IT 09 049

Examensarbete 45 hp
October 2009

RESTful Wireless Sensor Networks

Dogan Yazar

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

RESTful Wireless Sensor Networks

Dogan Yazar

Sensor networks have diverse structures and generally employ proprietary protocols to gather useful information about the physical world. This diversity generates problems to interact with these sensors since custom APIs are needed which are tedious, error prone and have steep learning curve. In this thesis, I present RESThing, a lightweight REST framework for wireless sensor networks to ease the process of interacting with these sensors by making them accessible over the Web. I evaluate the system and show that it is feasible to support widely used and standard Web protocols in wireless sensor networks. Being able to integrate these tiny devices seamlessly into the global information medium, we can achieve the Web of Things.

Handledare: Adam Dunkels
Ämnesgranskare: Per Gunningberg
Examinator: Anders Jansson
IT 09 049
Tryckt av: Reprocentralen ITC

Preface

I am very thankful to my supervisor Adam Dunkels for his invaluable support. Not only he gave great feedback in every part of this work but also encouraged me to perform better. Moreover, I would like to express my special thanks to Fredrik Österlind, Nicolas Tsiftes, Niclas Finne, Joakim Eriksson, Thiemo Voigt, Shahid Raza, Zhitao He and Luca Mottola for their assistance in various parts of this thesis. Thanks also to Per Gunningberg, my reviewer at Uppsala University, for reviweing my work and giving very useful feedback on the report. Even though I am the only author of this report, what I express as 'I' goes beyond a single person in most parts of this work thanks to the people who did not withold their kind contribution. SICS is a great research lab and I am very glad to be a part of this environment during my master thesis work.

Contents

List of Figures

Chapter 1

Introduction

Wireless sensor networks (WSNs) are used for various areas such as environmental monitoring [32], building automation [49], habitat tracking [30, 35] and health-care [9] applications. These networks typically consist of many embedded devices referred as sensor nodes and these devices are kept resource constrained to minimize the overall cost. Typically, each of these nodes consists of a small microprocessor, a transceiver, a number of sensors and is usually battery powered. Having minimal resources, many challenges appear and special care should have taken to use the resources as efficiently as possible and handle these challenges.

Various different types of applications are the consumers of sensor data so we need easy and feasible mechanisms to access WSNs. And as sensor networks move to IP [5], one idea is to integrate sensors into the Web and making them accessible over the Internet.

1.1 Motivation and Problem Statement

Sensors are devices that let us monitor and react to the physical world. For example, as its name implies, a temperature sensor can give us the temperature of the environment it is in. But how do we collect this useful information? We need sensor nodes to communicate and collaborate with each other which is realized via wireless sensor networks. However, currently most WSNs are based on specialized software and hardware platforms and due to performance reasons, custom protocols and APIs are used which are efficient but also tedious, error prone and have steep learning curve. These non-standard mechanisms are limiting accessibility and interoperability of the sensors which means they are also blocking the emergence of new type of applications that can improve our lives in every stage of daily life.

We need uniform and easy mechanisms to access these devices for the sake of interoperability and indeed integrating these devices into the Internet would not only provide it but also yield many other opportunities the Internet provides. The idea of connecting these tiny things to the Internet is mentioned as the Internet of Things and there exists two main ways to establish this purpose. One way is to employ gateways, which work as converters between protocols of the Internet and custom protocols used in the wireless sensor

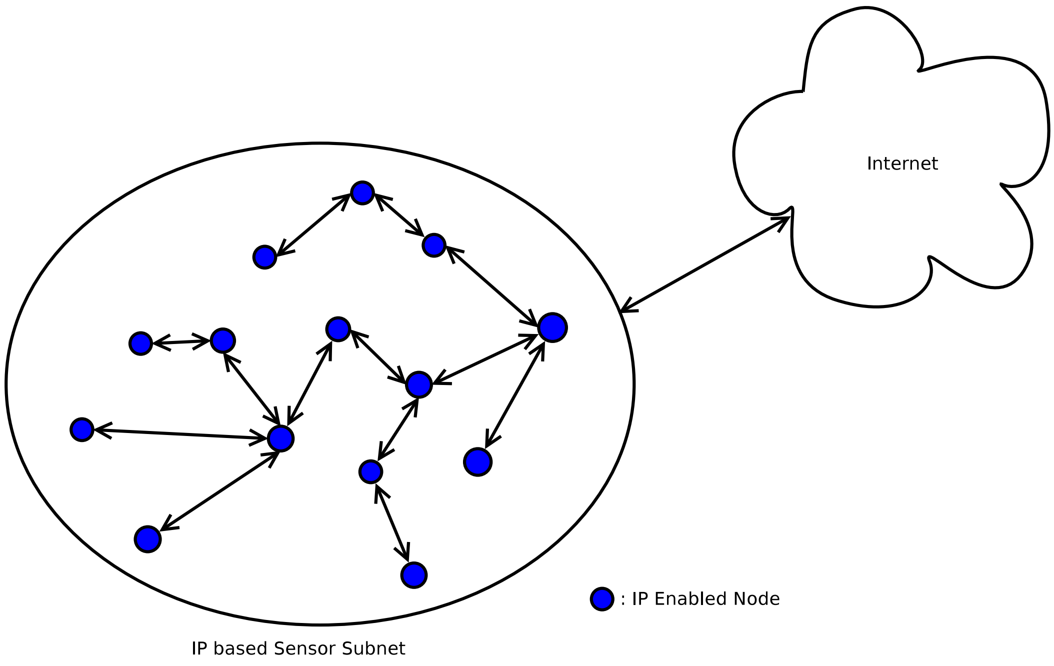


Figure 1.1: Connecting wireless sensor networks to the Internet directly via IP enabled sensors.

networks. The other way is having end-to-end communication using IP enabled things, as shown in Figure 1.1. Especially with IPv6, we have enough address space to identify all of these things. The work in this thesis is about the latter approach; eliminating the need of an Application Layer Gateway whose job is to convert between standard formats to custom ones. For that matter, I focus on using proven and widely-used standards (HTTP, TCP/IP, etc) directly on sensor nodes hence not needing protocol conversion. Clients will be able to interact with the sensor nodes just as they do with traditional web servers out there in the Internet. They may not even be aware of the fact that they are accessing a very resource-constrained device.

First step of this task, namely network layer interoperability is already achieved using IP protocol in sensor nodes [18, 22]. Previously, TCP and IP were considered unsuitable for wireless sensor networks believing that TCP and IP are very heavyweight, but they are proven to work well with sensor domain [18, 22]. The next step is to provide application level interoperability. For this concern, we need to attack the problem as the same way IP made viable for sensor nodes; we need to make higher level protocols, i.e. HTTP viable. The importance of HTTP is that it is the most important requirement on the way of the Web of Things.

This thesis offers a solution to integrate sensors into the Web using the principles of REST, the architectural model of the Web. I also analyze the costs of using RESTful principles over WSNs and identify the ways to minimize the costs by certain design choices and optimizations to prove that HTTP is viable in sensor networks. I use Tmote Sky motes [42] as the platform and have achieved completion times in terms of hundred milliseconds for RESTful Web services to collect sensor data.

The motivation for this work is the idea to access sensors using standard Web tools such as browsers, feed readers, etc. Tiny devices have the capability to be embedded into the physical world, therefore, if we can integrate these devices into the Web, a huge amount of data about physical world will be available. So, combining it with the serendipitous nature of the Web, new types of applications, beyond the ones that we are already familiar, are very possible to appear.

Increasing popularity of Web Mashup applications [56] shows us that they can be the driving force of generating applications using real world data. A Web Mashup is a web application that gathers data from different external sources (usually APIs) to offer new services. Mashup applications are very popular lately since new applications can be generated easily by using existing applications and since already existing resources are used, fast development is possible. By putting physical objects in the picture, mashups can use real world data in real time hence generating a new bunch of useful applications that may not be predictable by now. Example mashup applications involving physical object has already started to appear such as the ones appearing in the work of Guinard and Trifa [17].

In this work, sensor, node, and mote words are often used to describe the tiny embedded devices related to the study, however the concepts are equally applicable to actuators as well, i.e. it is possible to access actuators over the Web and control the behaviour of real world objects. Moreover, actuators may coexist with monitoring applications, or even better by working together, i.e. they can act upon the conditions in the environment such as HVAC (heating, ventilating, and air conditioning) systems.

1.2 Internet Of Things and Web Of Things

Even though sometimes Web and Internet terms are used as if they mean the same thing, in reality their meanings are quite different. Internet is the network of computer networks realized by IP protocol, whereas Web is a set of resources that are connected to each other by hyperlinks. These resources are uniquely addressed by URIs and accessed via HTTP protocol over the Internet. There exist Internet applications which are not part of the Web such as emails.

Internet of Things means connecting the computers on the Internet and physical objects (sensors and actuators) to have the opportunity of monitoring and reacting the physical world events. But, I believe that is not enough, on top of it we should have the Web of Things in which physical things are accessed using standard Web mechanisms. For example; sensors should have URIs and their readings should be accessible over these URIs. Erik Wilde has written a technical report [55] which describes the Web of Things and its vision very well.

1.3 RESTful Sensor Network Applications

Showing that RESTful Web services are viable for WSNs does not mean anything alone if we do not have reasonable use cases. For example; how can we collect sensor readings in a RESTful way? I provide an overview of the Web communication model and describe how it differs from typical sensor application problems.

The Web architecture depends on a client-server model so clients have to actively pull the content instead of getting it pushed to them. However, for a typical physical world monitoring application, we need to have some form of asynchronism for several reasons:

1. To access the updated readings instantly, i.e. the sensors asynchronously send updates in real-time rather than as responses to synchronous service calls.

2. To be able to support time-consuming operations which do not return results immediately, such as tasks running in background continuously and sending results when they are available.

3. To utilize bandwidth usage, i.e. nodes communicate only when new data exists rather than being polled periodically.

The Web has a synchronous communication model only. A client opens a connection to a server, makes a request, the server responds to the request and the connection is finalized (see Figure 1.2). This is a very simplified view, there exist some other points in this discussion such as persistent connections and Ajax [53]. Ajax is a web programming technique for creating more interactive and faster web applications with a mechanism of data trade without doing a full page refresh. Although, both Ajax and persistent connections have advantages, actually they still use the same synchronous interaction model of Web. Also, working on a domain of devices having extremely small resources, persistent connections do not seem to be a good selection since the devices do not have enough memory to handle many persistent connections. It can be feasible if the sensor will only be connected with a single application though, but this would be a serious limitation for most applications. There are also other protocols than HTTP which has better real time behavior such as Extensible Messaging and Presence Protocol (XMPP) [48], however my concern here is HTTP since it is the main transport protocol of the Web.

In Section 7.1.1 I offer a RESTful approach to solve typical sensor application problems.

1.4 Method

This study is experimental, it includes both implementation as well as measurements to evaluate the system considering performance and energy-efficiency. I have implemented lightweight HTTP Server, REST Engine and Logger modules on Contiki as well as modified and ported a simple XML parser to Contiki. I have performed many optimizations on the code and carried out various evaluations. The measurements are done for power consumption and completion time in order to quantify the performance of the system. Moreover,

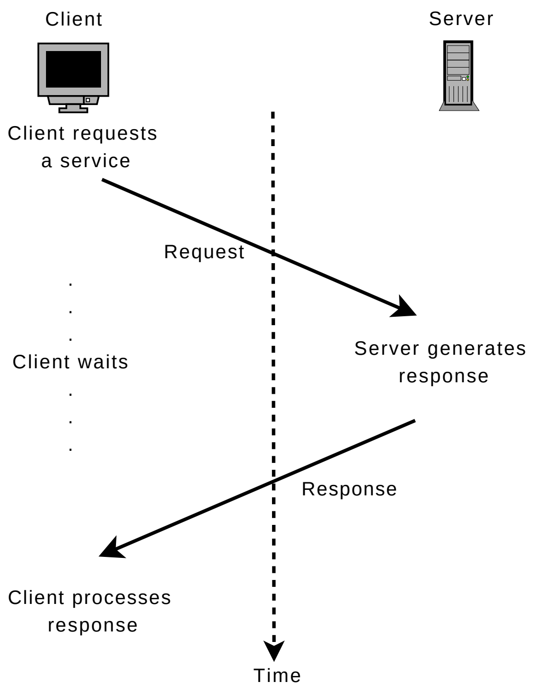

Client
Server
Client requests a service
Request
Client waits
Server generates response
Response
Client processes response
Time

Figure 1.2: Synchronous communication model of the Web

results of different methods and optimizations are provided to compare them and achieve the best result.

1.5 Limitations

This thesis focuses on RESTful Web services and their underlying protocols and standards (HTTP, TCP, XML) over WSNs. Although security is very important, due to its complexity that it would bring on the work level as well as on the code size level, security related issues are not discussed or implemented in this thesis. Moreover, network layer and lower layers are not the main concern of this work. That is why the routing between the sensor nodes is out of scope in this thesis. The only exception is the work I perform on MAC layer, i.e. tuning X-MAC [13] for better TCP communication.

1.6 Alternative Approaches

This thesis is about integrating WSNs to the World Wide Web. For that purpose, I have decided to build and evaluate a framework following Representational State Transfer (REST) principles since REST is the software architecture style underlying the World Wide Web. There exist many other technologies to build Web applications, most notably SOAP-based Web services. There appear two main problems with SOAP-based Web services; firstly, they use the Web as a transport medium instead of integrating to it, and secondly, they are heavier and more complex than RESTful Web services in terms of memory, bandwidth and computation requirements especially due to SOAP layer and constantly growing WS-* stack. Related to that matter, we can also talk about traditional distributed middlewares such as DCOM, CORBA which have the same disadvantages of SOAP-based Web services in our context. Moreover, they have an extra disadvantage; they do not work through firewalls or proxy web servers since HTTP is not employed.

Another approach would be to employ translation gateways, which work as converters between protocols of the Internet and custom protocols used in the wireless sensor networks. Main advantage of this method is efficiency; using optimal protocols for WSNs would definitely be more efficient. However, this approach has drawbacks; a translation gateway is always required in between sensor network and the Internet, and interoperability between sensors would be very limited due to the custom protocol, namely it would be very hard to combine different platforms in the same network. Also, past experience shows us that protocol gateways are very complex to design, manage and deploy, besides other problems are also possible due to conversion such as inconsistent routing, QoS, transport and network recovery techniques [22]. So the question here is extra overhead for having maximum interoperability acceptable or not? I prove that it is reasonable and promising in Section 6.

1.7 Scientific Contributions

This thesis contains two scientific contributions. Firstly, I show the feasibility of using RESTful Web services on IP-based multi-hop low-power sensor networks by employing a number of optimizations. These optimizations include programming techniques to achieve small code and data memory usage, MAC layer optimizations to improve throughput while conserving power as well as HTTP mechanisms to save bandwidth. Thanks to these optimizations, it is possible to achieve completion times in terms of hundred milliseconds for RESTful Web services with a power consumption of just several milliwatts. Secondly, I evaluate the performance of the work, prove that it is reasonable and also compare it with SOAP-based Web services. My evaluations show that RESTful approach outperforms SOAP based technique, namely for a typical actuator example of controlling LEDs, RESTful approach is 58.9% more energy efficient and responds 4.7 times faster than SOAP-based approach. Furthermore, I perform evaluations for RESTful Web services over IPv6 and according to my evaluations, in average IPv6 has 42.4% more overhead than IPv4 counterparts in terms of completion time.

1.8 Thesis Structure

This thesis is structured as follows. Chapter 2 describes the background of the work and explains concepts related to this thesis, namely an overview of Web services and the underlying technologies/standards are given. Related work also exists in this chapter. After that, I present design of the system in Chapter 3 and analyze how to make HTTP viable in Chapter 4. In Chapter 5 implementation of the overall architecture is described. In Chapter 6, evaluations are given and interpreted. I conclude in the last chapter. Finally, in Appendix A I present the code API.

Chapter 2

Background

Wireless sensor networks consist of sensor nodes that monitor physical conditions. IEEE 802.15.4 is a standard used in these networks addressing physical and MAC layers. There appears power saving MAC protocols such as Low Power Probing (LPP) [45] and X-MAC [13] aiming to conserve energy. The sensor nodes in these networks are sometimes referred as motes as well and Tmote Sky is one of the mote platforms commonly used. Many operating systems are present that are targeted to run on sensor nodes such as TinyOS [29] and Contiki [19].

The Web is a distributed system of interlinked documents running over the Internet. HTTP, URI and XML are the basic Web technologies. REST is the underlying architecture model of the Web. Web services are used to develop interoperable distributed applications usually using Web-related standards. Web services are generally categorized in two classes: SOAP-based Web services and RESTful Web services. Shortly, SOAP-based Web services employ Simple Object Access Protocol (SOAP) standard, however RESTful Web services employ REST principles so they are resource oriented and lighter since they work on top of HTTP directly.

2.1 Wireless Sensor Networks

A wireless sensor network (WSN) is a type of wireless network consisting of large number of small embedded devices which are referred as sensor nodes. Having equipped with sensors (and/or actuators) and wireless communications devices (i.e. radio transceiver), these nodes collaborate to sense their physical and environmental conditions such as motion, temperature, smoke, light etc. This type of networks have many application areas, some of which can be counted as military applications, home automation, environment monitoring, etc.

2.1.1 Mote

A mote, which is also known as a sensor node, is a wireless sensor device that represents a node in a WSN. The main components of a mote are microcontroller, radio transceiver,

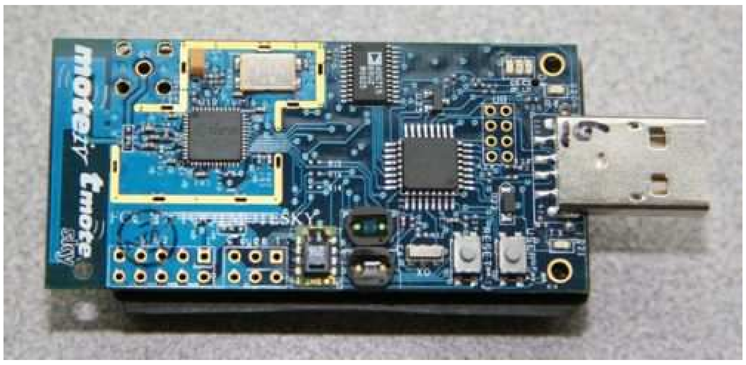


Figure 2.1: Tmote Sky Mote

external memory, power source and a number of sensors. Motes can sense and monitor physical assets of the environment using their sensors, process data using the microcontrollers and communicate with their neighbours in range using their transceivers. Batteries are usually the main power supply of the motes and radio transceiver is usually the most power consuming component of a typical sensor node. Regarding external memory, Flash memories are mainly used due to their low cost per unit capacity. There are many different motes are used in WSNs; Sun SPOT, MicaZ, Tmote Sky just to name a few.

2.1.2 Tmote Sky

For this thesis, the mote platform is chosen as Tmote Sky [42] which is shown in Figure 2.1. It is an ultra low power IEEE 802.15.4 compliant wireless mote having the following key attributes:

- 8MHz Texas Instruments MSP430 microcontroller (10k RAM, 48k Internal Flash, 1MB External Flash)

- Integrated Humidity, Temperature, and Light sensors

- 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver having maximum raw data rate of 250kbps

2.1.3 IEEE 802.15.4

IEEE 802.15.4 is a standard that specifies the physical layer and media access control for low-power personal area networks (LoWPANs). Its main aim is to support long battery life by offering limited capabilities; small frame sizes (the maximum frame length is 127 bytes), low bandwidth and transmit power.

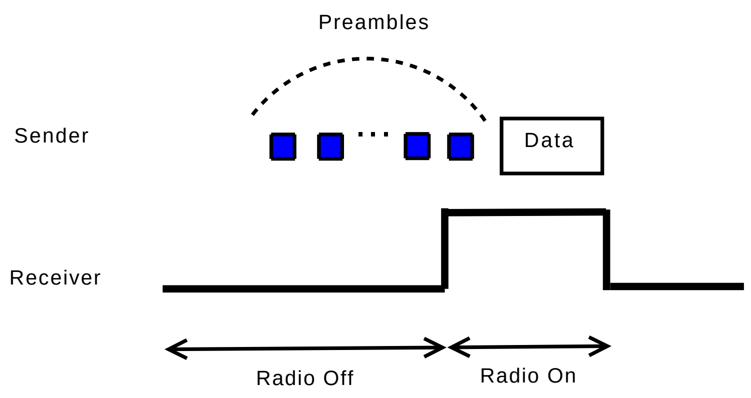


Figure 2.2: X-MAC: Sender transmits preambles until the receiver is awake and then transmits actual data.

6LoWPAN

6LoWPAN [27] is an adaptation layer between MAC and network layer that is used to provide IPv6 support over IEEE 802.15.4 radio. Its aim is to enable low power operation by compressing headers and hence saving bandwidth and power.

2.2 X-MAC

X-MAC [13] is a low power MAC protocol that uses a sequence of short preambles to wake up the receivers. Radio transceiver is the most energy consuming component of a typical sensor node and idle-listening constitutes the main part of total energy usage. X-MAC addresses this problem; motes save energy by switching off the radio most of the time and hence reducing idle radio listening. Nodes wake up for a short time in regular periods to listen for preambles. When a node wakes up and receives a preamble addressed to itself, it replies with an acknowledgement showing that it is awake. Upon reception of the acknowledgement from the receiver, sender transmits the whole packet.

2.3 Contiki

Contiki [19] is a lightweight open-source operating system, specifically designed for memory-efficient networked embedded systems and wireless sensor networks. It provides both full IP networking and low-power radio communication mechanisms via three communication stacks: Rime [20], a lightweight layered communication stack that provides basic communication primitives on top of which more complex protocols are built, μIP [18] is a minimal fully RFC compliant TCP/IPv4 stack, and μIPv6 [23], is the world's smallest fully RFC compliant TCP/IPv6 stack.

Currently, Contiki has support for 3 different MAC layer protocols, namely NULLMAC, X-MAC [13] and Low Power Probing (LPP) [45]. NULLMAC is the simplest MAC protocol in which transceiver is always on, whereas X-MAC and LPP are the power conserving protocols.

The X-MAC implementation that Contiki offers has two important parameters which are related to this thesis:

- On Time: How long the sensor keeps the radio switched on while listening for strobes

- Off Time: The duration between two listening times in which the radio is off

Contiki has an on-line power profiling mechanism [21] which is used to evaluate power consumption in this thesis. It estimates the energy consumption by measuring the duration each component is in various modes such as low-power mode, transmitting.

2.4 SLIP

The Serial Line Internet Protocol (SLIP) [47] is an encapsulation of the Internet Protocol designed to operate through a serial connection. SLIP is a very simple protocol that frames IP datagrams to send them over serial connections. Although it is mostly obsolete now, thanks to its small overhead, it is still used for connecting constrained embedded systems.

2.5 TCP

Transmission Control Protocol (TCP), described in RFC 793 [43], is a connection-oriented protocol; a connection is established and kept open during the data exchange between each endpoint. Endpoints are defined by IP address and TCP port number pair.

TCP is one of the main protocols of the Internet Protocol Suite (TCP/IP). It lies in the transport layer and it is reliable; it guarantees data delivery and that packets will be delivered in the same order as they were sent. TCP also supports sophisticated congestion and flow control mechanisms via adaptive windowing techniques.

2.6 Web Architecture

The World Wide Web (WWW), abbreviated commonly as the Web, is very widely used, with some other technologies such as e-mail, P2P, usenet, and IRC, it made the Internet so much popular. Currently, most of the users of the Internet are human beings, but this picture is changing with Web 2.0 since programmable web notion is getting more popular each day. In another words, some of the clients of some web applications are programmed machines.

The aim of rest of this section is to describe REST and basic Web technologies relevant to this thesis; HTTP, URI, and XML.

```
<Sensors>
  <Sensor>
    <Name>temperature</Name>
    <Value>25.1</Value>
  </Sensor>
</Sensors>
```

Figure 2.3: A simple XML document

2.6.1 URI

A Uniform Resource Identifier (URI) is a string that identifies a resource on the Internet. It is described in RFC 3986 [10]. URI is one of the main components of the Web that is used to name and address all the piece of data that clients want to access. Every URI labels exactly one resource and every resource present on the Web has at least one URI.

2.6.2 XML

Extensible Markup Language (XML) [12] is an open standard recommended by the World Wide Web Consortium (W3C). XML is a markup language designed to transport and store data in a plain text form. Its simple and flexible structure made it a very important exchange format of data on the Web. An example XML document is given in Figure 2.3

In the context of the Web services, an XML parser is needed to process XML documents and extract the information needed. Currently there exists two main XML parser types; Simple API for XML (SAX), Document Object Model (DOM). SAX parser is an event based parser; it parses the XML data into a series of events such as tag opened, closed etc. Parser invokes the callback functions with corresponding events while going over the document, in another words, it pushes events to the user. It is fast and efficient but generally referred as hard to use since user has no control over the parsing after callbacks are set. DOM parser generates a tree data structure from the XML data which can later be travelled to extract needed data. Using DOM parser is generally considered to be easier than using SAX parsers but due to the overhead of extra data structure representing the document created, it is much more greedy in memory usage than SAX parser. Therefore, if the available memory is very small or the XML document is very big, DOM parser is not a good choice.

Additional to DOM and SAX APIs, there also exist new parsing approaches such as Pull parsers. Pull parser resembles to SAX parser since it converts the XML document into a sequence of events, however as its name implies, users pull data from the parser (instead of data getting pushed to the user) which makes it more flexible than SAX parser. Therefore, Pull parsers are mentioned as having the efficiency advantage of SAX parsers whereas being easier to use.

```
GET / HTTP/1.1
Host: www.sics.se
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.10)
 Gecko/2009042523 Ubuntu/8.10 (intrepid) Firefox/3.0.10
Accept: text/html,application/xhtml+xml,application/xml
;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Figure 2.4: An example HTTP request

2.6.3 HTTP

HTTP (Hypertext Transfer Protocol) is an application layer protocol used by the World Wide Web to access resources in a stateless and loosely coupled way. HTTP is the transfer protocol that makes the Web work and it also helps the Web to scale via techniques such as stateless communication, caching, persistent connections. The simple, mature and ubiquitous nature of HTTP made it very popular for many types of distributed applications. HTTP/1.1, described in RFC 2616 [25], is the version of HTTP that is commonly used now.

HTTP works in a client/server fashion and uses URIs to access resources. In most cases, HTTP uses TCP as the underlying transport protocol even though it is not mandatory to do so. A typical and simple interaction would be as follows; client establishes a TCP connection to the server (on port 80 as default), then the client sends request message, server processes the request and returns a response. The type of the responses may vary depending on the client requests and server capabilities, for example; HTML files or images may be returned for web browsers. An example HTTP request, the request done by my browser when I visit www.sics.se, is provided in Figure 2.4. Cookie part is not presented for the sake of simplicity.

Client requests consist of the request line (in the example: ”GET / HTTP/1.1”), request headers and an optional entity body).

And the response of the server is given in Figure 2.5. Entity body which includes HTML document of the web page is not written but instead just presented with [HTML Data].

Server responses consist of a line for the status (”HTTP/1.1 200 OK” states that the operation is successful), response headers and entity body (usually this part has the representation of the resource).

2.6.4 REST

Representational State Transfer (REST) is an architecture style defined by Roy Fielding in his PhD thesis [26]. It aims to design distributed networked applications using HTTP as application layer protocol and it is actually the architecture model of the Web.

```
HTTP/1.1 200 OK
Date: Fri, 12 Jun 2009 12:45:31 GMT
Server: Apache/2.2.6 (Unix)
X-Powered-By: PHP/5.2.4
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Last-Modified: Fri, 12 Jun 2009 12:45:31 GMT
Cache-Control: store, no-cache, must-revalidate
Cache-Control: post-check=0, pre-check=0
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
[HTML Data]
```

Figure 2.5: Response of the server to the example request

The main abstraction of REST is the resources. Every resource should have URI and using these URIs it is possible to link resources hence leading hypermedia. It is possible to have different representations for the same resource which is a very powerful concept, i.e. a server can serve HTML content for human consumption and XML or JSON for machines.

REST uses standard methods HTTP defines; i.e. GET is used as a safe and idempotent operation to access a resource, PUT is an idempotent operation that can be used to create or update a resource with a known URI, DELETE is idempotent as well and used to remove a resource and lastly POST is used for anything else. Although there exist other HTTP methods such as OPTIONS and HEAD, the four of the methods described are the most popular ones. Moreover, a new HTTP method is about get into standard soon; PATCH. It allows clients to do a partial update on a resource hence enabling bandwidth save. And as response statuses, HTTP has a well defined and rich set of response codes such as this resource created or resource temporarily unavailable, etc.

Working over HTTP, REST has stateless communication which means servers do not keep application states of the clients, instead clients should send all necessary state information in requests. The only state in the servers are the resources. Stateless nature is one of the main reasons under the scalability of Web.

However, REST is not the silver bullet for everything. It is definitely not the cure for all type of applications; other architecture styles or specific technologies may perform much more better depending on the requirements of the problem. So the aim of this thesis is definitely not to claim that all types of wireless sensor network problems can be solved by employing REST principles but to show that it is possible to integrate physical objects into the Web using REST principles.

2.7 JSON

JavaScript Object Notation (JSON) [15] is a lightweight and language independent text format to interchange data. The idea is to serialize data structures (numbers, arrays, etc)

```
{"Sensors": {
  "item": [
    {"name": "Temperature", "value": 26.1},
    {"name": "Light", "value": 87}
  ]
}}
```

Figure 2.6: A simple JSON document

as JSON formatted strings. JSON offers a better solution than XML for at least Javascript environments since instead of parsing it as XML, it is directly fitted into the proper data structure. That advantage makes JSON an important player in Web 2.0 applications. There exists a JSON parser available almost in every language. An example JSON document is provided in Figure 2.6

2.8 Remote Procedure Call

A remote procedure call can simply be described as a mechanism in which applications are able to make calls on remote machines transparently, i.e they appear as local procedure calls to the users. The complexities are handled by the RPC libraries, such as converting the calls to a TCP connection between client proxy and server stubs and marshalling/demarshalling the parameters and return values. There exist many RPC systems currently used still such as CORBA and Java RMI.

2.9 Web Services

According to W3C: ”A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.” [11]

2.9.1 SOAP-Based Web Services

In this thesis, Web services that employ WS-* stack are called SOAP-Based Web services. Most people think of them when Web services are mentioned currently. Main two components of these Web services are SOAP [37] and Web Services Description Language (WSDL) [14]. SOAP actually presents an envelope only but regarding SOAP-Based Web services, it is used as a wrapper for RPC calls and WSDL is an XML language for defining interfaces. SOAP is designed as protocol independent, but in practice, it is often implemented on top of HTTP. This type of Web services are also called Big Web Services or WS-* Web services in the literature.

2.9.2 RESTful Web Services

In simplest terms, RESTful Web services mean to apply REST design principles to develop Web services. REST is the underlying architecture style of the Web, so applying REST principles means direct integration to the Web. Rather than focusing on functions, RESTful Web services use Web resources as the main abstraction. RESTful Web services book by Richardson and Ruby is a very good source for the details of the subject [46].

RESTful Web services have many advantages that the Web offers. For example:

– Standard and very well known data types are used to represent data, also different types for representing same resource is possible depending on the client needs. For example; the result of some statistical data can be provided in HTML for human beings whereas in excel for some programmed client to make calculations.

– Uniform interface is provided since standard HTTP methods are used

– True language independence

– Since HTTP is employed under, tunneling over firewalls is no problem

– Caching is possible to increase the performance

– HTTP is used as an application layer so all the features of HTTP standard are inherent in RESTful Web services. Some of these features are; encryption, authentication and caching.

These advantages provided many popular Web 2.0 services to employ RESTful Web services such as Yahoo, Amazon, Flickr.

2.9.3 RESTful Web Services vs SOAP-Based Web Services

There exist many discussions about the comparisons of these two notions such as [41]. Certainly, both of the types have their advantages and disadvantages that makes them more suitable over each other for different specific cases. However, the aim of this thesis is to make sensors a part of Web, that is why RESTful Web services are a better option. Their lighter nature is an extra advantage for our limited environments.

2.10 Related Work

There have been several work about building RESTful applications and frameworks on WSNs to date. To my best knowledge, my work is the first one to have IP support on the sensor nodes; the other work employ gateways on the border of sensor networks especially for IP-to-Custom Protocol mapping. I also provide evaluations regarding the real system performance. These two are the main points that my work adds to the previous work.

TinyREST [34], is developed as part of a Home Services Framework. Its goal is to generate a specific REST based approach for the framework rather than providing a generic framework that this thesis aim for. Other than IP support, the work also does not include multihop routing and reliability within WSN, both of which are supported in my work, especially reliability is inherently supported thanks to my approach of using standard TCP/IP. A gateway connected to a base station is used to map the set of requests to TinyOS [29] messages and vice versa, which also performs some other tasks such as validity checks. More recent researches about same subject focuses on the Web of Things.

Dickerson et al. [16] have vision of World Wide Sensor Web, in which sensor data streams are accessed over the Web such as Web feeds but in a more suitable way for sensors, i.e with more capabilities (server-side filtering, streaming support and real-time updates) and in a more efficient way.

Guinard and Trifa have work emphasizing on application possibilities the Web of Things offers. In [17], they present real mashup applications using sensor data and existing tools, hence showing the opportunities that the Web of Things brings. My work differs from theirs in the part of the problem I attack. Namely, I concentrate on lower details; making HTTP and RESTful Web services viable on IP enabled tiny networked devices whereas their work mostly focuses on higher part; developing useful mashups using the sensor data from different sensors. For that matter, the authors employ gateways to connect devices to the Internet. I believe that their and my work complete each other for the vision of the Web of Things.

SOAP-based Web services are out there and used successfully for a long time now. That is why, when one mentions Web services, SOAP-based Web services first come to mind rather than RESTful ones. There exist several work about SOAP-Based Web services on sensors in literature and I also have a look on them to make performance comparison between their and my approach; using REST principles.

The first work I have found about realizing Web services on IP-enabled sensors are the works done by Othman et al. [28, 40]. The work are about providing an embedded Web Service Framework for WSNs with an emphasis on sinkless model in which no sink or gateway is used and so applications are directly accessing the sensor nodes. The authors describe a prototype implementation using μIP over TinyOS [29] for Telosb motes [42]. Simulation results of the work are provided in which it is shown that sinkless model has a better lifetime due to lower energy consumption and network load. Although the main idea of the work to be standard compliant Web service framework, it is not mentioned why they needed to implement a client side SOAP processor instead of using any well known lightweight client side APIs. Removal of optional SOAP sections, XML namespaces and SOAP Headers, have high chance to be the reason since most of the client APIs use them. My work does not involve any client side tools since I offer true Web integration so standard Web tools are possible to interact with my framework.

Microsoft research has done implementations and evaluations in their work [44] using μIP as their TCP/IP stack. The paper has a similar aim to this thesis which is to show that connecting the sensor networks to the Internet based on widely-used open standards such as TCP/IP and it has a good job showing the overhead incurred by TCP/IP. Also the optimizations are suggested to reduce the overhead, however they do not seem very practical to perform. For example; using persistent connections is not practical at all for constrained devices that will really interact with Web because of the state that needed to be saved. For that purpose, their aim is to realize Web services on sensornets in spite of the significant energy and bandwidth overhead. The authors have achieved to connect sensors to the Internet using HTTP Binding with WSDL standard. Most of the details are not given but looking at the examples they have provided, the system they produced seems to be very specific (such as sensor providers generate WSDL and code too) and it seems to be more like an REST-RPC hybrid rather than genuine SOAP-based Web services. Real SOAP handling part is done using an intermediary that intercepts the SOAP message from the client, extract method name and parameters and send them to the relevant sensor over HTTP directly. On the other hand, I have implemented a server side SOAP API to compare it with my lightweight REST framework so my implementation do not require any protocol/data converter in between. In their work, the authors have achieved to have just 23.09 ms increase in the completion time for 40-byte respose data size which seems really promising. However, this evaluation ignores the latency incurred by request message and overheads of opening and closing connections. In my work, I add HTTP analysis and provide completion times of Web service calls with more realistic data sizes and prove that it is still reasonable.

The SenseWeb project [31] offers an architecture whose aim is to share sensor data across the Internet. The work differs from ours in that a central server and gateways are required to access sensor data. Web services are used for having a flexible and uniform mechanism.

An important aspect for my work is to have small energy and bandwidth usage. One approach to reduce bandwidth and energy overheads is to compress XML data in Web services. One can find many researches on that matter in the literature as well, i.e. a bibliography is present here [33]. Moreover, comparison between different compression techniques are provided in [8]. However, first of all, compression XML data is not a complete solution since it will only decrease the transmission and receiving power consumption but still there is the part of the problem; processing and storing big XML files. In addition, there is this additional work of XML compression/decompression on the nodes for which we need ported compression software on sensor motes that will take extra extra code space as well as will increase the processing time and energy usage. On the other hand, using a binary representation may be a good solution since parsing, storing and transmission of the file will be much more efficient however there is the issue of the standardization. W3C formed the Efficient XML Interchange (EXI) [3] is announced very recently which may solve all these problems, but providing a very small parser for EXI may be a challenge first. Another binary representation of XML is WAP Binary XML (WBXML) [36] . It is already used by some mobile devices such as phones.

// Chapter 3

RESThing

I present RESThing, a lightweight RESTful framework designed for memory-constrained sensor nodes. It offers interfaces and reusable components so that developers can develop their RESTful applications easily. Sensor network programming is hard and error prone [39]. Debugging is even harder. Therefore, my framework eases the burden of developing RESTful applications in WSNs.

The main challenge of developing software for sensor platforms is the resource bottleneck. Some of the underlying components, i.e. HTTP server, TCP/IP stack and XML parser have high complexity which makes the job harder. First of all, it is not possible to reuse PC versions of these software into sensor platforms especially due to their heavy nature, so clever design choices and careful implementation/refactoring should be done. Removing non-mandatory features is a must, however special care should be taken to have a standard compliant software for the sake of interoperability.

I provide the details of the components of the system in the following section.

3.1 Architecture Details

Keeping the challenges given above in mind, I have tried to build the software as efficient and small as possible. The data structures are designed and used in a very efficient way such as sharing the buffers for different purposes. Moreover, the system consists of a modular structure to ease the process of reusing and replacing parts of the software.

No client side support is provided in my software since there is no need. Clients can use standard web tools to communicate with my REST framework. For example I have used curl [2] and restlet [6] for testing and evaluations.

Software architecture of RESThing is shown in Figure 3.1. It consists of HTTP Server, REST Engine, SAX based XML parser and Logger modules. Developers can add a number of RESTful Web services on the REST engine as symbolized by RWS. These are nothing but realization of REST resource concept actually. RESThing offers an easy interface to create resources since they are the main abstractions of RESTful Web services.

The details of individual components are provided in the following sections.

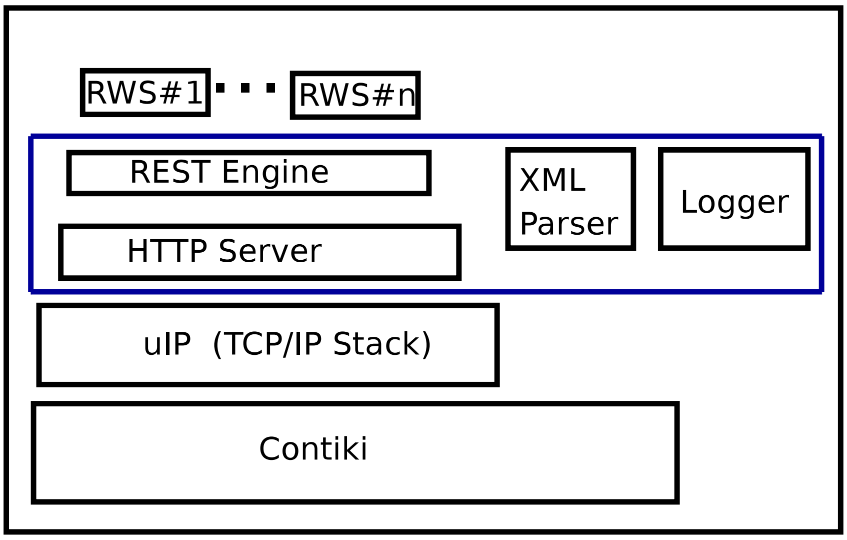


Figure 3.1: Architecture of RESThing

3.1.1 HTTP Server

HTTP server is a small footprinted server to handle the incoming and outgoing HTTP traffic. It provides interface to perform certain HTTP related stuff such as accessing request details (headers, entity body, URL path), constructing an HTTP response etc. Not only REST Engine but also SOAP Engine works on top of it. A basic SOAP Engine has been developed to see its feasibility and compare it with RESTful architecture. SOAP Engine is described in Section 3.2.

3.1.2 XML Parser

Due to data and code memory limitations, a small and memory efficient parser is needed. Simple XML parser [7] is found to be best candidate so it is ported to Contiki. It is very small in code size and being a non-validating SAX based parser makes it memory efficient. Some new functionalities such as XML generation and a basic namespace support are added. The parser is used by SOAP engine as well as user applications (both SOAP-based and RESTful Web services). SOAP engine needs it to parse incoming SOAP messages and extract related data.

3.1.3 Logger

Visibility of wireless sensor network applications are lower than traditional applications due to the inherent properties of WSNs [54]. Due to that reason, I see logging has an increased importance to monitor the events taking place. That is why I also designed a logging module in my framework. It offers three severity levels; Error, Info and Debug. Any level include levels preceding it , i.e. when Info level is defined, Error messages are printed as well. Basic mode just prints the strings whereas there is also a persistent mode which saves the log messages in flash memory (using Coffee File System [52]) periodically. So one can execute certain operations for a long time and then gather the results from the flash memory later.

3.2 SOAP Engine

I also provide a minimal SOAP processing engine for fulfilling SOAP-based Web service calls. It reuses the HTTP server and XML parser components. Engine parses the SOAP message using the XML parser, extract the method information and execute it. Then the response SOAP message is built using the XML parser

Chapter 4

Making HTTP Viable for Wireless Sensor Networks

IP-enabled sensor nodes made it possible to use protocols relying on IP such as HTTP in wireless sensor networks. Supporting HTTP in sensor nodes brings many advantages, mainly direct integration to the Web. However, these advantages do not come for free and challenges need to be handled to have HTTP protocol running on sensor nodes. Limited resources in sensor nodes requires HTTP implementations to be small in code size and conserve memory. Using TCP as the transport protocol brings some more challenges; connection setup and acknowledgments causes extra latency and bandwidth usage. Regarding HTTP, stateless model helps sensor nodes to conserve memory however HTTP is text-based and maximum packet size supported in the domain is small for a text-based protocol which may cause segmentation into several packets. HTTP offers many mechanisms and WSNs can benefit from some of them such as Conditional HTTP GET, Delta Encoding, Range Headers to have better performance.

The challenges appearing are due to the resource limitation in tiny embedded networked devices. So first of all, I have developed a minimal and small footprinted HTTP server for Tmote Sky motes over Contiki in the context of this thesis.

Reliable nature of TCP increases the latency significantly, i.e. 3-way hand-shaking mechanism to establish the connection and ACKs. Persistent connections can remove the overhead incurred by 3-way hand-shaking but it is only applicable to specific cases, such as only a single application is interacting with the sensor. Also Delayed ACK optimization implemented in most of the TCP/IP stacks have a negative effect for our domain due to small request and response sizes. These overheads of TCP over WSNs are already studied and evaluated by Nissanka et al. in [44]. Header compression mechanisms for TCP and/or IP may alleviate the problems by increasing throughput and decreasing latency.

Additional negative effects related to TCP are possible especially due to minimal resources. For example; TCP/IP stacks available in sensor nodes must have minimal buffers so TCP window size may be very limited such as a single packet size. This makes the situation worse since throughput reduces further, also Delayed ACK situation gets worse since every packet sent experiences 200ms delay. This situation should be handled by increasing the input/output buffer size of the TCP/IP stack in the sensors. However, as a side note, I did not experience Delayed ACK problem in my measurements. Although I could not find any documentation, it seems that TCP/IP stack in Ubuntu 9.04 has some form of smart Delayed ACK optimization maybe relying on the previous traffic. Lastly, one interesting point is; although slow start behaviour of TCP is a problem for general short lived HTTP traffic in the Internet, it does not necessarily affects WSNs due to very small TCP window.

Yet another big challenge is to conserve power while achieving reasonable completion times. For that matter, I analyze TCP over X-MAC and present Session-aware X-MAC; an optimized version of X-MAC protocol that is aware of TCP behaviour hence providing improved completion times. I analyze TCP over X-MAC and describe Session-aware X-MAC in Section 4.1.

Stateless nature of HTTP is a big advantage for our limited nodes since no client application state needed to be kept in already limited memories, however it is a text based protocol and maximum packet size supported by 802.15.4 is only 127 bytes which will also include TCP, IP and MAC layer headers. Therefore, data may need to be segmented into several packets which will bring extra latency. So, messages transmitted in between should be as compact as possible, i.e. unnecessary and long HTTP headers should be avoided by the clients. Moreover, certain other optimizations should be used to alleviate the challenges due to the characteristics of sensor networks. This thesis focuses on realizing everything using standard methods for the sake of interoperability, hence current Web techniques are considered, for example, HTTP standard has a perfect caching structure which is one of the main reasons that World Wide Web scales that good. The mechanisms are discussed in Section 4.2, only with their usages relevant to our topic.

4.1 Session-aware X-MAC: A TCP Friendly X-MAC

When interacting with wireless sensor networks, getting good completion times is not enough, you need to also do it in an energy efficient way. For that purpose, I use X-MAC [13] as MAC layer protocol. Although X-MAC is being used efficiently in typical WSN applications for some time now, to my knowledge there is no study about performance of X-MAC as a lower layer for TCP traffic and how to make it suit better to the TCP connections, so I make my own tests to see the results. X-MAC turns off the radio to save energy and this behaviour is causing significant delays in the TCP communications.

In a typical TCP communication, there is continuous traffic in both directions until the connection is closed. This is mainly because TCP is a reliable communication protocol; ACKs are transmitted in the opposite direction of the data delivery. This means that both packets and their corresponding ACKs suffer from the wake-up time imposed by X-MAC. The problem is more clear in our resource limited domain since the TCP/IP stack used, namely μIP, has a TCP window for just one packet, so each packet has to be acked to continue the rest of the transmission.

The problem may be alleviated by letting the radio stay on for a while after a mote sends a packet to the client, i.e. until the ACK is received. By this way, at least the ACKs destined to the sensor node do not suffer from wake-up delay. However, this does not solve the problem completely since the packets still may experience delays so a more thorough solution is required such as letting the radio be on for the entire communication.

I present Session-aware X-MAC which is a TCP aware X-MAC derivative. Session-aware X-MAC lets the radio be switched on during a TCP connection; precisely between the periods of SYN packet reception and FIN packet transmission. This solution decreases the delays significantly since the only packet that suffers from wake-up delay is the first SYN packet.

Evaluation comparing Session-aware X-MAC with original X-MAC can be found in Section 6.

4.2 HTTP Optimizations

4.2.1 Conditional HTTP GET

In this section, I focus on Conditional HTTP GET, an existing HTTP optimization mechanism, since I foresee that it is an important mechanism to connect WSNs to the Web efficiently. First I give details about it and then describe how it is applied in our domain.

Conditional HTTP GET is designed to save time and bandwidth by employing certain response (Last-Modified and ETag) and request headers (If-Modified-Since and If-None-Match). The idea is if the data is not changed after the last time client fetched it, the server can notify client by 304 (Not Modified) status and do not send the data again hence saving bandwidth and time. But how does that work? Every time a server sends data, it includes Last-Modified (last time the data was changed) and/or ETag headers (opaque string symbolizing a specific version of data) and when the client asks for the same resource later it provides these information in If-Modified-Since and If-None-Match headers, hence allowing the server to make a decision whether the resource has changed or not. If it is changed, a response code of 200 (OK) and the new data in the entity-body is served, or else 304 (Not Modified) is returned only, then the client uses its cached data knowing the fact that the underlying data hasn't changed since the first request.

The Last-Modified header field indicates the date and time at which the resource was last modified. This creates a problem for wireless sensor networks; the synchronization of the date and time is needed. Either the clock of the mote should be synchronized during installation or only ETag header should be used. The latter seems like a better solution to for the sake of easiness. ETag is an opaque string so it is very flexible for server to choose a representation for it, usually some version system or checksum is employed though. The only problem using ETag alone is that it is only HTTP1.1 compliant which is not a big problem since almost all clients support it nowadays.

An example for the scenario in Section 7.1.1 is as follows: Sensor will fulfill the requests with ETag header.:

```
HTTP/1.1 200 OK
Server: Contiki
Content-Type: text/xml
ETag: v1
[Entity Body]
```

And client will provide the same string retrieved from server in ETag into If-None-Match:

```
GET /temperature HTTP/1.1
Host: www.example.com
If-None-Match: v1
```

And if the content is not changed, sensor will not transmit Entity Body hence saving bandwidth. Status 304 (Not Modified) is returned:

```
HTTP/1.1 304 Not Modified
Server: Contiki
Content-Type: text/xml
ETag: v1
```

4.2.2 Delta Encoding

Delta Encoding in HTTP is defined by RFC 3229 [38]. Using delta encoding, rather than the whole document, the client can ask for a difference against his/her own copy from the server. The server knows which version client has with the If-None-Match header. An example request is copied below for clarity.

```
GET /foo.html HTTP/1.1
If-None-Match: v2
A-IM: diffe
```

The main idea is that the server knows the differences of at least the recent versions of a resource presentation. The main challenge for sensor networks is that it may be a little bit complicated to keep the differences.

4.2.3 Range Header

By using Range Header (that is defined in RFC 2616 [25]), client can obtain only a part of a resource representation. So in case of a situation that only part of the data is needed and it is known in advance, this mechanism can be employed. An example of getting the first 31 bytes:

```
GET /foo HTTP/1.1
Host: www.example.com
Range: bytes=0-31
```

4.2.4 Other HTTP Optimizations

There exist more subjects related to this subject that are expected to be included in HTTP standard in a very near future. These are prefer header [50], patch method [24] and batched HTTP requests [51].

PATCH method allows you to do a partial update on a resource which saves bandwidth since instead of sending the whole data over the network, only a set of changes are sent. Server will apply the changes to the resource and tell user what happens. The challenge here for a sensor node would be to recognize and handle the change format (i.e. diff).

Prefer header allows the clients to describe the format of response (not the mime type of content) they wants to receive. In relevance to our case, for example the client may prefer that the server not include an entity in the response to a successful request since the status code may be enough (i.e. 204 No Content or 304 Not Modified, etc). I believe this header proves that Web is not only for human beings anymore, browsers indeed need content to show it to the users but machines do not necessarily need content for all requests.

Batched HTTP allows multiple requests to be sent altogether over a persistent connection without waiting for individual responses. Requests can be both idempotent and non-idempotent. Batched HTTP requests can decrease latency and number of TCP packets transmitted.

Chapter 5

Implementation

I have implemented HTTP Server, REST Engine and Logger modules from scratch while Simple XML parser [7] is refactored and ported to Contiki environment. Certain optimizations and limitations exist in these modules to make them small and efficient enough for sensor motes. For example, HTTP server only saves the headers which are specified beforehand by the developer and unnecessary features of Simple XML Parser is stripped whereas XML creation support and a basic namespace support is added. Logger is implemented using C Macros so that configuring it would be done in compile time efficiently. It is possible to remove the logging support during compile time in which case log statements are expanded to nothing.

While implementing RESThing, I have taken the constraints imposed by low-cost sensor nodes into account to achieve a small footprint solution. I use C as the programming language. C offers a good control over memory and also Contiki APIs provide C interfaces, so I believe it is the most suitable choice for the context of this thesis even though it is not particularly thought as a Web application friendly language. I implement the code using Tmote Sky motes, however since it does not use any low level details of the mote platform, it is possible to extend it easily to other Contiki ported platforms in the future.

Static memory allocation is prioritized over dynamic allocation for the sake of reliability and dynamic memory allocation is used as less as possible since on a memory-constrained system it may make the heap fragmented. All programs share the same address space so having a fragmented heap would cause unpredictable problems. So special care taken while allocating memory from heap; it is only done where efficiency is important and for short periods. Needed memory is allocated as a whole to minimize the fragmentation. To be able to respond to maximum number of clients at the same time, data structures related to connection state are allocated on heap. Also, XML parser uses heap memory during parsing. It is necessary since number of elements and their length is unpredictable so having static memory assigned is not only inefficient but also inflexible.

WS-* stack is very big to fit in sensor nodes so my SOAP Engine is very light and implementation lacks many features. Also, a very limited validation of provided SOAP messages are provided. SOAP Engine is used to evaluate bandwidth and memory requirements and compare it with my REST framework.

Module	Code Size (ROM)	RAM Usage (Static)
HTTP Server	3976	72
REST Engine	692	4
XML Parser	5260	4
Logger (Basic)	34	2
Logger (Persistent)	710	20
SOAP Engine	2354	36

Table 5.1: Memory footprint of the modules.

The details of the implementation of modules regarding memory usage is given in Table 5.1. Dynamic memory usage is not given since it depends on the user settings and XML documents to be parsed. More precisely, user can decide connection state settings (i.e. request and response buffer sizes, number of headers to be handled etc) which affect the dynamic memory usage and also XML parser uses dynamic memory allocation for the sake of efficiency.

Although this thesis has emphasis on layers above network layer, to get a good performance, I had to get my hands dirty in the lower layers as well. Due to a preemption interference between timer handling of X-MAC and SLIP, I end up having unreliable behaviour, i.e. quite different completion times, in my evaluations. As described in Section 6, the connection between router and the computer is established using SLIP which basically encapsulates the IP packets in both directions. However, I have experienced packet losses over serial line which increased the completion time significantly. Mainly, the problem was because of the timer handling of X-MAC was creating a big burden over gateway and gateway was missing some part of data causing the checksum to fail and leading further retransmissions. The solution for that problem is simply let the radio be on for router all the time since router is already connected to computer which means it has always enough power so do not need to be battery friendly. Having the router whose radio is always on provides not only consistent but also slightly improved completion times. I call this preemption interference tweaking as Preemption Interference Fix (PIF) and evaluate its effect in Section 6.

Chapter 6

Evaluation

In the first part of this section, I evaluate X-MAC optimizations I have performed to have better TCP behaviour. This is especially important since later evaluations are done using these optimizations.

Next, I evaluate RESTful Web services in terms of power consumption and completion time using two different MAC level protocols, namely X-MAC and NULLMAC. I show that realizing RESTful Web services is possible and reasonable even with power saving MAC protocols.

Then, I evaluate SOAP-based Web services. The main goal is to compare the results with RESTful Web services to prove that although it is possible to serve SOAP-based Web services on these tiny embedded devices, it has much more complexity than RESTful Web services in terms of communication costs as well as memory requirements. However, the aim of this comparison is definitely not to find out which one is better. There exist already a lot of discussions about that topic.

I also estimate the battery life of sensor nodes running Web services using the power consumption data obtained. The estimation is useful to see how long a typical sensor node that serves Web services can stand.

Lastly, I estimate and present energy consumption on byte level especially to see the cost of TCP overhead on byte level.

6.1 Experimental Setup and Details

The experimental setup, shown in Figure 6.1, consists of a testbed of Tmote Sky motes and a desktop computer running Ubuntu Linux. One mote is used as a router that connects the sensor IP network and the desktop computer. The motes run Contiki as operating system and use μIP TCP/IP stack. Router sensor is connected to desktop computer via serial cable and to deliver/receive packets to/from router, SLIP and TUN are used. TUN simulates layer 3 (network layer) devices and is used for routing, whereas SLIP is used to encapsulate IP packets transported in between. Although SLIP is obsolete now, it is still popular for networked embedded devices thanks to its small overhead. Lastly, RESTful requests are done using curl [2].

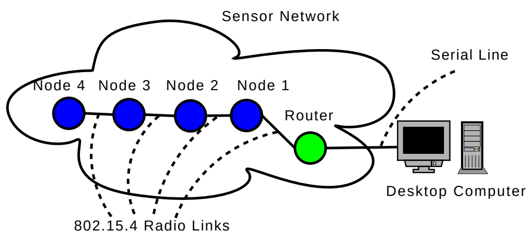


Figure 6.1: Experimental Setup

The only exception for the above setting is the completion time measurement using IPv6. In that measurement, μIPv6 TCP/IP stack is employed and TAN is used rather than TUN. The sensor node marked as router in Figure 6.1 is actually used as a bridge, acting as a link layer interface for the Linux host, in this setting. SICSlowpan header compression implementation of the 6LoWPAN adaption layer is used by Contiki over IPv6 packets.

X-MAC is used with two different configurations in these evaluations. Main goal is to compare the power consumption and completion times of these settings and hence to provide input about their feasibility. Differently configured X-MACs are labeled as X-MAC and X-MAC2. The only changed parameter is the Off Time parameter which presents the period of keeping the radio off until waking up to listen for the strobes. The configurations are:

- X-MAC
  - On Time: 1/200s
  - Off Time: 1/4s
- X-MAC2
  - On Time: 1/200s
  - Off Time: 1/2s

The testbed setup is intentionally simple to avoid irrelevant network effects. Pre-configured routing tables are used on every node in all experiments to avoid any effects of a dynamic routing protocol to influence the measurements.

6.2 Completion Time Improvement for TCP Connections over X-MAC

To measure the effects of the optimizations done on X-MAC usage, I measure completion times and power consumption both for original and optimized versions for the sensor node. I perform two optimizations, namely Session-aware X-MAC and Preemption Interference Fix, to have a better and more reliable TCP behaviour over X-MAC. Session-aware X-MAC is described in Section 4.1 and Preemption Interference Fix (PIF) is explained in Section 5.

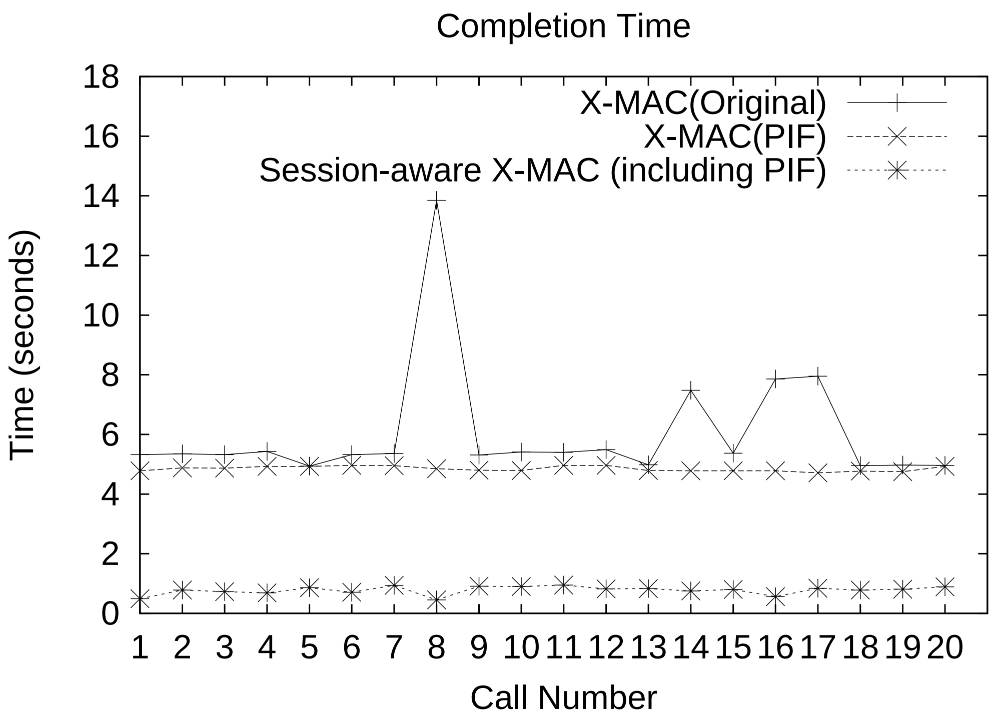


Figure 6.2: Preemption Interference Fix provides more stable completion times and Session-aware X-MAC decreases completion times significantly.

Sensors Web service, shown in Table 6.1, is used to obtain completion times and power consumption which can be found in Figures 6.2 and 6.3 respectively. X-MAC2 settings are used in the evaluation. Web services are called 5 times in 4 sets. Power consumption is measured for each set (5 Web services are called in 60 seconds), averaged and represented with their standard deviation. Measurements using Session-aware X-MAC also includes Preemption Interference Fix.

Regarding completion times, it is seen that Preemption Interference Fix lead more reliable results than original X-MAC. Original version has 4 spikes at 8th, 14th, 16th and 17th calls. With Preemption Interference Fix, X-MAC has more stable results since spikes do not exist. There are only small fluctuations which are normal and caused by difference in X-MAC timing to switch on the radio. Also it is clearly seen that completion times are improved slightly.

Session-aware X-MAC outperforms original X-MAC in terms of completion times. There are only small fluctuations, i.e. 1st, 8th and 16th calls are luckier than the others since they are roughly synchronized with the radio wake up period.

Regarding power consumption, Preemption Interference Fix decreases power consumption for the sensor node significantly since the router keeps the radio switched on all the time, sensor node does not need to send a lot of preambles to wake the router up and there-

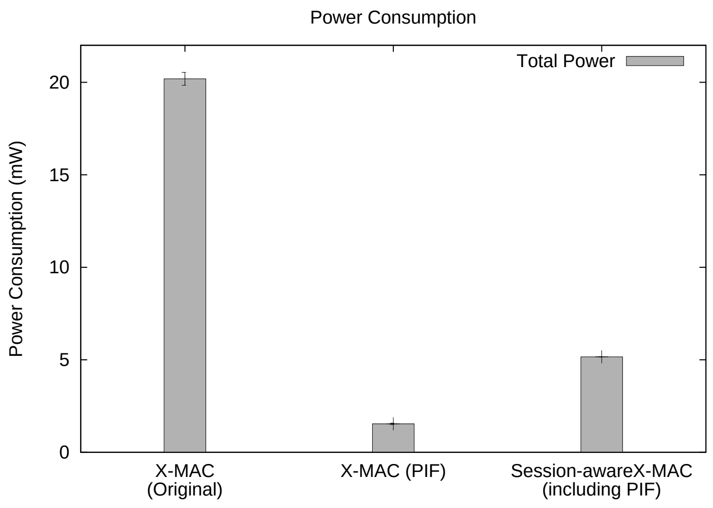


Figure 6.3: Preemption Interference Fix has much more smaller power consumption for 1-hop away node than original X-MAC. Session-aware X-MAC uses more energy for the sake of better responsiveness.

fore power consumption decreases thanks to less data transmission in total. Also, original X-MAC has bigger standard deviation than Preemption Interference Fix in power consumption which is expected due to the spikes.

Session-aware X-MAC consumes more power than when we employ Preemption Interference Fix alone. This behaviour is normal since Session-aware X-MAC keeps the radio switched on throughout the whole TCP connection. However, thinking it together with the great increase in the responsiveness of the system, power consumption increase is reasonable.

6.3 Power Consumption and Completion Times of RESTful Sensor Networks

I evaluate serving RESTful Web services on Tmote Sky motes in terms of power consumption and completion time. My evaluation confirms that it is reasonable to realize RESTful Web services on wireless sensor networks even though there exists significant overhead resulting from TCP/IP and the verbose nature of HTTP and XML standards.

Different RESTful Web service calls are analyzed to monitor the effect of different data sizes over power consumption and completion time and results from both X-MAC and NULLMAC protocol is provided for comparison, especially to prove that with the help of certain optimizations, it is possible to get good completion times using battery conserving MAC protocols.

The details of these services are provided in Table 6.1.

Among the RESTful Web services chosen, Dummy service is chosen as a small example

Web Service	Request Size (bytes)	Response Size (bytes)	Total Size (bytes)
Dummy	84	48	132
LED Control	89	52	141
Light	79	135	214
Temperature	85	141	226
Sensors	81	324	405

Table 6.1: Details of RESTful Web services. Sizes do not include TCP and lower layer header sizes.

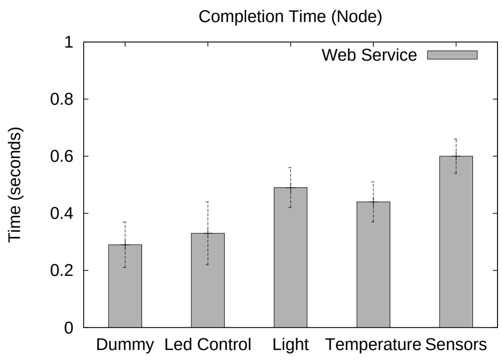


Figure 6.4: Completion times of Web services on sensor node.

whereas Sensors service has the biggest response and total data size.

Moreover, among the possible HTTP level optimizations described in Chapter 4, Conditional GET is applied to Sensors and Temperature Web services to show the effectiveness of standard optimization techniques.

Completion Time

Completion times are obtained using built-in `time` command of bash shell to measure the interval between issuing the Web service call via curl tool [2] and getting the response. Figure 6.4 shows the results of the measurements of the single-hop who communicates with the router using radio communication. The results are promising in the sense that requests can be fulfilled within a second using a power conserving MAC protocol.

Figure 6.5 shows completion time measurements of Web service calls to the router. This measurement is interesting since we have the chance to see overhead of the connection between Desktop computer and the router (overhead of serial line, SLIP and TUN). Additionally, by comparing it with Figure 6.4 it is possible to see the overhead of relaying the request over radio interface.

I also measure completion times of Web services using NULLMAC which does not conserve power and X-MAC2 which is more power conservative than X-MAC setup used above. The result can be seen in Figure 6.6 which is obtained by calling two Web services mentioned in the figure. As expected NULLMAC gives better results since it does

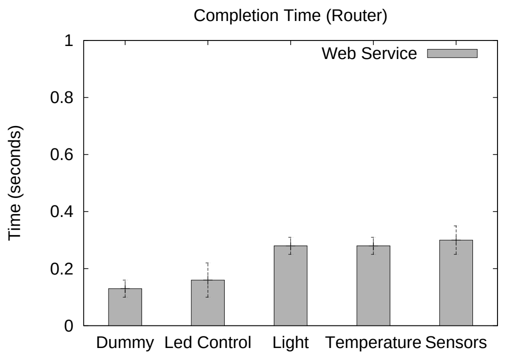


Figure 6.5: Completion times of Web services on router sensor node. Communication is done over serial line, therefore no radio communication overhead exists.

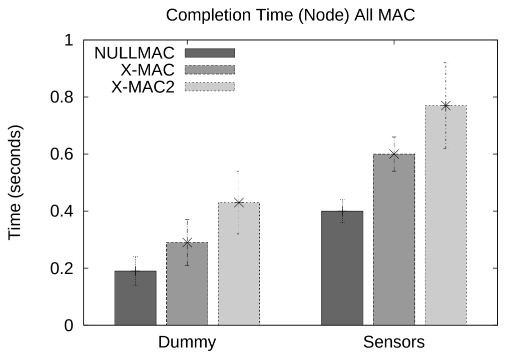


Figure 6.6: Completion times using different MAC protocols. NULLMAC performs the best because it does not switch off the radio.

not switch off the radio at all, however employing Session-aware X-MAC and Preemption Interference Fix optimizations, the difference between NULLMAC and X-MAC is small enough. On the other hand, X-MAC2 has doubled Off Time so the completion times are bigger.

Lastly, I measure completion times over IPv6. NULLMAC is used as MAC protocol and results are given in Figure 6.7. The results prove that IPv6 offers reasonable performance over wireless sensor networks.

Power Consumption

I evaluate power consumption of sensor motes that serve RESTful Web services. Services are called 5 times in 60 seconds period and power consumption is tracked using Contiki's power profiler [21]. Figure 6.8 shows power measurements of Web services calls over 1-hop away neighbour. To have more insight about where the energy is actually spent, I also provide the parts of the power consumption for Sensors service in Figure 6.9.

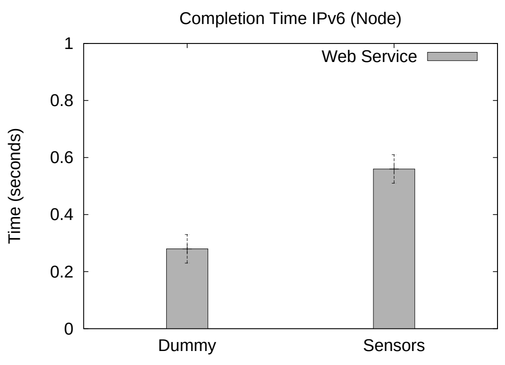


Figure 6.7: Completion times of Web services over IPv6 using NULLMAC.

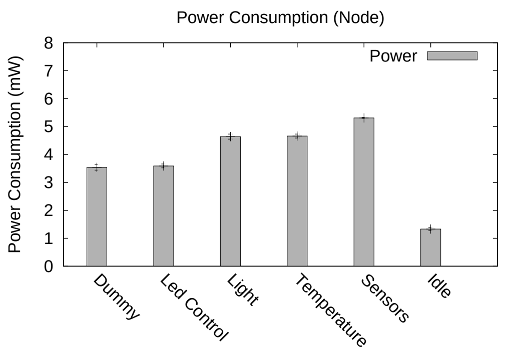


Figure 6.8: The cost of calling Web services in terms of power consumption. X-MAC is used as the MAC protocol.

Furthermore, Figure 6.10 shows the power usage of NULLMAC, X-MAC and X-MAC2 for two Web services as well as for a idle status in 60 seconds period. When node is idle, X-MAC2 conserves power better than X-MAC since it keeps the radio switched off longer. This also explains the measurement of Sensors Web service. However, one interesting observation is that X-MAC seems to save power more than X-MAC2 during Dummy Web service. This is a consequence of Session-aware X-MAC which keeps the radio on during TCP connection. Having less completion times, transmissions/receptions take less time with X-MAC and idle energy consumption can not suppress it for that example.

The bottom line of MAC protocol analysis is that the one suitable for the purpose should be chosen. For nodes connected to power supply all the time, NULLMAC may be the best solution of course, however depending on the trade-off between power utilization and latency, settings of X-MAC should be played with to find the optimal settings for the particular problems.

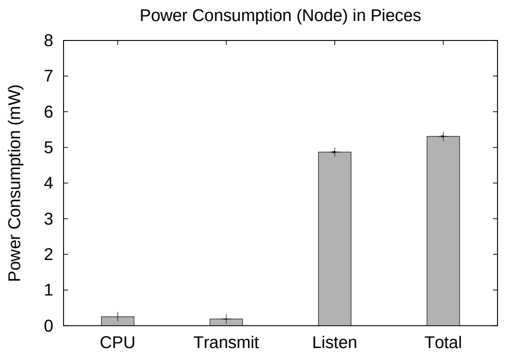


Figure 6.9: Power consumption of CPU, transmission, listening and their sum.

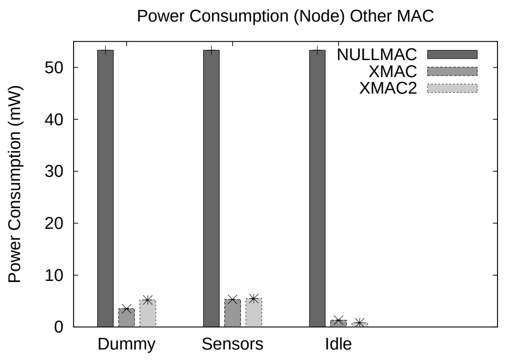


Figure 6.10: Power consumption compared with different MAC protocols.

Evaluation of Conditional GET

Conditional GET is a caching technique in which the client gets the content from its cache if data is not changed. The details are described in Section 4.2.1. I evaluate it to show that standard web optimizations which made Web scalable as it is today are likely to succeed for WSNs too. Two Web service examples, Sensors and Temperature from Table 6.1, are analyzed using Conditional GET and compared with the original results. Details of the cached versions is in Table 6.2. Figure 6.11 provides completion times and power consumption measurements. X-MAC is used for the evaluations. The results are given in Table 6.3.

Cached versions have a little bit bigger request data size because of the extra ETag header they transmit, whereas they have significantly smaller response data sizes since they

Web Service	Request Size	Response Size	Total Size
Temperature	91	55	146
Sensors	87	55	142

Table 6.2: Details of Web services employing Conditional GET.

Web Service	Data Size Decrease	Power Save	Completion Time Decrease
Temperature	35.4%	24.0%	31.8%
Sensors	64.9%	33.1%	53.3%

Table 6.3: Performance improvement provided by Conditional GET.

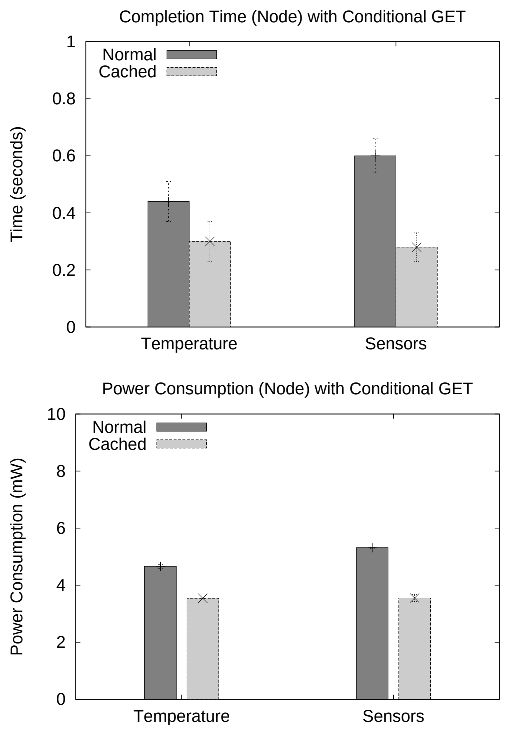


Figure 6.11: Conditional GET improves responses and power consumption significantly.

do not include any data content. As it is seen, the response sizes for the cached versions are same for both services since the same data is transfered, namely only the headers which happens to be the same for these examples. This also explains why the completion times are roughly same. Also, as expected, cached Sensors Web service performs better for saving power as well as decreasing completion time (in percentage) than Temperature service which is consistent to the bandwidth saves.

Results in a Multi-hop Network

In order to evaluate the effect of multi-hop communication for sensor network Web services, I measure completion times of a set of Web services over a multi-hop network. I use the Session-aware X-MAC on every hop of the network. Figure 6.12 shows the measured completion times, with a varied number of hops. The results show that delay caused by relaying RESTful requests in a wireless sensor network is quite reasonable even in a multi-

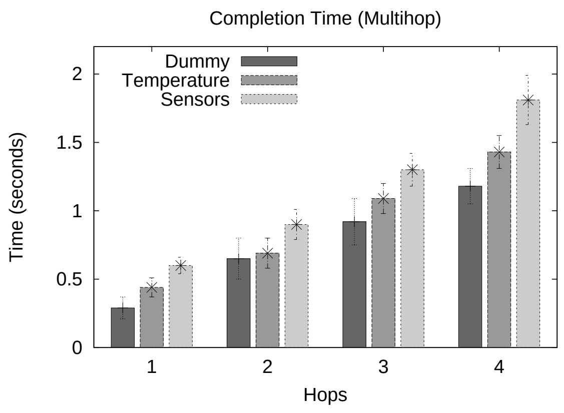


Figure 6.12: Completion times of Web services over multiple hops.

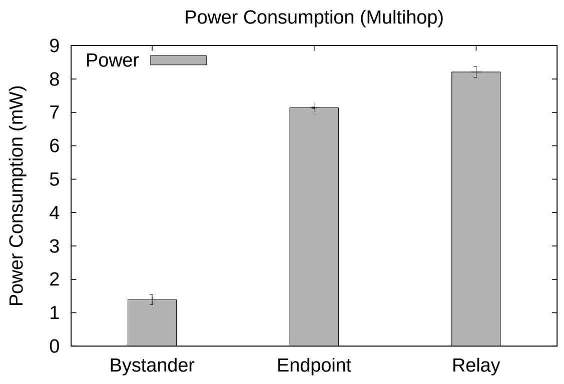


Figure 6.13: Power consumption of a bystander node, an endpoint node, and a relay node in a multihop network.

hop network.

Figure 6.13 shows the power consumption of three nodes in the multi-hop network. The figure shows the power consumption of a bystander node (not serving any Web service nor relaying it), an endpoint node (actually serving the Web service), and a relayer node (Web service is served by the next hop node). The Sensors Web service is used for all measurements. The results show that the power consumption increases for nodes that are either endpoints or relay nodes. Relay nodes have a slightly higher power consumption because the session-aware MAC protocol enables duty cycling some time after the session has been closed by the endpoint node.

Web Service	Request Size	Response Size	Total Size
LED Control	576	498	1074

Table 6.4: Details of SOAP-based Web service

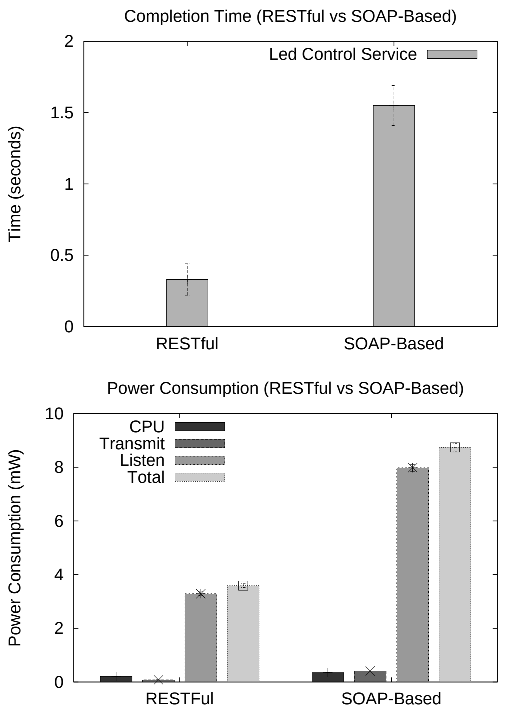


Figure 6.14: RESTful Web service outperforms SOAP-based Web service in power consumption as well as completion time.

6.4 RESTful Web Services vs SOAP-Based Web Services

I show that having SOAP-based Web services in sensor motes is also possible with the prototype I developed and the evaluations given. However, SOAP-based Web services use verbose SOAP messages over HTTP which makes these Web services less reasonable than RESTful Web services for wireless sensor networks. The two main reasons for that are:

– Verbose SOAP messages cause extra latency and transmission/reception power usage.

– Memory space required for SOAP implementation and execution is much more higher both in data and instruction memory.

As a side note, my aim here is to compare overhead of REST approach with SOAP rather than claiming REST approach is better than SOAP. To do the comparison, I have

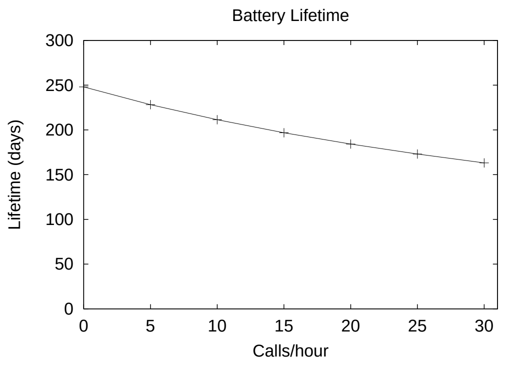


Figure 6.15: Battery life vs Number of Service calls

selected LED Control RESTful Web service (given in Table 6.1) and implemented it as a SOAP-based Web service. The details of the SOAP-based Web service is given in Table 6.4.

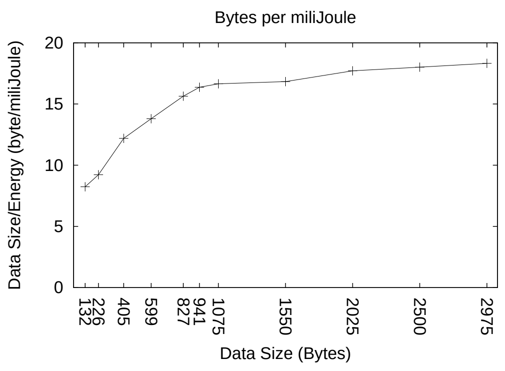
I evaluate LED Control as a SOAP-based Web service to compare it with the RESTful counterpart. csoap [1] is used as the client library to make SOAP-Based Web service calls and X-MAC is used as MAC layer protocol. Cost of both Web services in terms of power consumption and completion time are given in Figure 6.14. As it is expected, SOAP-based Web service consumes much more energy, especially for communication. However, the increase of processing power usage due to parse and processing of SOAP messages are ignorable when compared to power usage of radio. This is due to the fact that power consumption of CPU is significantly smaller than the consumption of transceiver in a typical sensor node. Regarding completion times, RESTful Web service outperforms SOAP-based one as expected.

6.5 Battery Lifetime

I estimate battery lifetime of a sensor node serving a typical sensor monitoring service: Temperature service in Table 6.1. I assume to have two AA batteries offering 2.5 Watt-hours each and X-MAC2 setting is used. Then, estimate battery life of the sensor node depending on the number of calls is as in Figure 6.15.

6.6 Energy Consumption on Byte Level

I measure energy consumption and calculate throughput/power and power/throughput to find out energy consumption on byte level. I call various sized Web services and Figure 6.16 presents the results. As expected, number of bytes for unit energy increases when data size of the Web service increases since TCP overheads (especially connection establishment and closing) are less dominant for bigger data sizes. However, the maximum packet size supported by 802.15.4 is only 127 bytes which means that even for big data sizes, TCP header, IP header and ACKs have almost same overhead, so the graph gets stable quickly.



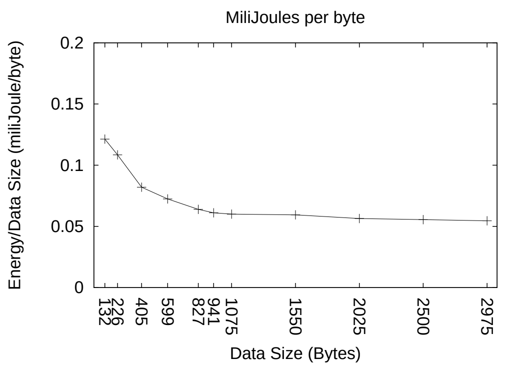


Figure 6.16: For bigger data sizes, TCP overheads are less effective so number of bytes per unit energy increases.

Chapter 7

Conclusions and Future Work

Currently, the Internet has limited capability of monitoring and acting on the real world. Both network layer and application layer interoperability are needed between WSNs and networked objects in the Internet to solve this problem. Emergence of IP-enabled smart objects has already solved the former issue by using IP as the network layer. And this thesis focuses on solving the latter issue; employing RESTful Web services on WSNs to provide an interoperable application layer. Up to now, diverse structures of sensor nodes and the non-standard protocols used were the biggest obstacles in front of integrating wireless sensor networks into the Web. Standard and widely used protocols are needed. Typically, employing HTTP and XML, RESTful Web services is a very good option for that matter. Overheads introduced, especially due to the verbose nature of HTTP and XML specifications, are reasonable compared to the advantages gained using them. Hence, realizing Web services on sensor nodes is feasible since both HTTP and XML are well known, proven and highly used specifications which means that they increase interoperability and reliability of the wireless sensor networks. Being quite light and simple, RESTful Web services is a good candidate to be an important part of Future Internet for wireless sensor networks integration because of the exact reasons the Web flourished. So, REST architecture style seems to be an important candidate to connect sensor nodes to each other and to the Internet on the way of achieving the Web of Things. Extending the Web with WSNs enables new, promising and easily developed applications.

In the vision of the Web of Things, this thesis presents a RESTful Web service architecture for sensor networks that allow direct integration between the Web and IP-based sensor networks. I provide an extensive performance evaluation of the system, showing sub-second completion time of RESTful Web service requests to low-power sensor nodes in both single-hop and multi-hop networks.

7.1 Future Work

Security was not a concern in this thesis. Only plain HTTP requests are used which means that eavesdropping between the clients and the servers are possible. Using HTTPS would solve this problem by securing communication on transport level, so an interesting challenge for future work would be the addition of HTTPS on these tiny devices.

Some improvements for the XML parsing is possible. Currently, I use a non-validating SAX parser, so an idea would be to add validation support in future based on a schema. However, that may complicate the parser too much for a memory limited environment. On the other hand, according to my experience writing SAX-based codes are much more harder and complicated than equivalent DOM-based codes. Due to efficiency reasons I mentioned before, it is very hard to use a DOM parser in sensor nodes, however implementing/porting pull parsers and compare it with SAX counterpart would be a good challenge. Pull parsers are well-known for their efficiency as well as easy usage. Besides, another aspect is exchanging a more compact form of XML to save bandwidth and hence energy in significant amounts. EXI seems to be very promising to allow efficient interchange of XML documents and an open source Java implementation named EXIficient [4] exists. Developing an EXI parser for sensor platforms would make it possible to exchange compact XML documents in an interoperable way.

This thesis focused mainly on using XML as representation of data however JSON is getting popular more and more each day in Web 2.0 applications which makes it worth to look into in future such as implementing/porting a small JSON parser for nodes.

Caching in HTTP is one of the main reasons that made the Web very scalable. Traditional web servers implement some set of caching elements to make it work with the clients and proxies. So an interesting future work would be to search for the best approaches to make caching as efficient as possible for tiny servers in WSNs.

A very important work area would be to analyze how well RESTful approaches can encounter typical sensor application problems. For that matter, I present and analyze a RESTful approach to solve typical real world monitoring problems in the following sections. I leave the implementation and evaluation of the work as future work.

7.1.1 Solving Asynchronous Real World Problems in a RESTful Way

Typical sensor network applications usually benefit from asynchronous behaviour. I believe it is feasible for wireless sensor networks to have the illusion of asynchronism by splitting the operation into two or more synchronous requests as described in [46]. The first request creates the operation while the subsequent ones are used to get informed about the result of the operation.

Suppose for example, we want to create a task in which temperature values will be read in 30 minutes intervals. Then we can create a new task resource by POST:

```
POST /task?type=temperature;period=30 HTTP/1.1
Host: www.example.com
```

The sensor accepts the request, creates a temperature monitoring task. If it is a long time running task, sensor can return 202 (Accepted) status code immediately to notify that the operation is created:

```
HTTP/1.1 202 Accepted
Location: http://www.example.com/task/id123
[Entity Body]
```

Now the sensor created a new task as a resource and returned its URI so that client can make GET requests to this particular URI and see the current state of the task:

```
GET /task/id123 HTTP/1.1
Host: www.example.com
```

In our particular example, this may return the last number of readings, or the most up-to-date reading, or even something more complicated such as the collection of temperatures from the nearby sensors. The returned representation may be XML, JSON or something else depending on the client's request and sensors' capabilities.

Later client can cancel the task by using DELETE method of HTTP:

```
DELETE /task/id123 HTTP/1.1
Host: www.example.com
```

7.1.2 Discussion

In this section, I analyze the idea about how well it solves the real world monitoring problems stated in Section 1.3. This approach solves the second problem, time-consuming operations, without any extra support. For example, the process of a data collection in a wireless sensor network can be quite time consuming and keeping the connection open may not be a good idea. However, the above mechanism fits very well. Regarding the first and third problems -immediate access to new readings and network utilization respectively- still client needs to poll the resource constantly to access newly updated data, however using optimizations, it can be made very feasible. One optimization example is the standard client side caching mechanism named Conditional GET that saves bandwidth to decrease the effect of intermediate pollings when the result did not change, i.e. no new temperature reading for our particular example, or operation still in progress. Almost all of the major web tools support it already. Details of Conditional GET and how it can be used in scenario here can be found in Chapter 4.

This approach is very flexible since a lot of different usages are possible especially by using the power of URIs. For example; Collect both light and temperature:

```
/task/light;temperature
```

Collect light values between given periods:

```
/task?type=light?start=18.30;end=21.30 or /task?type=light/18.30-21.30
```

Appendix A

Code API

A.1 TYPE DEFINITIONS

```
#ifndef bool
  #define bool unsigned char
#endif /*bool*/

#ifndef true
  #define true 1
#endif /*true*/

#ifndef false
  #define false 0
#endif /*false*/
```

Type definitions

A.2 HTTP-COMMON

```
#define STATE_WAITING 0
#define STATE_OUTPUT 1
```

current state of the request, waiting: handling request, output: sending response

```
#define LINE_FEED_CHAR '\n'
#define CARRIAGE_RETURN_CHAR '\r'
```

definitions of the line ending characters

```
extern const char* httpString;
```

needed for web services giving all path (http://172.16.79.0/services/light1)
instead relative (/services/light1) in HTTP request. Ex: Restlet lib. does it

```
extern const char* httpGetString;
extern const char* httpHeadString;
extern const char* httpPostString;
extern const char* httpPutString;
extern const char* httpDeleteString;
```

HTTP method strings

```
extern const char* spaceString;
extern const char* httpv1_1;
extern const char* lineEnd;
extern const char* contiki;
extern const char* close;
extern const char* headerDelimiter;
```

Various other strings

```
extern const char* HTTP_HEADER_NAME_CONTENT_TYPE;
extern const char* HTTP_HEADER_NAME_CONTENT_LENGTH;
extern const char* HTTP_HEADER_NAME_LOCATION;
extern const char* HTTP_HEADER_NAME_CONNECTION;
extern const char* HTTP_HEADER_NAME_SERVER;
extern const char* HTTP_HEADER_NAME_HOST;
extern const char* HTTP_HEADER_NAME_IF_NONE_MATCH;
extern const char* HTTP_HEADER_NAME_ETAG;
```

header names

```
#define PORT 8080
#define RESPONSE_BUFFER_SIZE 500
#define MAX_REQUEST_HEADERS 6
#define MAX_RESPONSE_HEADERS 6
#define REQUEST_BUFFER_SIZE 1000
#define MAX_URL_MATCHED_ATTRS 4
#define INCOMING_DATA_BUFF_SIZE 102 /*100+2, 100 = max url len, 2 = space char+'\0'*/
```

Configuration parameters

```
typedef enum
{
  NO_ERROR,
  /*Memory errors*/
  MEMORY_ALLOC_ERR,
  MEMORY_BOUNDARY_EXCEEDED,
  /*specific errors*/
  XML_NOT_VALID,
  SOAP_MESSAGE_NOT_VALID,
  URL_TOO_LONG,
  URL_INVALID
} Error_t;
```

error definitions

```
typedef enum
{
  TEXT_PLAIN, TEXT_XML, APPLICATION_XML,
  APPLICATION_JSON, APPLICATION_WWW_FORM, APPLICATION_ATOM_XML
} MediaType_t;

typedef enum
{
    CLIENT_ERROR_BAD_REQUEST = 400,
    CLIENT_ERROR_METHOD_NOT_ALLOWED = 405,
    CLIENT_ERROR_NOT_ACCEPTABLE = 406,
    CLIENT_ERROR_NOT_FOUND = 404,
    CLIENT_ERROR_REQUEST_URI_TOO_LONG =
            414,
    CLIENT_ERROR_UNSUPPORTED_MEDIA_TYPE =
            415,
    NOT_MODIFIED = 304,
    SERVER_ERROR_INTERNAL = 500,

    SERVER_ERROR_NOT_IMPLEMENTED = 501,

    SERVER_ERROR_SERVICE_UNAVAILABLE =
            503,
    SUCCESS_ACCEPTED = 202,
    SUCCESS_CREATED = 201,
    SUCCESS_NO_CONTENT = 204,
    SUCCESS_OK = 200,

} StatusCode_t;
```

Media Types and Statuses (both copied from Restlet)

```
struct UserData_t
{
  char* queryString;
  uint16_t queryStringSize;
  uint8_t numOfAttrs;
  struct Attr_t
  {
    char* pattern;
    char* realValue;
  } attributes[ MAX_URL_MATCHED_ATTRS ];

  char* postData;
  uint16_t postDataSize;
};
```

User Data type

```
struct Header_t
{
  char* name;
  char* value;
};
```

Header type

```
struct Response_t
{
    StatusCode_t statusCode;
    char* statusString;
    const char* url;
    uint8_t numOfResponseHeaders;
    struct Header_t headers[ MAX_RESPONSE_HEADERS ];
    uint16_t responseBuffIndex;
    char responseBuffer[RESPONSE_BUFFER_SIZE];
};
```

Response type

```
typedef struct
{
    struct psock sin, sout; /*Protosockets for incoming and outgoing communication*/
    struct pt outputpt;
    char inputBuf[INCOMING_DATA_BUFF_SIZE]; /*to put incoming data in*/
    char* resourceUrl;
    uint8_t state;
    HttpMethod_t requestType; /* GET, POST, etc */

    struct Response_t response;

    char connBuffer[REQUEST_BUFFER_SIZE];
    uint16_t connBuffUsedSize;

    uint8_t numOfRequestHeaders;
    struct Header_t headers[ MAX_REQUEST_HEADERS ];

    struct UserData_t userData;
} ConnectionState_t;
```

This structure contains information about the HTTP request.

```
void http_common_init_connection(ConnectionState_t* pConnectionState);
```

Initializes the connection state by clearing out the data structures

---

A.3 HTTP-Server

```
PROCESS_NAME(httpdProcess);
```

Declare process

```
typedef bool (*ServiceCallback) (ConnectionState_t* pConnectionState);
```

Type definition of the service callback

```
void http_server_set_service_callback( ServiceCallback callback );
```

Setter of the service callback, this callback will be called in case of HTTP request.

```
bool http_server_get_query_variable(
    ConnectionState_t* pConnectionState,
    const char *pcName,
    char* pcOutput,
    uint16_t nOutputSize);
```

Returns query variable in the URL.
Returns true if the variable found, false otherwise.
Variable is put in the buffer provided.

```
bool http_server_get_post_variable(
    ConnectionState_t* pConnectionState,
    const char *pcName,
    char* pcOutput,
    uint16_t nOutputSize);
```

Returns variable in the Post Data.
Returns true if the variable found, false otherwise.
Variable is put in the buffer provided.

```
const char* http_server_get_req_header_value(
    ConnectionState_t* pConnectionState, const char* pcHeaderName );
```

Returns the value of the header name provided. Return NULL if header does not exist.

```
bool http_server_handle_req_header(const char* pcHeaderName);
```

Requests to save the header; only a number of headers will be saved
in buffer due to resource limitations.
Return true if the header will be saved, false otherwise.

```
bool http_server_add_res_header(
    ConnectionState_t* pConnectionState,
    const char* pcName,
    const char* pcValue,
    bool bCopyValue);
```

Adds the header name and value provided to the response.
Name of the header should be hardcoded since it is accessed from code segment
(not copied to buffer) whereas value of the header can be copied
depending on the relevant parameter. This is needed since some values may be
generated dynamically (ex: e-tag value)

```
HttpMethod_t http_server_get_http_method(ConnectionState_t* pConnectionState);
```

Getter method for the HTTP method (GET, POST, etc) of the request

```
void http_server_set_http_status(ConnectionState_t* pConnectionState, StatusCode_t status);
```

Setter for the status code (200, 201, etc) of the response.

```
char* http_server_put_in_conn_buffer(ConnectionState_t* pConnectionState, char* pcValue);
```

Puts the provided string in the connection buffer, used by a bunch of other functions
as well as rest module.

```
char* http_server_get_res_buf(ConnectionState_t* pConnectionState);
```

Return a pointer to the response buffer in case the user wants to have direct access.

```
Error_t http_server_copy_to_response(
      ConnectionState_t* pConnectionState, const char* pcBuffer, uint16_t nSize);
```
Copy the provided buffer contents to the response buffer.

```
Error_t http_server_concatenate_str_to_response(
      ConnectionState_t* pConnectionState, const char* pcBuffer);
```
Copies the provided string to the end of the response buffer.

```
const char* http_server_get_post_data(ConnectionState_t* pConnectionState);
```
Returns pointer to the Post Data buffer.

```
void http_server_get_base_url(char* pcOut);
```
Generates base url (ex: "http://172.16.79.0:8080") and copies it into the buffer provided.

```
const char* http_server_get_relative_url(ConnectionState_t* pConnectionState);
```
Returns the relative URL (ex: /temperature) of the resource accessed.

```
bool http_server_set_representation(
      ConnectionState_t* pConnectionState, MediaType_t mediaType);
```
Set the header "Content-Type" to the given media type.

A.4 REST

```
PROCESS_NAME(restWebServicesProcess);
```
Declare process

```
typedef void (*RestfulHandler) (ConnectionState_t* pConnectionState);
typedef bool (*RestfulPreHandler) (ConnectionState_t* pConnectionState);
typedef void (*RestfulPostHandler) (ConnectionState_t* pConnectionState);
```
Signature of handler functions/*

```
struct Resource_t
{
   struct Resource_t *next; /*points to next resource defined*/
   HttpMethod_t requestTypesToHandle; /*handled HTTP methods*/
   const char* pUrlPattern; /*simple template of handled URLs
                                    ex: "/task/{id}" id is parameterized*/
   RestfulHandler handler; /*handler function*/
   RestfulPreHandler preHandler; /*to be called before handler, may perform initializations*/
   RestfulPostHandler postHandler; /*to be called after handler, may perform finalizations (cleanup, etc)*/
   void* pUserData; /*pointer to user specific data*/
};
typedef struct Resource_t Resource_t;
```
Data structure representing a resource in REST.

```
#define RESOURCE(name,typesToHandle,url) \
void name##_handler(ConnectionState_t* pConnectionState); \
Resource_t resource_##name = { NULL, typesToHandle, url, name##_handler, NULL, NULL, NULL }
```

Macro to define a Resource
Resources are statically defined for the sake of efficiency and better memory management.

```
void rest_init(void);
```

Initializes REST framework and starts HTTP process

```
void rest_activate_resource(Resource_t* pResource);
```

Resources wanted to be accessible should be activated with the following code.

```
bool rest_invoke_restful_service( ConnectionState_t* pConnectionState);
```

To be called by HTTP server as a callback function when a new HTTP connection appears.
This function dispatches the corresponding RESTful service.

```
bool rest_set_url(ConnectionState_t* pConnectionState, const char* pcUrl);
```

Sets "Location" header

```
char* rest_get_attribute(ConnectionState_t* pConnectionState, const char* pcPattern);
```

Returns the value of the attribute mapped to the template URL.
Ex: Template URL "/task/{id}" matches "/task/5" and so calling this function with
"id" pattern will return "5".

```
void rest_set_user_data(Resource_t* pResource, void* pUserData);
```

Setter method for user specific data.

```
void* rest_get_user_data(Resource_t* pResource);
```

Getter method for user specific data.

```
void rest_set_pre_handler(Resource_t* pResource, RestfulPreHandler preHandler);
```

Sets the pre handler function of the Resource.
If set, this function will be called just before the original handler function.
Can be used to setup work before resource handling.

```
void rest_set_post_handler(Resource_t* pResource, RestfulPostHandler postHandler);
```

Sets the post handler function of the Resource.
If set, this function will be called just after the original handler function.
Can be used to do cleanup (deallocate memory, etc) after resource handling.

A.5 SIMPLEXML

```
typedef void *SimpleXmlParser;
```

The simple xml parser structure.
SimpleXmlParsers should be created and destroyed using the functions
simpleXmlCreateParser, simpleXmlDestoryParser.

```
typedef enum simple_xml_event {
  FINISH_TAG, ADD_ATTRIBUTE, FINISH_ATTRIBUTES, ADD_CONTENT, ADD_SUBTAG
} SimpleXmlEvent;
```

Enumeration describing the event types that are sent to an SimpleXmlHandler
by an SimpleXmlParser.
see #SimpleXmlTagHandler
see #SimpleXmlParser

```
typedef struct simplexml_value_buffer {
  /* buffer data */
  char* sBuffer;
  /* size of the buffer */
  long nSize;
  /* insert position in buffer */
  long nPosition;
} TSimpleXmlValueBuffer, *SimpleXmlValueBuffer;
```

Value buffer.
This structure resembles a string buffer that
grows automatically when inserting data.

```
struct TSimpleXmlValueBuffer;
enum WriteState {OPENED_TAG,CLOSED_TAG};
typedef struct
{
  SimpleXmlEvent state;
  TSimpleXmlValueBuffer xmlWriteBuffer;
} XmlWriter;
```

Added for xml generation

```
typedef void (*SimpleXmlTagHandler) (
  SimpleXmlParser parser,
  SimpleXmlEvent event,
  const char* uri,
  const char* szName,
  const char** attr
);
```

Callback function to handle simple xml events.
The SimpleXmlTagHandler is invoked by a SimpleXmlParser
whenever one of the following event types occur:

FINISH_TAG
indicates that parsing of this tag has finished, szName contains the tag name, szAttribute and szValue are NULL, the result of the handler is ignored.

ADD_ATTRIBUTE
indicates that an attribute for this tag has been parsed, szName contains the tag name, szAttribute the attribute name and szValue contains the attribute contents, the result of the handler is ignored.

FINISH_ATTRIBUTES,
indicates that parsing of attributes for this tag is finished, szName contains the tag name, szAttribute and szValue are NULL, the result of the handler is ignored.

ADD_CONTENT
indicates that content of this tag has been parsed and should be added, szName contains the tag name and szValue contains the data to add, szAttribute is NULL and the result of the handler is ignored.

ADD_SUBTAG
indicates that a subtag has been parsed, szName contains the name of the subtag read, szAttribute and szValue are NULL, the result of the handler should either be NULL to indicate that this subtag is not of interest and should be skipped a SimpleXmlTagHandler that is used for handling the subtag.

see #SimpleXmlEvent
see #SimpleXmlParser

```
extern SimpleXmlParser simpleXmlCreateParser (
   const char *sData, long nDataSize
);
```

Creates a new simple xml parser for the specified input data.
The input data may be parsed with simpleXmlParse and the parser returned by this function as parameter.
Note: The parser will not copy the input data or in any way modify it.
However any modifications of the input data in a callback handler while parsing will have an undefined result!
param sData the input data to parse (must no be NULL).
param nDataSize the size of the input data buffer (sData) to parse (must be greater than 0).
return the new simple xml parser or NULL if there is not enough memory or the input data specified cannot be parsed.

```
extern void simpleXmlDestroyParser (
   SimpleXmlParser parser
);
```

Destroys the specified simple xml parser.
param parser the parser to destroy (must have been created using simpleXmlCreateParser).

```
extern int simpleXmlInitializeParser (
   SimpleXmlParser parser, const char *sData, long nDataSize
);
```

Reinitializes the specified simple xml parser for parsing the specified input data.
param parser the parser to initialize.
param sData the input data to parse (must not be NULL).
param nDataSize the size of the input data buffer (sData) to parse (must be greater than 0).
return 0 if the parser could not be initialized, > 0 if the parser was initialized successfully and parsing may be started using simpleXmlParse.

```
int simpleXmlParse (SimpleXmlParser parser, SimpleXmlTagHandler handler);
```

Starts an initialized (or newly created) xml parser with the specified document tag handler.
Note: This function may only be called once after creation or initialization of a parser. To reuse the parser it has to be freshly

initialized (using simpleXmlInitializeParser) prior to calling the function again.
param parser the parser to start.
param handler the handler to use for the document tag.
return 0 **if** there was no error, and error code $>$ 0 **if** there was an error.

```
char* simpleXmlGetErrorDescription (SimpleXmlParser parser);
```

Returns a description of the error that occured during parsing.
param parser the parser **for** which to get the error description.
return an error description or NULL if there was no error during parsing.

```
long simpleXmlGetLineNumber (SimpleXmlParser parser);
```

*Returns the line number of the current input line that the parser has read.
In case of an error this method will return the line number on which the error was encountered after a call to simpleXmlParse.
If called from a handler during parsing this function will return the current line number.
If called after a successfull simpleXmlParse run this function will return the line number of the last line parsed in the xml data.*
return the current input line number of the parser or -1 if it is unknown.

```
#define SIMPLE_XML_USER_ERROR 1000
#define SIMPLE_XML_USER_ERROR_XML_NOT_VALID 1001
```

Minimum value for a user abort.
see #simpleXmlParseAbort

```
void simpleXmlParseAbort (SimpleXmlParser parser, int nErrorCode);
```

Causes the simple xml parser to abort parsing of the input data.
This method may only be called from a tag handler.
The active simpleXmlParse run will be aborted and the simpleXmlParse function will return with the specified error code.
param nErrorCode the error code with which to abort (the error code must be >= SIMPLE_XML_USER_ERROR **else** the abort request is ignored!)

```
void* simpleXmlGetUserData(SimpleXmlParser parser);
int simpleXmlSetUserData(SimpleXmlParser parser, void* pData);
```

Added for getting user data

```
const char* simpleXmlGetAttrUri(size_t nNumber, const char** attr);
const char* simpleXmlGetAttrName(size_t nNumber, const char** attr);
const char* simpleXmlGetAttrValue(size_t nNumber, const char** attr);
size_t simpleXmlGetNumOfAttrs(const char** attr);
```

Attribute handlers

```
void simpleXmlStartDocument(XmlWriter* xmlWriter, char* buffer, unsigned short size);
void simpleXmlStartElement(XmlWriter* xmlWriter, const char* ns, const char* name);
void simpleXmlAddAttribute(XmlWriter* xmlWriter, const char* ns                    ,
  const char* name, const char* value);
void simpleXmlCharacters(XmlWriter* xmlWriter, const char* value);
void simpleXmlEndElement(XmlWriter* xmlWriter, const char* ns, const char* name);
void simpleXmlEndDocument(XmlWriter* xmlWriter);
```

Added for xml generation

A.6 LOGGER

```
#ifdef LOG_ENABLED
```

Logging is enabled via LOG_ENABLED Macro

```
enum eLevel{L_NONE, L_ERR, L_INFO, L_DBG};
```

Debug levels specifying how much information will be printed

```
void logger_helper(unsigned char level, const char* func_name);
```

Helps the logger by printing log record number, log level etc.

```
#ifndef PERSISTENT_LOG
  #define basic_logger(...) printf(__VA_ARGS__)
#else /*PERSISTENT_LOG*/
  void persistent_logger(char* fmt, ...);
  #define basic_logger(...) persistent_logger(__VA_ARGS__)
#endif /*PERSISTENT_LOG
```

Defines functions depending on whether persistent or basic logger is wanted.

```
#define TOOLS_LOGGER(level,...) \
do \
{ \
  logger_helper(level,__func__); \
  basic_logger(__VA_ARGS__); \
} while(0)
```

Main Logger Macro.

__func__ added by C99 standard and it is not a macro though;

the preprocessor does not know the name of the current function.

```
#define LOG_ERR(...) TOOLS_LOGGER(L_ERR, __VA_ARGS__)

#if (LOG_ENABLED >= 2)
  #define LOG_INFO(...) TOOLS_LOGGER(L_INFO, __VA_ARGS__)
  #if (LOG_ENABLED >= 3)
    #define LOG_DBG(...) TOOLS_LOGGER(L_DBG, __VA_ARGS__)
  #else
    #define LOG_DBG(...)
  #endif
#else
  #define LOG_INFO(...)
  #define LOG_DBG(...)
#endif

#else /*LOG_ENABLED*/
  #define LOG_ERR(...)
  #define LOG_INFO(...)
  #define LOG_DBG(...)
#endif /*LOG_ENABLED*/
```

Logger macro definitions

A.7 SOAP

```
typedef struct
{
  char name[50];
  char urn[50];

  uint8_t numOfParams;
  #define MAX_PARAMS 8
  struct Param_t
  {
    char name[20];
    char value[20];
    char type[20];
  } params[MAX_PARAMS];

  char *action;
} SoapMethod_t;
```

Representation of a SOAP Method

```
typedef struct
{
   char* faultcode;
   char* faultstring;
   char* faultactor;
   char* detail;
} SoapFault_t;
```
Representation of a SOAP Fault

```
typedef struct
{
   SoapMethod_t method;
   SoapFault_t fault;
} SoapBody_t;
```
Representation of a SOAP Body

```
typedef struct
{
   SoapBody_t body;
   char* buffer;
} SoapEnvelope_t;
```
Representation of a SOAP Envelope

```
typedef struct
{
   SoapEnvelope_t env;
}SoapContext_t;
```
Representation of a SOAP Context

```
typedef void (*SoapHandler) (SoapMethod_t* request, SoapMethod_t* response);
```
Signature of handler function

```
typedef struct
{
   struct SoapWebService_t *next;
   const char* pUrl;
   const char* pUrn;
   const char* pMethodName;
   SoapHandler handler;
} SoapWebService_t;
```
SOAP Web Service Data Structure

```
#define SOAP_WEB_SERVICE(name,url,urn,method) \
void name##_handler(SoapMethod_t* request, SoapMethod_t* response); \
SoapWebService_t soapWebService_##name = { NULL, url, urn, method, name##_handler }
```
Macro to define a SOAP-Based Web Service

```
void soap_init(void);
```
Initializes SOAP library and starts HTTP process

```
void soap_activate_service( SoapWebService_t* pWebService );
```

Web services should be activated with the following code to be accessible

```
bool soap_invoke_service( ConnectionState_t* pConnectionState );
```

To be called by HTTP server as a callback function when a new HTTP connection appears. This function dispatches the corresponding Web service.

```
void soap_set_method_name(const char *pcUrn, const char *pcName, SoapMethod_t* pMethod);
```

Sets the name and urn of the method to be called over SOAP.

```
void soap_add_param(
      const char *pcName, const char *pcValue, const char *pcType, SoapMethod_t* pMethod);
```

Adds new parameter(name, value and type of it) in the SOAP Method.

Bibliography

[1] csoap web site. http://csoap.sourceforge.net/. Accessed on June 2, 2009.

[2] Curl web site. http://curl.haxx.se/. Accessed on May 8, 2009.

[3] Efficient xml interchange working group (exiwg). http://www.w3.org/XML/EXI/. Accessed on May 8, 2009.

[4] Exificient web site. http://freshmeat.net/projects/exificient. Accessed on May 31, 2009.

[5] Ipso alliance. http://www.ipso-alliance.org. Accessed on June 2, 2009.

[6] Restlet web site. http://www.restlet.org/. Accessed on June 7, 2009.

[7] Simple xml parser web site. http://simplexml.sourceforge.net/. Accessed on May 31, 2009.

[8] Christopher J. Augeri, Dursun A. Bulutoglu, Barry E. Mullins, Rusty O. Baldwin, and Leemon C. Baird, III. An analysis of xml compression efficiency. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 7, New York, NY, USA, 2007. ACM.

[9] H. Baldus, K. Klabunde, and G. Muesch. Reliable set-up of medical body-sensor networks. In *EWSN'04: Proceedings of the First EuropeanWorkshop on Wireless Sensor Networks*, Berlin, Germany, 2004.

[10] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax. RFC 3986, Internet Engineering Task Force, January 2005.

[11] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Iona Michael Champion, Chris Ferris, and David Orchard. Web services architecture. May 2004.

[12] Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. Extensible markup language (xml) 1.0. World Wide Web Consortium, Recommendation REC-xml-19980210, February 1998.

[13] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, New York, NY, USA, 2006. ACM Press.

[14] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. World Wide Web Consortium, Recommendation REC-wsdl20-20070626, June 2007.

[15] Douglas Crockford. The application/json media type for javascript object notation (json). Internet RFC 4627, July 2006.

[16] R. Dickerson, J. Lu, J. Lu, and K. Whitehouse. Stream feeds - an abstraction for the world wide sensor web. In *IOT '08: Proceedings of of the Internet of Things Conference)*, 2008.

[17] Vlad Trifa Dominique Guinard. Towards the web of things: Web mashups for embedded devices. In *MEM '09: Proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, 2009.

[18] Adam Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003.

[19] Adam Dunkels, Björn Grönvall, 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.

[20] Adam Dunkels, Fredrik Österlind, and Zhitao He. An adaptive communication architecture for wireless sensor networks. In *Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*, Sydney, Australia, November 2007.

[21] Adam Dunkels, Fredrik Österlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV)*, Cork, Ireland, June 2007.

[22] Adam Dunkels and Jean-Philippe Vasseur. IP for Smart Objects, September 2008. IPSO Alliance White Paper 1.

[23] 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.

[24] L. Dusseault and J. Snell. Patch method for htt. Internet draft, Internet Engineering Task Force, 2009.

[25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[26] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000.

[27] Jonathan Hui Gabriel Montenegro, Nandakishore Kushalnagar and David Culler. Transmission of ipv6 packets over ieee 802.15.4 networks (rfc 4944). http://tools.ietf.org/html/rfc4944.

[28] Roch Glitho, Ferhat Khendek, Nuru Yakub Othman, and Samir Chebbine. Web services-based architecture for the interactions between end-user applications and sink-less wireless sensor networks. pages 865–869, Jan. 2007.

[29] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[30] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107, New York, NY, USA, 2002. ACM.

[31] Aman Kansal, Suman Nath, Jie Liu, and Feng Zhao. Senseweb: An infrastructure for shared sensing. *IEEE MultiMedia*, 14(4):8–13, 2007.

[32] Lynette Laffea, Russ Monson, Richard Han, Ryan Manning, Ashly Glasser, Steve Oncley, Jielun Sun, Sean Burns, Steve Semmer, and John Militzer. Comprehensive monitoring of co2 sequestration in subalpine forest ecosystems and its relation to global warming. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 423–424, New York, NY, USA, 2006. ACM.

[33] Greg Leighton. Xml compression bibliography. http://www.cs.ualberta.ca/ gleighto/research/xml-comp.html. Accessed on May 8, 2009.

[34] 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.

[35] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of*

the 1st ACM international workshop on Wireless sensor networks and applications, pages 88–97, New York, NY, USA, 2002. ACM.

[36] Bruce Martin and Bashar Jano. Wap binary xml content format. http://www.w3.org/TR/wbxml/.

[37] Nilo Mitra. Soap version 1.2 part 0: Primer. World Wide Web Consortium, Recommendation REC-soap12-part0-20030624, June 2003.

[38] Jeffrey Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and D. Hellerstein. Delta encoding in http. RFC 3229, January 2002.

[39] Luca Mottola. *Programming Wireless Sensor Networks: From Physical to Logical Neighborhoods*. PhD thesis, 2008.

[40] N.Y. Othman, R.H. Glitho, and F. Khendek. The design and implementation of a web service framework for individual nodes in sinkless wireless sensor networks. pages 941–947, July 2007.

[41] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. ”big”’ web services: making the right architectural decision. In *WWW ’08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[42] Joseph Polastre, Robert Szewczyk, and David E. Culler. Telos: enabling ultra-low power wireless research. In *IPSN*, pages 364–369, 2005.

[43] J. Postel. Transmission control protocol. Rfc, Internet Engineering Task Force, Sep 1981.

[44] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In Tarek F. Abdelzaher, Margaret Martonosi, and Adam Wolisz, editors, *SenSys*, pages 253–266. ACM, 2008.

[45] Razvan, Chieh, and Andreas Terzis. Koala: Ultra-low power data retrieval in wireless sensor networks. In *IPSN ’08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 421–432, Washington, DC, USA, 2008. IEEE Computer Society.

[46] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O’Reilly Media, Inc., May 2007.

[47] J. Romkey. Serial line internet protocol. http://tools.ietf.org/html/rfc1055.

[48] P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core, October 2004.

[49] Vipul Singhvi, Andreas Krause, Carlos Guestrin, James H. Garrett, Jr., and H. Scott Matthews. Intelligent light control using sensor networks. In *SenSys ’05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 218–229, New York, NY, USA, 2005. ACM.

[50] J. Snell. Prefer header for http. Internet draft, Internet Engineering Task Force, February 2008.

[51] J. Snell. Http multipart batched request format. Internet draft, Internet Engineering Task Force, 2009.

[52] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, San Francisco, USA, April 2009.

[53] Chris Ullman and Lucinda Dykes. *Beginning Ajax (Programmer to Programmer)*. Wrox, March 2007.

[54] Megan Wachs, Jung Il Choi, Jung Woo Lee, Kannan Srinivasan, Zhe Chen, Mayank Jain, and Philip Levis. Visibility: a new metric for protocol design. In *SenSys ’07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 73–86, New York, NY, USA, 2007. ACM.

[55] Erik Wilde. Putting things to rest. Technical Report 2007-015, November 2007.

[56] Jeffrey Wong and Jason Hong. What do we ”mashup” when we make mashups? In *WEUSE ’08: Proceedings of the 4th international workshop on End-user software engineering*, pages 35–39, New York, NY, USA, 2008. ACM.