans from the fact that HTTP was designed for synchronous requests. This

model would generally not suit the requirements of most sensor network applications. We intend to propose in this section a solution to map an asynchronous operation which happens on our gateway's Device Layer, with an HTTP request/reponse model, which works synchronously at the Web Server, in Presentation Layer.

We propose the use of *synchronizers* to transform asynchronous behavior into a synchronous one. For each request from the Web, the gateway will create a new thread to which an unique message-token is associated (*requestID*). The *requestID* is sent into an asynchronous request together with the token, to the device thread and obtains a lock on a unique synchronizer token (identified by the message-token). As soon as the response to the message-token arrives, the synchronizer lock is searched and the waiting thread is awakened. This will finally send back the HTTP response to the initial client. Of course, the duration of time it takes to complete the request (thus the time the client must wait to receive the response) is highly dependent of the nature of the sensor network the request is sent to.

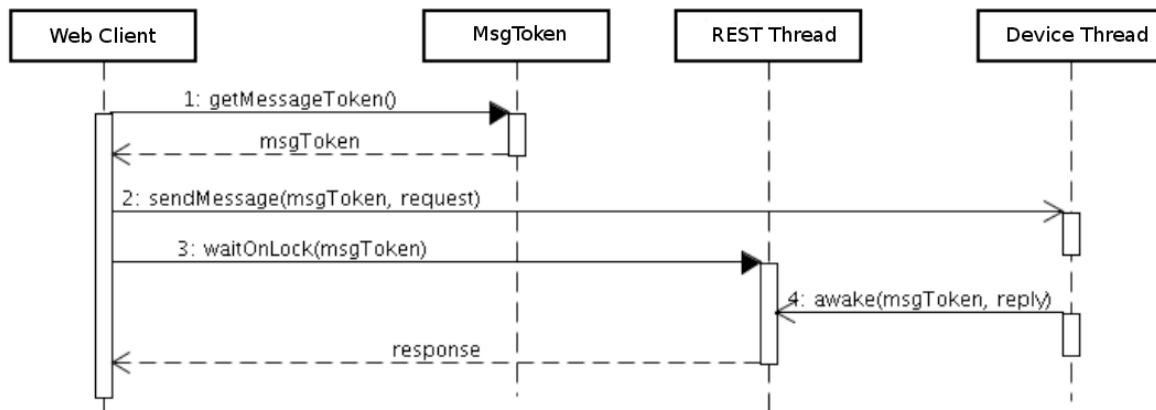


Figure 6.3: Translation between Synchronous-Asynchronous Operation.

Figure 6.3 shows the sequence diagram and Appendix B.1 shows parts of the implementation.

1. The client obtains a unique message token.
2. Together with the token, the request is sent asynchronously to the responsible device thread.
3. The client obtains (with the message token) a lock on the REST thread and puts itself to sleep.
4. The message arrives from the device thread. The middleware awakes the client's corresponding REST thread together with the message-token, and the synchronous execution continues.

The inspiration concerning this approach as well as most of the implementation part of it, must be credited to Samuel Wieland, during the integration of our master thesis together.

REST-enabled Sensor Devices

Based on the REST-oriented patterns we presented in Section 5, we developed software for Tmote Sky sensor devices operating with TinyOS, and on Tmote Sky sensor devices operating with Contiki operating system.

Contiki and TinyOS-enabled sensors, have the capability of following the Device Discovery pattern we showed in Section 5.1, in order to announce themselves to the gateway and transmit to it their device and resources' characteristics. Furthermore, their resource description data are compatible with REST style, as we illustrated in Section 5.3.

In addition, these sensor devices use the interaction types we defined in Section 5.4. A Web client can transmit a request and receive a response following the Request/Response model and he can start streaming procedure on resources which declare themselves capable of producing streaming. We didn't include the third interaction type (Event-Based model), - even though our gateway supports it by design - where the client can be informed about events that happen spontaneously, after he subscribed for them through the gateway. The reason is that our sensor devices do not have any significant, dedicated task to monitor, which could be some movement in the area or the opening/closing of a door.

We included in TinyOS sensor devices four services: measuring temperature, measuring light radiation, setting the sensor's LEDs and measuring humidity. For Contiki, we implemented these four services, but additionally we integrated a resource informing about the battery voltage of the device, and a resource which returns the radio transmission power used by the device.

7.1 Sensor Interaction

We illustrate here how we adapted the general REST-oriented patterns for the resource-constrained environment of wireless sensor networks. Concepts are the same, but the implementation mechanisms are different in order to reduce energy consumption by data coding and to assure some advanced functionality by some retransmission mechanism added to sensor devices, during Device Discovery procedure.

We decided to add this section in the paper, even though we described the general procedure followed in Section 5.1, because here we show in detail how the actual interaction was implemented. An implementation for Bluetooth devices such as mobile phones, or computers with Wi-Fi, could follow a different implementation direction, to take advantage of the physical link's specific characteristics. For example, an interaction pattern for a device with TCP/IP support, would possibly not require retransmission mechanisms, since that is provided "for free" by TCP protocol.

In TinyOS, we made use of Active Messages for our communication needs, while in Contiki, we took advantage of RIME communication stack for our interactions. In both cases, we used the native messaging offered by these two operating systems, through IEEE802.15.4 network standard.

7.1.1 Message Templates

In general, Message Templates must be defined both in Contiki and TinyOS. These templates are necessary to be declared, so both sides can understand the semantics of the data being exchanged. Both the gateway and the sensor device, must "agree" on advance about the structure of the message, which will be transmitted between them.

In TinyOS, we defined the Active Message's structure to be as following:

```
nx_struct tinyOS_msg {
    nx_uint8_t nodeid;
    nx_uint8_t subject;
    nx_uint8_t data[PACKET_SIZE];
};
```

where *PACKET_SIZE* is the default maximum size of an Active Message. *nodeid* is the unique address of the sensor mote sending the message, *subject* is the message's content description and *data* is an array where the contents of the message are kept, as well as the result of the request's execution (eg. a requested temperature value).

In Contiki, we defined the RIME message structure to be as following:

```
struct contiki_msg {
    uint8_t subject;
    uint8_t data[PACKET_SIZE];
    uint32_t result;
};
```

where, in this case, a slightly different approach is followed. Node unique ID is encapsulated inside array *data*, while there is a distinct section for *result*. The other parameters have the same semantics as before.

In general, for both operating systems, we decided, *subject* to be just a single character, in order to reduce energy consumption. All the information are transformed in *short* data type in order to be efficiently transmitted by the wireless media (with only exception the *result* value in Contiki case, where the system requires an *integer* data type to be employed). Encoding/decoding is performed in both ends, to ensure character-integer translation.

7.1.2 Operational Phases

In this subsection, we examine the different phases each device changes during its operation and cooperation with the gateway. These phases follow, of course, the Device Discovery pattern, but we present them to show specifically what happens inside the sensor devices that participate in this procedure. We focus on the implementation logic inside the sensor device's software, to indicate our specific approach to effectively touch this issue. We have split the whole procedure inside device level in four different phases, which we will introduce in the following lines.

Sensor ID	Sensor Name	Sensor Type	Operating System used
7	M4AOCFAO	TmoteSky sensor	TinyOS
178.63	M4AOCFB1	TmoteSky sensor	Contiki

Table 7.1: Example of Device Description Data in TinyOS and in Contiki.

Scanning Phase

This is the initial phase, when a device seeks for a gateway. Instead of following the general REST-oriented pattern, where a broadcast message is indicated, we decided to employ unicast HELLO messages, setting statically the address of the message's destination, which is obviously the gateway's address. Gateway's address is hard-coded inside *GATEWAY_DEFAULT_ID*, which resides in sensors' software. We adapted this approach to save energy from broadcasting messages that target the gateway, but must be received (without basically reason), from the other neighboring remote sensors.

Each HELLO message is sent every 5 minutes and consists only of the address of the sensor mote sending the message and the character *H*, inside the *subject* of the message. As soon as the device receives an acknowledgment message from the gateway, it enters *Device Description Phase*.

Device Description Phase

In this phase, the device sends description information, which, in general, represent the device. In Table 7.1, there are two examples of possible device description data, one for TinyOS and one for Contiki.

In case a device description message is lost and no acknowledgment returns from the gateway in a specified time interval, the device retransmits the message, for a predefined number of times, set in \mathcal{J} in our case, after a small delay of \mathcal{J} seconds for every attempt. If no acknowledgment arrives, in any of these attempts, the device assumes that it has lost connection to the gateway and returns to its initial scanning procedure, broadcasting HELLO messages, seeking again for a gateway to connect to. In case an acknowledgment message arrives, in any of the attempts, device concludes that the gateway has received its device description information, so it starts *Service Description Phase*.

Service Description Phase

This is the phase when the sensor device starts, sequentially, to send all the service description information it has to the gateway. Every service description message is encapsulated inside a single message. Due to the small size, which is by default provided for every message by TinyOS and Contiki native communication stacks, it was not possible to include a number of service description information in a single packet. Therefore, in case a sensor device offers, for example, five services, it must send sequentially five service description packets to the gateway. In Table 7.2, we can see all the services and their description information, that are offered by the TinyOS and Contiki software we developed.

A retransmission mechanism, similar to the one described in Device Description phase, exists also here. Whenever a service description message is lost and no acknowledgment arrives from the gateway after an amount of time, the device tries to resend the message 5 times in total, after an interval of 5 seconds expires for every attempt. We chose a bigger number

Name	Description	MIME type	Capabilities	Parameters
Temperature	measurement in Celcius	text/plain	GET,EVENT	-
Humidity	% value measured	text/plain	GET,EVENT	-
Light	set RGB Leds	text/plain	POST	color (char)
Radiation	measure light radiation in Lux	text/plain	GET,EVENT	-
Battery	voltage in Volts	text/plain	GET	-
Power	Radio Transmission in mW	text/plain	GET	-

Table 7.2: Services offered by Contiki and/or TinyOS sensors along with their description information.

for attempts and delay, than in the case of the device description message, since it is more possible for a service description message to get lost, as it is a procedure that lasts larger time than the Device Description Phase. We also observed that by assigning smaller values to these variables, many transmission conflicts appeared in scenarios when more than ten sensor devices were trying concurrently to connect to the same gateway. In Section 8.2.1, the reader can see in detail, after our experiments, the optimal values for these variables, in order to achieve the best performance during Device Discovery procedure.

We must note that this procedure holds for each different service description message that the device must send. As we previously saw, in case of no acknowledgment after all of these attempts, the device assumes that it has lost connection to the gateway and returns to Scanning Phase. In case of an acknowledgment message from the gateway for a specific service message in any of these attempts, device trasmits the next service description message. After all the service description messages are sent and they are all successfully acknowledged by the gateway, device enters *Working Phase*.

Working Phase

After the Device Discovery phase is over and the device has sent all its device and resource information to the gateway, it enters Working Phase. In that phase, it allows the gateway transmit requests to it. We must note that only the gateway, with which it has interacted together during discovery phase, can send requests. All other requests coming from other gateways or sensor devices are simply discarded.

No more retransmission ability is provided by the sensor devices. From now on, any message losses are responsibility of the gateway to take care of. The device has simply the duty of answering requests and performing streaming, after dedicated streaming requests from the gateway.

Disconnection Identification

A sensor device during Working Phase, if it notices that it has not received any request from the gateway for a considerable amount of time, determines that it is not any longer connected to it and enters again Scanning Phase, seeking for a gateway device. This time period, until the device determines that the gateway is unreachable, is correlated to the time interval of the *Aliveness Check*, which is regularly performed by the gateway (see Section 6.2.4). After that time interval passes, device can safely determine that it is not any more connected to it. A time interval of 2 minutes is assigned to the gateway to perform Aliveness Checks, for every

identical device attached to it. A device identifies disconnection in case 5 minutes pass and it receives no sign of existence from the gateway.

To reduce energy consumption, gateway and devices, don't exchange aliveness messages when normal requests are regularly posed from Web clients. Both requests and responses are a sign that the gateway and the corresponding device which answered the request, operate normally and they are both still "alive" and connected.

7.2 Sensor Deployment

In order to support devices with TinyOS or Contiki, a setup shown in Figure 7.1 is necessary. To achieve communication between a gateway and a number of remote sensor devices, a sensor mote operating as a *Base Station* must be connected to a USB port of the PC where the gateway system is installed. Base Station's functionality is to forward IEEE802.15.4 wireless packets from/to the Gateway, through the serial-over-USB port, acting like a proxy between the two sides.

Both TinyOS and Contiki, need their own Base Stations to properly operate. Since we wanted to integrate both sensor operating systems in our implementation, we used two sensors connected on two USB ports of our laptop.

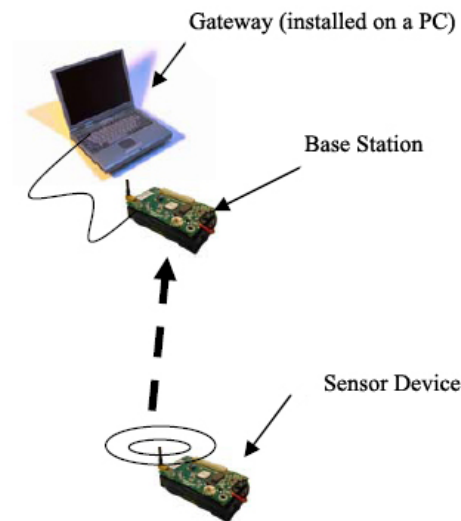


Figure 7.1: Necessary Infrastructure for TinyOS/Contiki support.

Appendix A.1 shows the installation procedure to enable our REST-oriented software on TinyOS devices. Appendix A.2 presents how to install our RESTful software on Contiki devices.

7.3 Error Handling

In this last section, we deal with some possible problems that can appear during sensors' operation. These errors have mostly to do with message exchange, because of the distributed nature of our system. We describe in the following subsections each possible problem, showing

at the same time the countermeasures we took in each case, to eliminate the failure possibility of our system.

7.3.1 Same Identity

This conflict can be caused only in the case of TinyOS-enabled sensors, only when two or more sensor motes are programmed with our TinyOS software using the same ID during installation. In Contiki this can not be a problem, since every sensor has a different ID, provided to it during its manufacturing process. Although extra care must be taken not assigning the same ID to two or more different motes, gateway makes a check for duplicate device registration during Device Discovery phase. Only once, a device with a specified ID can be added to the system. In the case where two or more motes use the same ID, they will both be represented by the same data structure in gateway's memory and they will be treated as if one device was involved.

7.3.2 Sensor Interference

We avoid, from the very beginning, sensor-to-sensor interaction by forcing the sensor motes seek gateways only at a *GATEWAY_DEFAULT_ID* address, which is defined in our implementation as *1* in TinyOS case and *41.41* for Contiki. The only sensor-to-sensor interaction allowed, is between the remote sensor mote and the Base Station. In addition, devices accept messages only from that specific *GATEWAY_DEFAULT_ID*, ignoring all other messages from neighboring sensor motes.

7.3.3 Transmission Conflicts

This conflict is caused during Device Discovery operation, when a description message is lost during transmission from the device to the gateway. This description message can be either a device description message or a service description message. As we already mentioned, we involve a retransmission mechanism to take care of this issue, by retransmitting the current description message for a number of times, after a specified time interval passes and no acknowledgment is received from the gateway. In case all the transmission attempts for the specific message fail, the device concludes that the gateway is unavailable and returns again to Scanning Phase.

7.3.4 Device Registration Error

This problem appears in the case when a device sends description messages without being registered in gateway's lists. This can happen for example when the gateway loses the information it kept about devices. We take care of this issue by forcing the gateway stop acknowledging description messages from unregistered devices. In that case, after a number of unsuccessful requests, the device returns to Scanning Phase.

7.3.5 Request/Response Message Issues

During device's Working Phase, loss of a request or a response message should be dealt exclusively by the gateway and not by the device. We already presented in Section 6.2, mechanisms for facing this kind of problems. The only case where a device should be responsible, is when a request is sent from a gateway for a service invocation, without the device being registered

to that gateway (following the normal Device Discovery procedure). In that case the request is simply ignored.

7.3.6 Gateway Selection

Gateway selection from the device point of view, is currently a static procedure, since the device in Scanning Phase seeks available gateways by sending messages only at a specified *GATEWAY_DEFAULT_ID* address. This process should be dynamic and the device should have the possibility to broadcast HELLO messages and select the best gateway found (according to some criteria eg. signal strength or physical distance) to connect with.

Gateway unique naming is a problem that should be solved in a higher level, in scenarios where a number of gateways form a network and interact together to achieve some global goal. In the single gateway level where we are, this causes no serious problems. Nevertheless, it must be considered for large deployments, where several gateways are present.

Evaluation

In this chapter, we attempt to evaluate our system experimentally. We created different scenarios to test our infrastructure under real-time and heavy workload circumstances. Through these tests, we measure the performance and reliability of our system. At the end of the chapter, we also discuss about bottlenecks that exist in our system, which could have reduced the total functionality of our model.

8.1 Experimental Setup

The experimental setup used to evaluate our framework is shown in Figure 8.1. The gateway software has been installed on a laptop (Intel Core Duo 2.2 GHz). Around the gateway, we deployed 14 tmote Sky sensor nodes running our RESTful software for embedded devices, half of them using Contiki, and the other half TinyOS. Tmote sky's specifications are illustrated in Section 4.1. Two additional sensor devices were plugged in the USB ports of the laptop, to serve as Base Stations - one for TinyOS and the other for Contiki. Base Station's general operation is briefly described in Section 7.2. Also, every sensor, for the sake of the experiments, offers four different services: measures temperature, changes the color of its LEDs, measures light radiation and at last, measures the levels of humidity of the environment. Sensor devices, through the experiments, provide an equal number of services, to compare the performance of TinyOS and Contiki operating systems in general.

8.2 Experiments

Here we describe the experiments we performed to test the gateway with actual scenarios. We prepared four different experiments, each covering a different aspect of the whole system. First, we examine the timing needed at the Device Discovery phase with different number of sensors. Then we measure the performance of our gateway and afterwards we compare our gateway with a proxy, which simply forwards requests to sensor devices. At last, we compare TinyOS and Contiki operating systems, focusing on their operational abilities.

8.2.1 Device Discovery Phase

In the first experiment we were interested in measuring the time required by the Device Discovery phase to complete, that is how much time it takes for a variable number of sensors operating concurrently to send their device and service description data to the gateway.

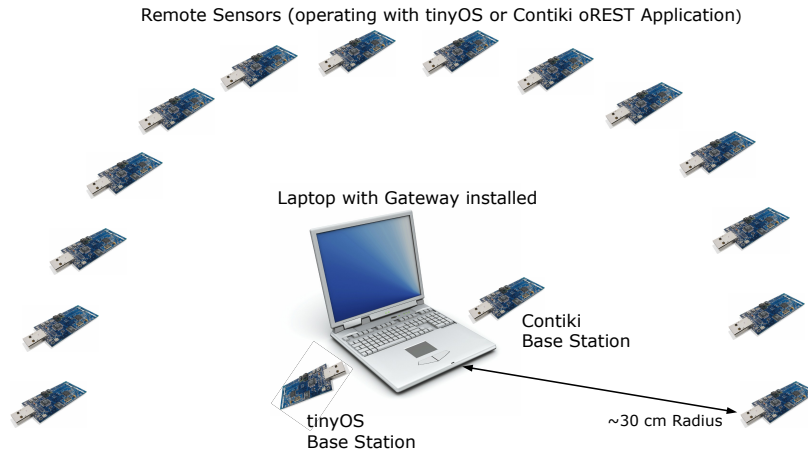


Figure 8.1: Experimental Setup used for Evaluation purposes.

In general, every device and service description message is encapsulated, in TinyOS, in a single message of 40 bytes size, while in Contiki, every device description message is embedded in a single message of 29 bytes size and every service description message in a message of 42 bytes.

Figure 8.2 shows the measurements performed and Table 8.1 shows the most efficient values that should be given to some system variables of the sensor device's RESTful software, in order to achieve the best performance during Device Discovery procedure. These variables and their semantics, were described in Section 7.1.2.

We considered adjusting these values, examining every scenario on its own, because during the experiment we noticed that as the number of devices was growing, the high percentage of transmission conflicts had a severe impact on the total amount of time until the discovery was over. By increasing the retransmission interval time for device and service description messages, system's performance was accelerated.

In the specific case when 14 devices were deployed, we observed that some devices were interrupting their discovery procedure, returning to initial Scanning Phase. That happened as the number of unsuccessful retransmissions of some description message, forced the device to conclude that the gateway was not available any more. To eliminate this possibility, we increased the number of service retransmission attempts.

Concerning Table 8.1, *DevNum* is the number of devices at the current experiment, *ScanInterv* is the retransmission interval between sequential HELLO messages from the sensor device, *DevInterv* is the retransmission interval at device description message transmission, *ServInterv* the interval for service description messages, *DevRetr* the total number of retransmissions for device description messages, before the device returns to Scanning Phase and last, *ServRetr* the retransmission attempts for service description messages.

Discussion: We can observe that for a small number of devices (at most eight), less than a minute is sufficient for the phase to complete, which is considered a satisfactory amount of time.

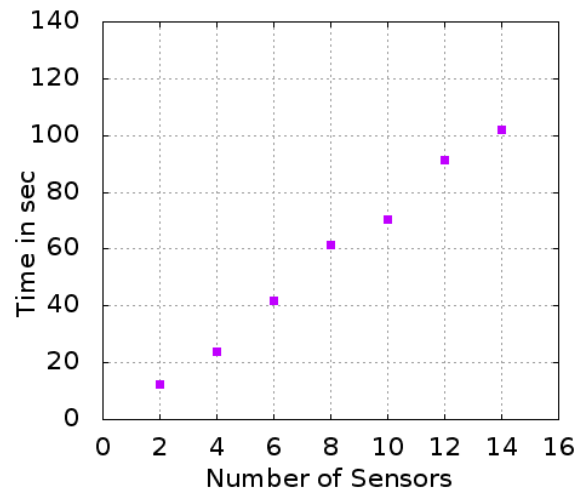


Figure 8.2: Time needed for Device Discovery Phase with a variable number of sensors.

DevNum	ScanInterv	DevInterv	ServInterv	DevRetr	ServRetr
2	5 sec	1 sec	1 sec	3	3
4	5 sec	3 sec	2 sec	3	3
6	5 sec	3 sec	3 sec	3	3
8	5 sec	3 sec	4 sec	3	3
10	5 sec	3 sec	4 sec	3	3
12	5 sec	3 sec	4 sec	3	3
14	5 sec	5 sec	7 sec	3	5

Table 8.1: Parameter Value Assignment for optimized operation of Device Discovery Phase.

When more devices try to connect to the gateway this interval grows linearly and it reaches 102 seconds, in case of 14 devices. We believe that this is an acceptable amount of time, as this procedure will happen most probably only once during gateway's operation and only at the beginning of its execution.

8.2.2 Gateway's Performance

In this experiment, we evaluate the gateway's performance when concurrent requests are posed from multiple Web clients. For that reason, we simulated a testing scenario where a variable number of Web clients makes requests randomly on devices already available through the gateway, at random services they offer, at random times, with frequency of one request/minute/client.

A request could also target enabling some streaming from the device to the gateway and we tried to constraint this possibility in order to have in most of the cases two devices sending streaming every minute, with time intervals of 10 seconds for every different measurement. All the request and response messages were encoded, in TinyOS in single Active Messages of 10 bytes each and in Contiki, RIME single message packets of 10 bytes size were also used.

We enabled also caching mechanism and we assigned a valid time interval of 2 minutes, which is a logical amount of time in real-time cases. The total time for every simulation is 15 minutes. We tested scenarios with 6, 10 and 14 sensor devices when 5, 10, 20, 50 and 100 concurrent web clients use the system.

The measurements concern mean response times and their corresponding caching success percentages and can be seen in Figure 8.3.

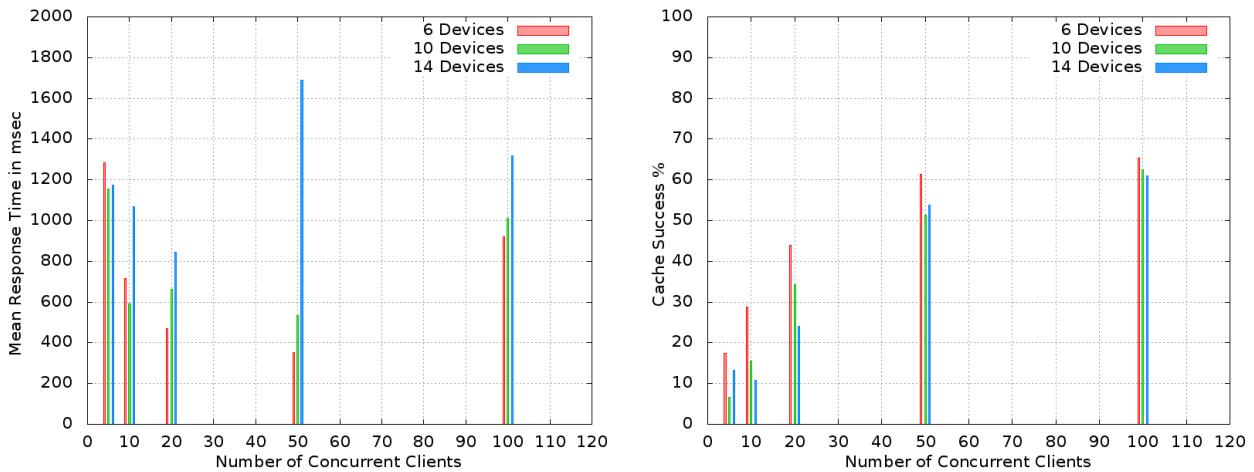


Figure 8.3: Gateway's mean response times when multiple web clients are concurrently making requests (graph on left) and the corresponding caching success percentages (graph on right).

Discussion: To test gateway's performance, we decided to focus on the mean response times during its operation. We believe that response time is the most appropriate parameter in order to judge the efficiency of the gateway, since in general Web clients are mostly interested in the total amount of time until their requests are answered.

By observing the results, we can conclude that the gateway is generally stable and performs well even with 100 requests per minute. In that case, mean response time is lower than a second when 6 and 10 devices are used, and 1.3 seconds, when 14 devices are used. The reader could wonder why an increased number of devices leads to bigger response times. The answer in this question lies in the caching mechanism we implemented in our gateway and which is capable of performing better when less devices are used, since the probabilities of finding a cached value are generally increased.

There is a great correlation between the mean response times and their corresponding caching success percentages. A bigger percentage of cache "hits" improves significantly the mean response time, since the system does not need to follow the expensive in time procedure of putting the request in the request queue of the appropriate device, transmitting it, waiting for device to execute it, receiving the response and forwarding it back to the client. It just answers the request fast with the cached value. We must admit that caching plays a significant role in gateway's overall performance and is considered an important factor of the system.

The best response times appear when 50 requests are produced per minute, because in that case the caching mechanism works in full coordination with the request queues. When the requests are doubled, even though caching operates still well, there is a small congestion of requests in request queues, a phenomenon that leads in larger mean response times. The complexity of our system as well as the considerably large number of parameters which play an important role during gateway's operation, are the main reasons for the non-linearity of the results in the graphs.

In the simplest case, when only five requests are made from Web clients every minute, the response needs around 1.2 seconds to arrive in all different scenarios, with a variable number of devices. Considering the whole procedure and the translation from the synchronous environment to the asynchronous and vice-versa, that amount of time is judged as satisfactory, even though there is, still, space for improvement.

8.2.3 Gateway Vs Proxy

As we posed in Section 2.3 the question of the necessity of a gateway towards the direction for the *Web Of Things*, we decided to include an experiment in our evaluation procedure, where we would compare our gateway with a proxy, which simply receives requests from web clients and immediately forwards them to the devices without any additional mechanisms involved (in contrast to the gateway's case). Someone could argue that a use of a gateway is not necessary with today's sensors' technological standards and that a simple proxy would be sufficient for this job. We are optimistic that our measurements will help us answering this question.

To test this experiment, we made use once more of the simulator we developed in the experiment of Section 8.2.2. We used the same metrics as before, during the test cases. We tested again a large number of scenarios, covering cases with a varied number of concurrent Web clients making use of the system.

In Figure 8.4 we can see a comparison concerning response times in both cases, gateway and proxy, using a variable number of 6, 10 and 14 Devices. In Figure 8.5, the reader can see the request failure percentages, when the proxy was used. There were no request failures, in the cases when the gateway was used. Also for this experiment, we used the same numbers for devices, namely 6, 10 and 14.

Discussion: Comparing performance between gateway and proxy, we can see from the corresponding graphs that when a small number of requests is sent from Web clients per minute, the proxy performs similarly to the gateway (even though gateway maintains better response times in all the tests). However, as the requests increase, the proxy can not follow the gateway's operational standards and this fact becomes obvious in the case of 50 and 100 requests per minute, where the proxy is impossible to handle the heavy workload.

Generally, increasing the number of devices, slightly reduces the gap in mean response times, between the two entities. The reason is because by adding more devices, caching percentages in the gateway are reduced, and that leads to a small decrease in its performance. Another conclusion that can be extracted by observing the graphs, is that the proxy operates better

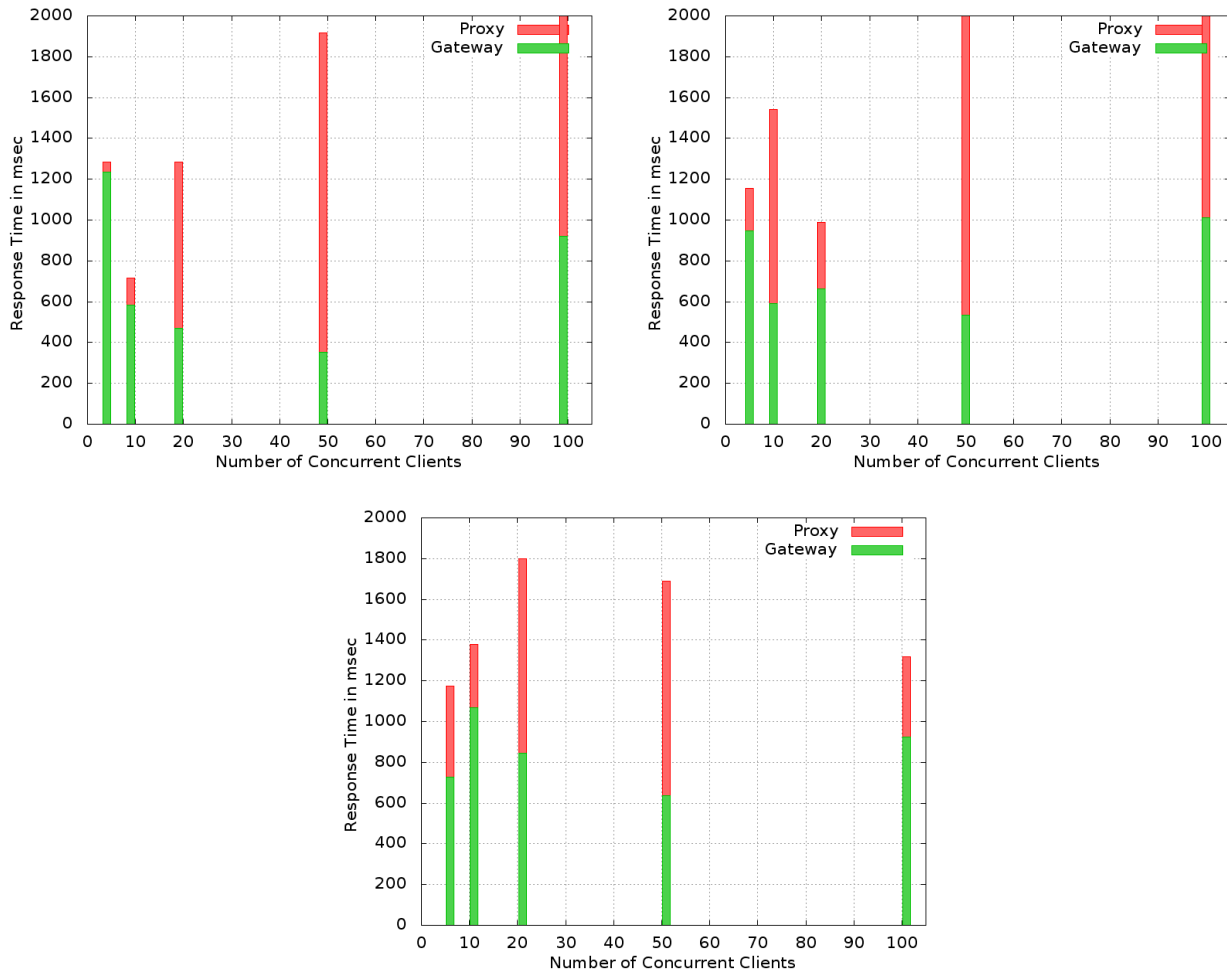


Figure 8.4: A comparison between Proxy's and Gateway's response times when multiple Web clients are concurrently making requests. A variable number of devices is used in every case, namely 6 (graph on left, top), 10 (graph on right, top) and 14 (graph on bottom)

when increasing the device number. This makes sense since the proxy, having only forwarding capabilities, maintains some improved functionality when the number of devices increases, because pending requests for every device are not congested and concurrently transmitted with high probabilities (resulting in transmission conflicts and inability of sensor devices to follow the proxy's request transmission frequency).

The use of a gateway in our implementation, created some additional demands, in reference to reliability, during the indirect interaction between Web clients and sensor devices. Considering reliability during our experiment, we discovered that in the case of the gateway, we didn't lose a single request during its operation. This was not the case with the proxy, where, as we see from the corresponding graph, as the request frequency is increased, the failure percentage is also significantly increased, having in the heaviest scenario of 100 requests per minute, more than 50% probability of failure for every single request. In the scenarios where 5 or 10 requests are produced every minute, however, we see that the failure probability is below 20%. So, in the proxy case, a single retransmission ability would be enough to mask all the failures for these, not so demanding, operations.

Based on these general observations, we can safely conclude that if we want to develop a real, concurrent, multi-user application, we definitely need a "smart" gateway system. Proxies,

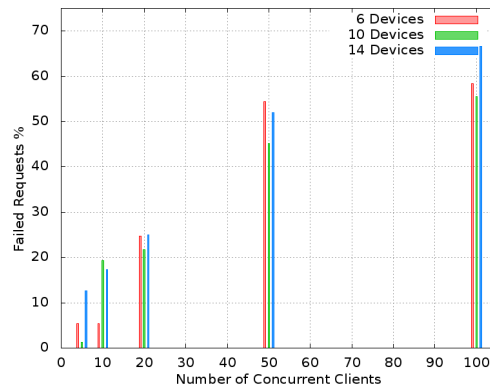


Figure 8.5: Proxy’s failure percentages when multiple Web clients are concurrently making requests.

although comparable with gateways in light workload scenarios, are inappropriate for heavy cases. With today’s Web applications, where standards like reliability and end-user satisfaction are required in a high degree, failure probabilities that appear in proxies are not acceptable and directions towards advanced gateway systems are inevitable.

8.2.4 TinyOS Vs Contiki sensors

Our last experiment concerns a performance comparison between Contiki and TinyOS sensor devices, during their operation inside the simulation we developed specifically for our experiments (see Section 8.2.2). We decided to include this metric, mainly for matters of completeness in our research. Furthermore, during the previous experiments we observed that Contiki devices presented some unexpected behavior sometimes, which led to reduced performance of the system. For example, they needed relatively longer times for request answering and they were sometimes resetting in the middle of the operation without a logical cause, causing extra delays for the system.

To test this case, we deployed a predefined number of 10 sensors, according to the experimental setup we described in Section 8.1, but this time we didn’t split the sensor device software installation, half in TinyOS, half in Contiki, but we rather used only one type of software for all the sensor nodes. For both versions of our RESTful software for embedded devices, we committed experiments with 5, 10, 20, 50 and 100 Web clients, making concurrent use of the system. We left the same parameter values, as we had in the previous experiments, namely caching mechanism interval of 2 minutes and simulation duration restricted in 15 minutes.

The results of this experiment, measuring once more mean response time in every different scenario, can be graphically seen in Figure 8.6.

Discussion: In light workload cases, when 5 or 10 Web clients, were making use of the system, Contiki-enabled devices operate slightly better than TinyOS-enabled sensor nodes. This time difference oscillates approximately between 600 msec. When the workload increases heavily, namely when 20, 50 and 100 concurrent Web clients, pose requests to the system with frequencies of one request per minute, it becomes obvious that TinyOS sensor devices perform significantly better than Contiki sensors. In the case of 50 clients, time difference reaches 2 seconds, which is a very big amount of time, considering our previous results. In the heaviest workload scenario, when 100 Web clients exist, Contiki sensors by themselves are not able to cope with the demands of the test, having mean response times more than 10 seconds.

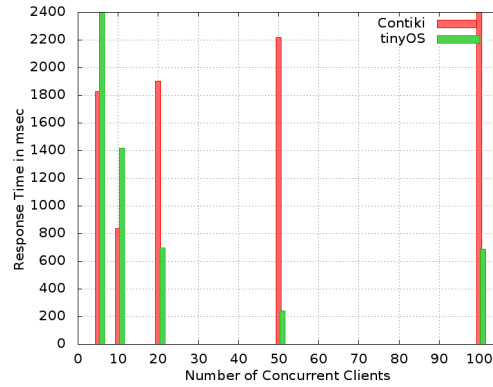


Figure 8.6: A comparison between Mean Response Time results in the case when only Contiki or only TinyOS sensor devices were deployed during the tests.

We can conclude that although Contiki sensors present better functionality in simple cases, TinyOS-enabled devices are much more reliable and efficient in the long run. We believe that this general inefficiency of Contiki sensor devices, in demanding scenarios, is caused by design, since they try to employ a lightweight protothread environment (see Section 4.2.2) in their architecture to accelerate performance, which probably brings some inadequacies when the device is really frustrated.

Nevertheless, we must credit Contiki for this issue, since it is under continuous development and it doesn't count the years of growth and experience, which are the case for TinyOS.

8.3 General Comments

Considering all the experiments we committed, in reference to their produced measurements and results, we can conclude that our gateway system has been successfully designed and implemented, since it performs according to the requirements we posed, from the very beginning. We achieved, in a high level, to accelerate performance and to augment functionality of embedded devices, keeping reliability and system robustness in the highest degree.

During the experiments, we also discovered with satisfaction, that the mechanisms we integrated in our gateway system, present functionality according to their specifications. In particular, caching and failure masking (in a very small number of cases, near the ending of the simulation procedure, some devices were resetting, mainly for Contiki case without the system losing a single request), were tested in heavy workload scenarios and presented a very stable operation.

We believe that our gateway, offers a high scalable and flexible solution, in cases when multi-user support is needed, in scenarios where interoperability between the Web and sensor networks is desired. We propose an efficient approach of solving this gap, between these two different infrastructures and we desire to help the scientific research, towards the direction of fully integrating embedded devices into the Web.

We must admit that, during experimenting, we discovered some bottlenecks, that may have reduced the overall functionality of our system. We present these issues in the subsequent subsection, with the aim of facing these problems in the near future, in order to further boost performance of our application environment.

8.3.1 Bottlenecks

Here we list some of the bottlenecks that appeared during our implementation efforts and affect our system.

- **Serial Listener.** Communication between Contiki's Base Station and Java, was done through a serial listener, which did not always interpret the response data that arrived from remote Contiki sensor devices properly. Exceptions were produced during parsing and messages were wrongly discarded.
- **Active Message static structure.** TinyOS demands for the programmer to declare statically the Active Message data structure that will be used in message exchange so both sides can understand the message semantics. This causes problems to our implementation, since we want at the beginning to send larger messages, when we describe services and smaller during normal operation, when we create requests and responses. It is something that TinyOS developers must consider, in the newer versions of their operating system software.
- **Transmission Power Issues.** During the experiments and mainly during Device Discovery phase, we observed that TinyOS-enabled sensors' transmissions were more powerful than Contiki sensors' transmissions and sometimes that led to cases, when Contiki messages started arriving only after TinyOS sensors had transmitted their own messages. That obviously results in wasting time and energy due to retransmissions.

Discussion and Future Work

In this chapter we summarize the important aspects and conclude this master thesis. We identify future work and we present our general thoughts, concerning the work performed.

9.1 Future Work

After evaluating our architecture, we suggest that our implementation is satisfactory according to our requirements. However, many issues remain to be solved. We present in this section a few challenging topics for future research:

Load balancing In heavy scenarios, with a big number of Web clients making use of the system, a solution that balances the load between similar devices would reduce congestion in their request queues and increase performance by distributing the requests in balance between them. This mechanism would unload the burden from some devices, which are more "frequently" used than others and it could extend their battery life-time.

Advanced Eventing A very attractive idea is the development of an efficient and powerful eventing mechanism.

Viewing the gateway as a stand-alone entity, this mechanism could include the integration of a simple scripting language on top, in Presentation Layer, giving to a Web client the possibility of creating complex rules, that would be triggered after a sequence of events being executed. These rules, could be iteratively be checked for validity from the system every time a new event was triggered from a device, and an action, also specified by the user, would be executed, for example sending an sms or an email or even sending a message to a high-level application. These rules, could be stored in an efficient data structure, such as a database.

From a large deployment of gateways point of view, eventing mechanism could involve a distributed approach, where Web clients could create rules for devices and services which are deployed in some area, probably in the whole World Wide Web and are Web-enabled by our gateway systems. Gateways would maintain subscriptions from Web clients interested in some events that devices, already represented by these gateways, could produce. These produced events, could afterwards be sent as notifications to the corresponding gateways, where the interested Web clients reside. Then, these gateways, could check their lists of rules, for triggering of some complex, distributed events.

Thread Performance One of the main design concepts of our model is the representation of every device, connected to the gateway, as a thread. These threads operate in parallel, independently from each other, but it is still responsibility of the system to maintain the duration of time each thread is invoked for execution. We could examine the possibility of assigning thread priorities, according to how "popular" a device would be and exploit this "popularity" by letting a larger amount of time, for this thread to be executed. In other words, we seek in taking advantage of thread functionality to increase system's performance by avoiding unnecessary computational processing.

Prioritized Queues Another design characteristic of our system that could be exploited for efficiency reasons, is the concept of request queues, that are used in each device's thread environment to enqueue requests that arrive in the system, concurrently or in small intervals, from a multi-client application. A mechanism could be developed that would assign priorities to each request based on their significance. For example, normal client requests could be marked with *medium* priority, requests from user administrators with high and aliveness checks would be marked by *low* priority. Then it would be responsibility of the Device Layer's implementation to create an intelligent method, that would reorder the requests in the queue, according to some efficient algorithm, which would enhance performance and would provide, in general, user satisfaction.

Synchronization Request queues, could also be used from our gateway, in coordination to the devices connected to it, in order to be synchronized with their waking times. Then the gateway would send requests to them only during their awoken hours. This could increase dramatically their battery life-time. In other words, a synchronization between request queues of every device and the device itself could be created that would save energy consumption of the devices by reducing the duration of time they stay awake but at the same time being capable of answering requests in an acceptable amount of time.

Access Control The gateway has been developed as an open, freely accessible software. This, however, does not imply that all the resources on the gateway have to be accessible to everyone automatically. HTTP already provides the building blocks to perform access control by means of HTTP authentication. A future extension could directly reuse these existing algorithms and introduce restricted areas to the gateway.

Gateway Selection A general distributed problem, not faced by our approach, is the dynamic selection of a gateway in the neighborhood of the embedded devices, according to some specifications such as the signal strength of the available gateway devices or their physical distance from the devices. In a real-life scenario where gateways are interconnected and coexist in the same area, devices should have the possibility of dynamically selecting the best gateway to connect to, answering requests exclusively from it, disconnecting from it and connecting to others during their life-time. This mechanism should be developed, in cases of mobile devices, where the transition between gateways would be inevitable.

URL for Description Data To accelerate Device Discovery phase's duration, instead of following the normal procedure of sending device and service description messages, as it was described in Section 5.1, we propose in the future to implement transmission of a single message containing a URL (provided possibly be device manufacturer), where device and resource

description information is included and from where the Web clients can get a complete view about the specific device and its capabilities.

Device Statistics Maintaining statistics about the QoS of devices connected to the gateway, such as throughput, Round Trip Time (RTT) and packet loss "probabilities" would be a powerful mechanism in our system. Web clients would need to know which devices are the most reliable, in order to select them and perform their desired tasks.

Automatic Device Programming In the implementation of our gateway, the installation procedure of our RESTful software for embedded devices, follows a static procedure. The user of the system must hard-code the necessary command in order to begin the installation operation, as it is indicated in Appendix A. Our intention is to develop an automatic procedure of installing the software on devices, following a simple GUI, through the Presentation Layer. Through this GUI, the user must be able to select device type, USB port where the device is directly connected to the system and unique identification number (*deviceID*). Then the image file of the dedicated implementation of the RESTful software, for the corresponding device type, would be uploaded to the real device.

9.2 Conclusion

In this master thesis, we proposed an application layer based on REST architectural principles, which could be used in WSN, to build flexible applications on top of resource-constrained sensor devices. We integrated this framework into a mediator device, a gateway, to enable embedded devices on the Web. We view the new generation of physical sensor devices as independent, first-class citizens of the World Wide Web.

Our RESTful approach comes to give answers to problems such as heterogeneity and uniformity and offers an efficient and standardized way of interacting with such networks. We implemented a RESTful software for sensor devices, namely for contiki and tinyOS and by means of RESTful gateways we show that multi-user applications can be created with little effort, operating with great performance and reliability.

Our final goal, is the full integration of sensor devices on the Web, something that would trigger the realization of the Web of Things. We believe that gateways will be the foundational elements, which will guide us towards that direction. We intend to embed our gateway software on devices that are already massively deployed in the Web infrastructure, such as routers and normal computers, to make the use of our system, more enticing to the end users.

At the end of the day, a large deployment of "smart" gateways, could be used to create a distributed location-aware infrastructure for mobile devices [35], where even mobility on embedded devices, would be supported at the Web.

Through our approach, by means of gateways, we wanted to illustrate the endless possibilities that can be explored on top of our concepts and provide to the reader a new way of thinking, in the area of WSN.

9.3 Acknowledgments

First of all, I would like to dedicate my work to my father, who is not any more in life and who gave me a unique motivation to finish this master thesis with a way I can not express, merely with english language. I would like also to say thanks to Vlad Trifa, who gave me the

possibility of working from distance, when I had no other choice due to my family problems and for the great support and advices he gave me during these six months. To Samuel Wieland for the excellent cooperation in integrating our works together and to my cypriot friends, who recently visited me here in Switzerland and entertained me while I was adding my last details to my work.

Software Installation on sensor Devices

For people who are not familiar in programming sensor devices, the installation of the software we developed can be a little tricky procedure. We list here the necessary steps, that someone needs to follow in order to install our software on sensor devices simply and fast, both for Contiki and TinyOS.

A.1 TinyOS Installation

At first someone must install the latest version of TinyOS. An ideal web page, for that purpose, with additional, necessary libraries can be found under:

<http://www.tinyos.net/tinyos-2.x/doc/html/install-tinyos.html>.

Base Station's software in TinyOS 2.x installations can be found under `{tinyOS root dir}/apps/BaseStation`. To install it on the sensor mote, a user must use the command:

```
make platform install.1 bs1,/dev/ttyUSBx
```

where x is the USB port the mote is plugged in and *platform* is the sensor device type being used, in our specific implementation, *tmote* was used. The number after *install* keyword, in our case *1*, identifies the address of the sensor mote and respectively the address of the gateway. It is necessary for the system to work, to use *1* as Base Station's ID, since this number is hard-coded in remote sensor motes, to be able to seek for a gateway (see Section 7.1.2 for details).

Sensor devices' assigned USB port can be found by typing the command *motelist*. This command lists all the sensor devices connected to the USB ports of the system, as well as their associated USB port number.

All remote sensor motes, which will be deployed in the gateway's neighborhood, must be equipped with our RESTful TinyOS software. In order to install it on a sensor, the command:

```
make platform install.y bs1,/dev/ttyUSBx
```

must be executed inside TinyOS software's directory. Care must be taken to use a larger integer number as unique address of the newly deployed sensor mote (replacing y with that number). As before x is the USB port, mote is temporarily plugged in (only for the installation procedure, since then it will be deployed somewhere in the neighborhood and will communicate wirelessly with the computing device).

Execution After TinyOS software is installed on the sensor mote, execution starts immediately on sensor device's platform. The only pending requirement for the system, to start supporting TinyOS-enabled sensor devices, is the setup of an environmental variable. That variable is necessary for Java, to be able to listen to Active Messages and it is called *MOTECOM*. It is set by typing the command:

```
export MOTECOM=serial@/dev/ttyUSBx:platform
```

where, as before, x is the USB port where the Base Station is plugged in and *platform* is the specific sensing platform used, in our case *tmote*.

In addition, the user must specify, during invocation of the gateway, at the end of the normal command the string:

```
-y
```

to indicate support for TinyOS devices.

A.2 Contiki Installation

At first someone must install the latest version of contiki from Contiki's official web site [1].

To equip sensor devices with our RESTful Contiki software, the commands:

```
make tmote contikiDevice.c
make TARGET=sky contikiDevice.upload
```

must be executed inside Contiki software's directory. *contikiDevice* is the filename where the code of our software resides. The first command loads from Contiki's home directory all the necessary libraries, which are declared inside our code and adds them in the local directory. The second command installs our software on all the sensor devices, which are connected to any USB port of the system.

Contiki RESTful software does not separate Base Station implementation from remote sensor device's code. Both functionalities are achieved by the same software. To enable a sensor device, in order to operate as a Base Station, the user just needs to connect the device on some USB port of the system and press the *USER* button, on sensor device's surface. Then a blue LED starts lighting, to indicate that the specific sensor device is now used as a proxy for Contiki remote sensors.

Execution As in the case with TinyOS, the gateway must know exactly in which USB port number, the Contiki Base Station operates. The reason is because it listens to the serial listener (through USB) for incoming messages. Only by knowing the exact USB port number, the gateway can listen to these messages. The setup for this issue is much easier than with TinyOS. The user must just specify while invoking the gateway, at the end of the normal command the string:

```
-c USBx
```

where *-c* indicates that support for Contiki sensor devices is provided and x is the USB port number, where the Contiki Base Station is plugged in.

Source Code

B.1 Asynchronous/Synchronous Execution Synchronization

The code shows some parts of the synchronous/asynchronous synchronization mechanism for the integration of the master thesis of Samuel Wieland.

Listing B.1: Implementation of the synchronizer between asynchronous and synchronous behavior.

```
1 package ch.ethz.inf.vs.gateway.plugin.devices.drivers.motes;
2
3 public class MoteDevice extends Device {
4
5     // locking stuff...
6     /** the next free token. DONT MODIFY DIRECTLY, USE getToken() */
7     private static Long msgToken = new Long(1);
8
9     /**
10      * @return a token for the message id.
11      */
12     public static synchronized long getToken() {
13         synchronized (msgToken) {
14             return msgToken++;
15         }
16     }
17
18     /**
19      * dispatch a response to the synchronizer.
20      * @param r the low level response.
21      */
22     public static synchronized void dispatchResponse(Response r) {
23         AsyncToSync lock = synchronizer.get(r.getRequestID());
24         if (null == lock) {
25             return;
26         }
27
28         synchronized (lock) {
29             lock.setResponse(r);
30             lock.notifyAll();

```

```
31     }
32 }
33
34 /**
35  * helper class to synchronize the async communication to contiki/
36  * tinyos.
37  * @author sawielan
38  *
39  */
40 public class AsyncToSync {
41     /** my token. */
42     private final long token;
43
44     /** the response onto my request. */
45     private Response response = null;
46
47     /**
48      * constructor.
49      */
50     public AsyncToSync() {
51         token = MoteDevice.getToken();
52     }
53
54     /**
55      * @return my token.
56      */
57     public long getToken() {
58         return token;
59     }
60
61     /**
62      * @return the response
63      */
64     public Response getResponse() {
65         return response;
66     }
67
68     /**
69      * @param response the response to set
70      */
71     public void setResponse(Response response) {
72         this.response = response;
73     }
74 };
75
76 /** a hash map containing the synchronizer objects. */
77 private static Map<Long, AsyncToSync> synchronizer =
78     new ConcurrentHashMap<Long, AsyncToSync> ();
79
80 // \\ end of synchronizing
81
```

```
82     ... CODE THAT WAS OMITTED ...
83
84     /* Handles Responses from Smart Device by forwarding them to the
85     appropriate Internet Client who made the Request */
86     public void handleResponse(Response r){
87         System.out.println("Handling normal Response for service:"+r.
88             getServiceName());
89         dispatchResponse(r);
90     }
91
92     /* adds Request r in Request Message Queue */
93     public void addRequest(Request r){
94         this.msgQueue.addRequestMessage(r);
95     }
96
97     @Override
98     public String handle(org.restlet.data.Response response,
99         org.restlet.data.Request request) {
100
101         ... CODE THAT WAS OMITTED ...
102
103         Request re = new Request(deviceID,resourceName,method,params,
104             values, false,0);
105         Response r = waitSynchronous(re);
106         if (null == r) {
107             response.setEntity(new StringRepresentation(Constants.NACK));
108         } else {
109             log.debug(r.getResult());
110             response.setEntity(
111                 new StringRepresentation(r.getResult().toString()));
112         }
113         return null;
114     }
115
116     public Response waitSynchronous(Request request) {
117         // handle a request with lock wait...
118         AsyncToSync lock = new AsyncToSync();
119         synchronizer.put(lock.getToken(), lock);
120         try {
121             log.debug("wait on lock.");
122
123             request.setRequestID(lock.getToken());
124             addRequest(request);
125
126             synchronized (lock) {
127                 lock.wait();
128             }
129             log.debug("leaving lock");
130         } catch (Exception e) {
131             e.printStackTrace();
132             synchronizer.remove(lock);
133         }
134     }
135 }
```

```
131     }
132     log.debug("left lock");
133     synchronizer.remove(lock.getToken());
134     return lock.getResponse();
135 }
136
137 ... CODE THAT WAS OMITTED ...
138 }
```

Bibliography

- [1] Contiki: the operating system for embedded smart objects. Online at "<http://www.sics.se/contiki/>".
- [2] Tinyos: An open-source operating system designed for wireless embedded sensor networks. Online at "www.tinyos.net/".
- [3] M. Balazinska, A. Deshpande, M.J. Franklin, P.B. Gibbons, J. Gray, S. Nath, M. Hansen, M. Liebhold, A. Szalay, and V. Tao. Data management in the worldwide sensor web. *Pervasive Computing, IEEE*, 6(2):30–40, 2007.
- [4] Gaetano Borriello and Roy Want. Embedded computation meets the World Wide Web. *Commun. ACM*, 43(5):59–66, 2000.
- [5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Online at "<http://www.w3.org/TR/2001/NOTE-wsdl-20010315>", March 2001.
- [6] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: a REST-based protocol for pervasive systems. In *Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 340–348, Oct. 2004.
- [7] Adam Dunkels. Rime: A lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.
- [8] Adam Dunkels, Björn Grnvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.
- [9] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.
- [10] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O’Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. Making sensor networks ipv6 ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session*, Raleigh, North Carolina, USA, November 2008. Best poster award.

- [11] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.
- [12] P.B. Gibbons, B. Karp, Y. Ke, S. Nath, and Srinivasan Seshan. Irisnet: an architecture for a worldwide sensor web. *Pervasive Computing, IEEE*, 2(4):22–33, 2003.
- [13] Network Working Group. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Online at "<http://tools.ietf.org/html/rfc2045>", 1996.
- [14] Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. Online at "<http://tools.ietf.org/html/rfc2616>", 1999.
- [15] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Online at "<http://www.w3.org/TR/2007/REC-soap12-part1-20070427/>", April 2007.
- [16] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. Towards physical mashups in the web of things. In *Submitted to INSS 2009.*, 2009.
- [17] Usman Haque. Pachube. Online at <http://www.pachube.com>.
- [18] Open Geospatial Consortium Inc. Ogc sensor web enablement: Overview and high level architecture. White paper OGC 07-165, 2007.
- [19] Aman Kansal, Suman Nath, Jie Liu, and Feng Zhao. SenseWeb: an infrastructure for shared sensing. *IEEE Multimedia*, 14(4):8–13, 2007.
- [20] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, places, things: web presence for the real world. *Mob. Netw. Appl.*, 7(5):365–376, 2002.
- [21] Peter Ljungstrand, Johan Redström, and Lars Erik Holmquist. Webstickers: using physical tokens to access, manage and share bookmarks to the web. In *DARE '00: Proceedings of DARE 2000 on Designing augmented reality environments*, pages 23–31, New York, NY, USA, 2000. ACM.
- [22] Thomas Luckenbach, Peter Gober, Stefan Arbanowski, Andreas Kotsopoulos, and Kyle Kim. TinyREST - a protocol for integrating sensor networks into the internet. in *Proc. of REALWSN*, 2005.
- [23] SUN Microsystems. Web Application Description Language. Online at "<https://wad1.dev.java.net/wad120090202.pdf>", 2009.
- [24] Cesare Pautasso and Erik Wilde. Why is the web loosely coupled? a multi-faceted metric for service design. In *Proc. of the 18th International World Wide Web Conference (WWW2009)*, Madrid, Spain, April 2009.
- [25] Christian Prehofer, Jilles van Gurp, and Cristiano di Flora. Towards the web as a platform for ubiquitous applications in smart spaces. In *Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI), at Ubicomp 2007*, 2007.

- [26] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 253–266, New York, NY, USA, 2008. ACM.
- [27] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.
- [28] Leonard Richardson and Ruby Sam. *RESTful Web Services*. O'Reilly Media, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition edition, May 2007.
- [29] Want Roy, Fishkin Kenneth P., Gujar Anuj, and Harrison Beverly L. Bridging physical and virtual worlds with electronic tags. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 370–377, New York, NY, USA, 1999. ACM.
- [30] P. Schramm, E. Naroska, P. Resch, J. Platte, H. Linde, G. Stromberg, and T. Sturm. A service gateway for networked sensor systems. *Pervasive Computing, IEEE*, 3(1):66–74, Jan.-March 2004.
- [31] Vlad Stirbu. Towards a restful plug and play experience in the web of things. *Semantic Computing, 2008 IEEE International Conference on*, pages 512–517, Aug. 2008.
- [32] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. Hugly, and E. Pouyoul. Project JXTA-C: Enabling a Web of Things. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, pages 282–290, 2003.
- [33] J. van Gurp, C. Prehofer, and C. di Flora. Experiences with realizing smart space web service applications. *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1171–1175, Jan. 2008.
- [34] Juan Ignacio Vazquez, Diego López de Ipiña, and Iñigo Sedano. SoaM: A Web-powered Architecture for Designing and Deploying Pervasive Semantic Devices. *IJWIS - International Journal of Web Information Systems*, 2(3-4), 2006.
- [35] Philipp Bolliger Vlad Trifa¹, Dominique Guinard and Samuel Wieland. Towards the web of things: Design and implementation of a distributed location-aware infrastructure for mobile devices. 2009.
- [36] Samuel Wieland. Design and implementation of a gateway for web-based interaction and management of embedded devices. Master's thesis, Departement of Computer Science, ETH Zurich, 2009.
- [37] Erik Wilde. Putting things to REST. Technical Report UCB iSchool Report 2007-015, School of Information, UC Berkeley, November 2007.