Master Thesis

Design and implementation of a gateway for Web-based interaction and management of embedded devices

Author(s):
Wieland, Samuel

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005797101

Rights / License:
In Copyright - Non-Commercial Use Permitted
Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices

Master’s Thesis

Author : Samuel Wieland
samuel.wieland@gmail.com

Supervisor : Vlad Trifa
Professor : Friedemann Mattern

Institute for Pervasive Computing
Department of Computer Science
ETH Zürich

March 2009
Abstract

Embedding and integrating wireless sensor networks into the Web is ongoing research with many problems to be resolved. Although the Web by design provides scalability, extensibility and evolvability, we cannot directly apply these features to legacy sensor networks. Sensors and embedded devices have constrained resources in terms of CPU, memory and computational power, possibly interact in mobile environments, and most of the times they are not Web-enabled. Many solutions have been proposed and existing middlewares come up with promising ideas and concepts. Despite that fact, we think that no current solution properly integrates with the design principles of the Web. We therefore present the smart gateway - A framework that simplifies the integration of sensors and wireless sensor networks into the Web. The smart gateway helps rapid development and deployment of Web-mash-ups from existing sensor resources.

The smart gateway represents physical devices with existing Web technology, giving access and control to low-level devices to anyone through the Web browser. By implementing a device abstraction that allows to integrate any kind of device, we enrich the Web with first class citizens capable to sense and interact with the real world. The smart gateway therefore provides the first building blocks to empower users to create the next generation of the World Wide Web, namely the vision of the Web of Things.
# Table of contents

1 Introduction ................................................. 1
   1.1 Goals .................................................. 3
   1.2 Contributions and Outline ............................ 3

2 Infrastructures for the Web of Things ............... 5
   2.1 Services and Service Oriented Architecture .......... 5
      2.1.1 SOA .............................................. 6
      2.1.2 SOAP ............................................ 6
      2.1.3 WSDL ............................................ 7
   2.2 Representational State Transfer (REST) ............. 7
      2.2.1 HTTP ............................................ 9
   2.3 Related Work .......................................... 9
      2.3.1 RPC-Based Middlewares .......................... 9
      2.3.2 Web-Based Interaction ............................ 11
   2.4 Summary ............................................... 12

3 Technologies ............................................... 13
   3.1 Multipurpose Internet Mail Extensions .............. 13
   3.2 Data formats .......................................... 14
      3.2.1 XML ............................................. 14
      3.2.2 JSON ............................................ 15
      3.2.3 Protocol Buffers ................................ 15
      3.2.4 Summary .......................................... 16
   3.3 Events and Messaging ................................ 17
      3.3.1 Event notification ................................. 17
      3.3.2 XMPP ............................................. 17
      3.3.3 MQTT .............................................. 18
      3.3.4 JMS ............................................... 18
      3.3.5 OSC ............................................... 18
      3.3.6 Summary .......................................... 19

4 System design ............................................... 21
   4.1 Core .................................................. 21
      4.1.1 Configuration ..................................... 22
      4.1.2 Structure Maintenance ........................... 23
      4.1.3 System timers .................................... 26
Chapter 1

Introduction

"The pure and simple truth is rarely pure and never simple."

Oscar Wilde (1854-1900)

In the last fifty years, computers have gradually become the driving forces of our everyday life. Thanks to the technological progress, more computational power is provided by smaller integrated circuits, equipped with sensing and networking capabilities, everywhere, at all time and some of them even without attached power source. The progress in the manufacturing of computer hardware (especially in miniaturization) allows us to equip almost everything with computing capabilities.

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. - Mark Weiser (1988)"

Following Weisers vision, many appliances for ubiquitous computing have been proposed and implemented. The smart fridge, a fridge that monitors your food and automatically (re)orders items that are out of stock, a cleaner that automatically knows how to wash your clothes because the jeans inform about the washing-temperature, a mirror displaying you a decent clothing computed from the clothes available in your closet, are only some example scenarios that can be formulated with such technologies.

All those appliances have in common, that they require fundamental shifts in computing paradigms. Today we usually use a one-fits-it-all approach where one central computer orchestrates peripheral units by computing all the pending decisions centrally. This is comparable to the swiss army knife that is capable to perform all kinds of tasks. The central component needs to be generic in order to support all the different requirements, leading to a sub-optimal overall performance. Radically different is the one-for-each approach, where we equip all items with a processing unit capable to compute local decisions. Such everyday life objects are augmented
to smart items, capable to sense and experience their environment, and able to solve problems collaboratively.

The integration of ubiquitous, pervasive smart items into a network of autonomous things, is generally referred to as the Internet of Things. It tries to address the technical problems that need to be solved (Addressability, Connectivity, Security, Privacy, Context awareness, etc.), to provide the basic building blocks for a fully distributed smart space with autonomous smart devices.

Unfortunately, we are still far from the goal of a cosmos of devices. The lack of generally agreed standards makes interaction between different smart objects difficult. As a consequence, many different middlewares that attempt to hide away complexity and heterogeneity of the underlying devices and networking technologies have been proposed [36]. Despite those efforts, many middlewares still use proprietary protocols, rendering solutions from different vendors incompatible. In addition, many applications suffer in terms of scalability and evolvability due to a tight coupling and high complexity.

Recently, researchers try to follow a new approach to inter-connected devices. Inspired by the success of the World Wide Web in the distribution of hyper-text media, sensing devices are deployed directly on the Web instead of wrapping them behind a proprietary middleware.

The Web is an information system originally invented to exchange hyper-text documents between researchers. Its simplicity and openness soon attracted a vast community and so todays Web provides billions of Web pages hosted by millions of computers all around the globe. Early adopted standards by W3C or IETF help to exchange information between different communication partners following well defined protocols. Standardized machine readable documents provide the necessary support for automation, enabling electronic commerce.

Since the birth of the Web twenty years ago, it has grown to enormous dimensions, but still it does not fail and an ending of the growth is not predictable. The loose coupling between communication partners, the support for load balancing mechanisms, caching and inter-operability seem to allow the Web to expand even further. A corner stone thereby, is its ability to integrate heterogeneous applications, ranging from traditional Web-sites to interactive social networking applications.

The Web of Things is a vision that tries to populate the Web with sensing devices and other physical devices, building a huge ecosystem of freely available and (re)usable sensors. (Existing) applications on the Web can consume and (re)use sensor-input the same way as they do with other Web-resources. Programmers (and even non-experienced users) don’t need to learn new programming concepts from scratch - they simply patch different Web-resources together - and augment the Web with a completely new dynamic dimension. Devices integrated into the Web inherit many of the great properties that made the Web so powerful. In addition, search engines can index the sensors and proxy servers cache sensor values transparently.

A central question however remains: how do we present devices to the Web that are themselves not Web-enabled? Although there exist many architectural proposals already (Chapter 2), we motivate and implement a new middleware for the integration of sensor networks into the Web - the smart gateway.
1.1 Goals

The goal of this thesis is to design, develop, implement and evaluate an embedded gateway architecture that facilitates the development of applications that integrate information from real-world physical devices into other Web-based applications.

The gateway is a stand-alone application that can be run on embedded computers with restrictions on memory and computational power. We will develop a generic template that helps users to (re)use and integrate existing infrastructure and at the same time has the ability to adapt to new technologies. This includes the development of an interface that allows to integrate different communication mechanisms like Bluetooth or ZigBee transparently, facilitating the integration of Wireless Sensor Networks (WSN). A centralized component will store and route data between physical devices and higher level applications.

The gateway allows devices to register and leave at all time and therefore taking into account that many devices are mobile. Registered devices will have the capability to expose their interfaces (how they can be used, configured and accessed) to the Web. Applications running on top of the gateway infrastructure can use this information to build mash-ups of devices - eg. combining different sensors into new, high-level sensors or even generate business events for enterprise ERP systems. To simplify the use of device-level events, we will evaluate different event distribution mechanisms, and will motivate a flexible implementation that allows clients to register their interest into events triggered by devices.

To further simplify access to devices and gateways through the Web, we will propose a simple yet powerful hierarchical grouping and addressing schema.

1.2 Contributions and Outline

Together with Vlad Trifa and Hänni Michael [19] we developed the theoretical basis for the smart gateway infrastructure addressing different aspects for the integration of physical embedded devices into the Web.

The gateway masks away a lot of the difficulties that arise when integrating physical sensors into a larger applications. By using a proper level of abstraction which allows experienced users to manage and use devices in almost any possible way, and at the same time keeping things simple and self-explaining, the gateway aims at enabling devices for anyone through the Web. For WSN providers this opens the unique opportunity to present the services of their WSN to a much broader audience, namely the world wide Web.

Devices that don’t possess the capability to interact with the Web directly, can be Web-enabled using the gateway. They inherit automatically many additional features that allow other citizens from the Web to search and address them regardless of the actual communication between the gateway and the device. In addition, devices can be augmented with a configuration API that allows the configuration and management from the Web using standard Web browsing technology. By introducing additional levels of abstraction, device level events can be handled by Web citizens like any other Web response, therefore greatly reducing complexity and at the same time enhance scalability and reuse.
Aside the theoretical parts, my work focused the implementation and evaluation of the proposed architecture. This thesis is therefore structured into a theoretical and a practical part. The theory gives an overall introduction to the related work (Chapter 2) and technologies (Chapter 3). The practical part explains the architecture (Chapter 4), presents several performance evaluations (Chapter 6) and gives an overview to the different prototypes we built using the gateway (Chapter 5). To complete, the further steps to be taken and my overall conclusion are presented at the end (Chapter 7).
"It would appear that we have reached the limits of what is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years."

John von Neumann (1903-1957)

Historically grown, there are two different architectural styles to build distributed systems for the Internet and the Web, namely WS-* and REST. A good introduction to WS-* can be found in [2] whereas [30] gives an overview to REST. In [26] an objective comparison has been performed between the two.

This chapter shall give a short overview to both styles. We present the technology and a selection of middlewares built with WS-* or REST.

2.1 Services and Service Oriented Architecture

In the early days of computer science, programs were developed to run on single machines. This approach was perfectly well, as long as computers could be seen as isolated islands without communication in between. With the development of the ARPANET, data could be transferred more conveniently from one computer to another, using network infrastructures. The next logical step was to abstract from inter-connecting computers to computers, towards connecting and distributing software. First approaches to inter-link software (RPC\(^1\), CORBA\(^2\), DCOM\(^3\), RMI\(^4\)) proposed the idea of wrapping a software component behind an interface. The software

\(^1\)http://en.wikipedia.org/wiki/Remote_procedure_call

\(^2\)http://en.wikipedia.org/wiki/Corba

\(^3\)http://en.wikipedia.org/wiki/Distributed_Component_Object_Model

\(^4\)http://en.wikipedia.org/wiki/Java_remote_method_invocation
behind the interface is called the service. The interface specifies the methods that can be called
by a remote client and at the same time provides a mechanism to mask away the fact, that
the service runs on a remote machine. Client and service both have to compile the interface
into service stubs (handling the communication issues) and then connect the stubs to the ap-
lication logic. Thanks to the stubs, performing a remote procedure call through the interface
should be indistinguishable from calling the procedure locally.

\( \text{WS-}^* \) stands for a set of specifications, developed for the Web services stack (Section 2.1.2).
The goal is, to provide a composable (stackable) infrastructure for the delivery of data from
one application to another. Two communication partners agree on quality of service (QOS)
for the exchange of information (eg. security, reliability, accountability or persistence). The
different QOS are performed by dedicated modules plugged (dynamically) into the \text{WS}-stack.

### 2.1.1 SOA

The focus of Service Oriented Architectures (SOA) is integration of different services into
bigger business applications (Business to business integration or large scale enterprise comput-
ing). Letting the infrastructure automatically decide on integration decisions, the protocol to
use, the routing and distribution or the data transformations, helps to build heavy enterprise
applications with as little user intervention possible. SOA can therefore be seen as a paradigm
how well known services running in a distributed setup can be inter-connected and orchestrated.

Services within SOA can be discovered and connected automatically using UDDI (a cen-
tral lookup and description directory for services), with \( \text{WS-}^* \) taking care of the information
exchange. This infrastructure machinery has proven great success when connecting big servers
with enough computational power and broadband network connections.

### 2.1.2 SOAP

SOAP [17] is an XML language defining the structure and the format of SOAP messages
(\textit{envelope}). A SOAP message consists of a header (containing information for QOS) and a
body (containing the payload). SOAP messages are the basic building blocks of the \text{WS}-stack.

Originally SOAP was conceived as the minimal possible infrastructure necessary to perform
RPC through the Internet, however SOAP messages can be delivered via any kind of transport
medium (via USB-stick or via the Internet through HTTP).

To translate a RPC into a SOAP document, the call is first marshalled (serialized) into
XML (eg. using JAXB\(^5\)). Before storing the XML into the SOAP body the \( \text{WS-}^* \) -stack allows
a series of transformations to be applied to the payload, ranging from encryption to reliability
contracts. Figure 2.1 illustrates the process graphically.

Although SOAP is capable to use different transport protocols, today most implementations
use HTTP as transport protocol (and there only the \texttt{POST} capability of the protocol). The
serialization of all the RPCs into XML is resource consuming in terms of CPU and memory
(even if there exists dedicated hardware for SOAP already).

\(^5\)http://jaxb.dev.java.net/
2.2. Representational State Transfer (REST)

![Diagram of Application 1 calling an RPC on its local SOAP stub](image)

**Figure 2.1:** Application 1 calls an RPC on its local SOAP stub. The stub serializes the call into an XML format and delivers the XML via any transport protocol to the SOAP stub of application 2. The remote stub deserializes the XML and calls the method locally. The result of the method call is serialized the same way in XML and gets delivered back to application 1.

### 2.1.3 WSDL

The Web Service Description Language (WSDL) [7] is a XML language, that allows to describe service interfaces in a highly standardized format. Operations (with their in and out parameters), communication endpoints (porttype) and the transport protocol to use (binding) are defined abstract, allowing different interface compilers (eg. AXIS\(^6\), CXF\(^7\)) to generate service stubs that are programming language, operating system and transport protocol independent (SOAP, JMS, MQ, etc.).

The focus of WS-* has always been on integrating existing legacy applications. For existing programs a WSDL interface can be written, and the code in the service stubs defines the bridge between the stubs and the legacy program. For client and service, the WSDL formulates a contract where both communication partners can rely on.

The static structure of a WSDL is paradoxically one of its weakest points. Whenever a service provider changes its interface, all its clients have to recompile their stubs as well (even if the changed functionality is not used at all). In a mobile environment this is hard. For example: what if the WSDL-interface has been implemented in hardware to speedup an embedded sensor? This drawback makes WSDL hard to use if you have to integrate physical devices where the interface is not known in advance, or where the interface needs to evolve over time.

An other problem is the vague specification for the format of in and out parameters for operations defined in the WSDL. Assume the service provider encodes character strings in UTF-8, whereas the service consumer relies on ASCII. These two implementations, using exactly the same interface, will crash at runtime.

### 2.2 Representational State Transfer (REST)

In his doctoral thesis [11] Roy Fielding proposed several guidelines how distributed, loosely coupled architectures should be built, motivated at the example of the success of the World Wide Web. The basic principles state that:

**Identification of resources with URI.** Each resource can be uniquely identified and addressed. A URI [3] is comparable to a postal address or a social security number.

\(^6\)http://ws.apache.org/axis/
\(^7\)http://cxf.apache.org/
Stateless communication between resources. Two interacting resources do not maintain a local state (e.g., cookie) during a communication session. A call to a URI has to encode all the information needed to compute the state from the request. This equals to a state machine that is exchanged and altered between two communication partners.

Self-descriptive resource representations. Information exchanged between resources is represented as resource representations. Representations can have different formats (e.g., XML, HTML, etc.) described with Meta-data.

Exposure through a uniform interface. Similarly to the database operations CRUD (CREATE, READ, UPDATE, DELETE), resources are modified with a set of operations (PUT, POST, GET, DELETE), each with a well-defined task. GET to retrieve a representation, PUT to generate a new resource, POST to change the state, and DELETE to remove a resource. GET should always be idempotent.

Although REST formulates a generic template that can be applied to any application, today REST is most often used in combination with HTTP. Using HTTP as example of a REST architecture, the strength and weakness of RESTful applications will shortly be discussed.

Thanks to the statelessness of HTTP, intermediate proxy servers can cache, load balance or cluster requests from many clients resulting in great scalability and flexibility - as proved by the Web with its billions of citizens. The caching ability is interesting especially when dealing with resource constrained devices - if the cache can deliver the result without real communication to the device, resources can be saved efficiently and transparently.

The idea of resource representations is well established for humans, making it easy for them to use and adopt the concepts. It seems to be a natural way of abstraction that the client never modifies the resource itself but uses a representation as a placeholder to perform operations on (think about how natural it is to withdraw money from a bank account via credit card).

Applications built with RESTful principles in mind can be easily inter-connected using URI, and in some cases, their interfaces can be extended without breaking existing clients. This helps integration and reuse. The drawback lies in the details: first, as opposed to WSDL, client and server don’t have an explicit interface contract where the client can rely on at runtime. Therefore clients usually need to take special care to catch unexpected interface changes. Second, legacy applications without existing Web interfaces are hard to be integrated into the Web, leading to typical workarounds with CGI-scripts greatly reducing overall performance and scalability.

Currently there is no Web browser available that supports all the four REST operations (typically browsers support only GET and POST). So most RESTful applications can rely on only two operations reducing expressive power and flexibility. With the growing number of RESTful applications on top of the Web, future generations of browsers therefore will hopefully support all four REST operations. For now, fortunately there exists a plugin for Mozilla browsers, adding the missing REST support (Appendix C.2).

---

The REST principles led to the idea of Resource Oriented Architectures (ROA) [30]. ROA carefully applies the REST principles to the whole design process and the development of the application. Additionally to the basic rules, two additional guidelines help to build consistent and reusable interfaces.

Addressability states, that every interesting resource should be accessible through exactly one globally unique identifier. The identifier can be stored, transmitted via e.g. email, bookmarked and indexed by search engines. This implicitly contains the idea of an application with many entry-points through URI (basically each resource acts as an entry-point).

Connectedness requires resources to be linked to as many other resources possible, leading to a well-connected and easy browseable mesh of hypertext documents suitable for human interaction.

2.2.1 HTTP

The Hypertext Transfer Protocol (HTTP) [14] was originally developed to transfer documents from one computer in the Internet to another. HTTP represents a typical stateless request-reply protocol where the client initiates a call to a server by requesting a resource and the server replying with a copy of the information.

Since version 1.0 HTTP provides 8 different request methods that suffice to create, update and delete resources - namely: GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS and CONNECT.

A key feature of the HTTP protocol is its capability to negotiate the format of the information exchanged between client and server using meta-data. The client indicates one or several desired format(s) as MIME types (Section 3.1) together with language preferences, and the server will try to answer the request by finding the resource that matches the request the best.

With libraries implemented in almost every programming language and servers running on tiny embedded computers, HTTP is widely spread and thanks to its openness (open standards for MIME, URI, XML etc by W3C or IETF) the barrier to use HTTP is low.

2.3 Related Work

We present a few middlewares that aim at integrating sensors and/or sensor networks into applications in order to facilitate the use and programming of mobile devices. We show two different styles with representatives for each - RPC-based middlewares (Section 2.3.1) and Web-based middlewares (Section 2.3.2).

2.3.1 RPC-Based Middlewares

There have been many proposals how to integrate WSN into the Internet and especially how to make the data available for clients from the Web.

The Global Sensor Network (GSN) [1, 32] is a middleware developed at the Swiss Federal School of Technology Lausanne (EPFL). GSN allows to attach sensors from different WSN in a uniform way and gives access from the Internet to the sensors. Each sensor is represented as a virtual sensor that can be either just a software component (e.g. an aggregate from different
Infrastructures for the Web of Things

sensors) or a real sensor. Data streams from sensors are modeled as sequences of data tuples with a time-stamp. The middleware acts as a sink for all the streams in the WSN where all the data tuples get stored into a centralized database (Figure 2.2). From the Web this database can be accessed and queried. Different GSN instances can exchange data through XML-RPC\(^9\) over HTTP.

![Figure 2.2: WSN on the left sending data-streams to the GSN middleware. The middleware acts as link between WSN and Internet.](image)

The SenseWeb [31, 29] project is developed at Microsoft research. At the core of the middleware lies a central coordinator that maintains references to all the WSN registered. The coordinator answers queries from external clients. Sensor data is cached at the coordinator, so that the WSN has not to be polled upon recurring client requests. Each WSN is represented as a sense-gateway acting as an interface between coordinator and WSN. Coordinator and sense-gateway are both Web-services accessible through SOAP.

The CoolTown project [9] describes the ”Web Presence Manager” (WPM) that provides Web representations for physical entities (devices). With the help of these representations, different physical devices can interact and provide an extensive context-enhanced media-oriented platform. XML and SOAP is used for the communication and WSDL and UDDI for the discovery and device advertisements. Devices periodically broadcast a XML-beacon containing configuration information. Devices interested extract a Web-page URI from the XML, leading to a XML page with the device information.

IrisNet [5] provides a decentralized database for sensor data. Sensing services submit data values for later query and retrieval, but allow real-time streaming together with real-time data reduction (eg. compression or feature extraction).

RPC-based applications heavily rely on tight interface contracts (eg. WSDL), increasing coupling and complexity (binding devices strongly to a specific application scenario). Connecting incompatible middlewares requires expensive translation and transformation of data and makes it impossible to transfer a sensor from one application to another.

Accessing a sensors via the middleware requires additional software. This makes it hard to get started for non-experienced users.

\(^{9}\)http://xmlrpc.com
2.3. Related Work

2.3.2 Web-Based Interaction

Instead of wrapping a WSN behind a service, Web-based solutions try to provide an interface towards the Web, that can be accessed and browsed with a Web browser (eg. via HTML).

SensorBase.org [6] is a centralized storage and search middleware for sensor data developed at the University of California Los Angeles (UCLA). WSN send data in plain format (eg. as csv-file) to the central component, that acts as publisher of the data. The data is stored into a database that can be accessed for search and data retrieval from the Web through a graphical user interface in the Web browser. Through RSS feeds, changes in sensor data can be distributed asynchronously in a publish/subscribe paradigm. Additionally, the middleware can be accessed through SOAP as a traditional Web service.

Pachube\textsuperscript{10} follows a similar approach. Sensors push their data to the centralized Pachube infrastructure. Clients from the Web can either poll the sensors, or subscribe to an RSS or Atom feed representing the sensor, and responses can be served in different formats (EEML, JSON, CSV). The RESTful user interface allows search and localization of the sensors (also graphically).

The data-centric approach with a central repository for storage and retrieval of sensor data, suits well for data collection and evaluation scenarios. SensorBase was developed especially to centralize measurements from many sensors all over the campus, helping researchers to find and share historical sensor data. However, the scalability and evolvability of a centralized architecture to Web-size is questionable.

Sensors are completely abstracted away, and all that is left are text documents representing the sensors values to certain times. Interaction and management with the physical device is not addressed at all.

According to [37], physical ”items” (eg. sensors) should be integrated directly into the Web as resources, following the design principles of the Web (eg. REST principles). Sensors should be uniquely addressable via URI (comparable to hypertext documents), and clients that want to communicate to a sensor, can use existing Web-browsing technology, lowering the overall requirements.

TinyREST [22] proposes a gateway, that directly embeds devices as resources into the Web. The RESTful interface allows clients to send POST and GET requests directly to the devices via URI (eg http://gwIP/sensori or http://gwIP/sensori/light). In addition a third command SUBSCRIBE is used for the event management, that allows publish/subscribe between the sensors and the Web client.

TinyREST is a promising idea with the big picture fitting our view. Sadly, the violation of the REST principle, by introducing new operators for the event management renders the solution incompatible to other RESTful applications.

Reuse and composition of documents and streams on the Web have recently gained broad attention. pREST [10] describes a protocol how resources can be interlinked with REST principles in mind. Everything in pREST is modeled as resource (primitive resources for atomic items such as strings, and complex resources for items containing nested resources). Producers and consumers that are pREST-enabled can be linked together by so called endpoints. These can be either producers or consumers of sensor-data (eg. camera sending an image to the Web-

\textsuperscript{10}http://www.pachube.com/
server that upon receipt publishes the image). Yahoo pipes\textsuperscript{11} allows users to aggregate and mix RSS-streams and Web-pages into new documents. SUNs Zembly\textsuperscript{12} helps to create widgets and social applications for social platforms. This ”mashupability” of Web items motivates us to integrate physical sensors as first-class citizens onto the Web, augmenting mash-ups with real sensor input.

2.4 Summary

With REST and WS-* we presented two architectural styles for distributed architectures. We try to summarize both styles and give a short judgement.

The Web-service approach tries to move computation from the client into the infrastructure with the infrastructure taking care of the different QOS. This reduces client code and omits code duplication. For enterprise computing scenarios (business to business) with sufficient computational power, WS-* provides powerful tools helping integration and reuse. The structured interface contract between consumer and producer (WSDL) helps building reliable software and allows to mask any kind of application behind the interface.

The RPC-based communication requires complex XML transformations for each request/response consuming a lot of CPU power, memory and battery - For embedded devices these are valuable resources that need to be spent with care.

In REST, items (physical or virtual) are modeled as resources, presented to the client as resource representations that can be chosen at runtime. The computation is shifted away from a centralized infrastructure into a distributed, loosely coupled network of resources (network centric computation). Standards from the Web ensure compatibility and allow to inter-connect different applications via URI.

The missing feature of an interface clearly specifying a resources API complicates integration of applications (additional exception handling has to be performed by the application to catch unexpected interface errors). The Web Application Description Language (WADL)\textsuperscript{23} tries to bridge this gap, however there is no full consensus on the format and the structure of the interface yet.

The experience gained by studying other middlewares (Section 2.3) let us conclude, that the integration of devices into the Web (as resources) offers the ideal substrate to build the Web of Things. For our implementation we propose a careful design following the REST principles.

\textsuperscript{11}http://pipes.yahoo.com
\textsuperscript{12}http://zembly.com/
In the last decade the growth of the Web has lead to a series of technologies. Many difficulties have been addressed and solved successfully, many others remain. This chapter tries to give an overview to existing technology and motivates why we chose one or another.

3.1 Multipurpose Internet Mail Extensions

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

MIME categorizes format and encoding into types called Internet Media Type or simply MIME type. The Internet Assigned Numbers Authority (IANA) maintains a list of all the official MIME types\(^1\) (applications are allowed to define their own types). Types contain subtypes to further organize and classify. A MIME type for an image encoded as png is expressed with image/png, an audio in mp3 with audio/mp3. Within a MIME document, the MIME type is specified with the keyword Content-type.

Documents that use MIME have a header and a body. The body contains the payload (the encoded document(s)) and the header maintains instructions how the payload has to be interpreted. Listing 3.1 shows a sample document encoded as MIME.

\(^1\)http://www.iana.org/assignments/media-types/
A MIME message containing two documents. One as XML, one as plain text.

This document is encoded as plain text.

Listing 3.1: MIME encoded document

3.2 Data formats

Exchange data between two programs or transmitting from one computer to another over a network has always been an interesting research area. Many data formats have been proposed all aiming at the optimization of different aspects (speed, overhead, genericity). We pick out a few and try to show their properties.

3.2.1 XML

Extensible Markup language (XML) [4] provides a simple way how data can be represented in textual form where meta-tags enclose the data item (Listing 3.2). Together with an XSL style-sheet that acts as a grammar describing how to read the XML, almost any kind of data can be encoded. XML is therefore widely used in serialization processes (Section 2.1.2).

Listing 3.2: Example XML with a list of friends.
3.2.2 JSON

JavaScript Object Notation (JSON) [16] is a data format that claims to be human readable and at the same time easy to parse for machines. Opposed to XML, JSON tries to remove as many meta-tags to minimize the amount of characters used for the serialization of an object or data.

Although everything in JSON is represented as a JObject there are two basic subtypes/structures to distinguish:

- Key/Value pairs: A chunk of data is stored into the JSON data structure together with a lookup key (Comparable to a hash-map).
- Lists: A series of data values stored comma-separated within "[" and "]".

Listing 3.3 shows a JSON object holding a list of two JSON objects.

```
{
  "friends": [
    {
      "firstName": "Vlad",
      "lastName": "Trifa",
    },
    {
      "firstName": "Michael",
      "lastName": "Haenni",
    }
  ]
}
```

Listing 3.3: Example JSON object with a list (friends) of two JSON objects holding both two key/value pairs

3.2.3 Protocol Buffers

Protocol Buffers\(^2\) is a highly optimized data structure developed by Google. The structure of the data is specified once in a .proto file, then the encoding/decoding will be performed by the library (Figure 3.1). The encoding in textual form uses a similar approach as in JSON, however with the .proto-file type-safety can be maintained. To further optimize performance, protocol buffers allow to be encoded into binary format massively speeding up the transmission speed and reducing data overhead.

\(^2\)http://code.google.com/apis/protocolbuffers/docs/overview.html
Figure 3.1: Example .proto file for a Friend message (left) and the corresponding textual encoding (right).

3.2.4 Summary

From the aspect of performance and transmission size protocol buffers seem to be the best choice. With its intermediate textual structure similar to JSON human readability is maintained helping debugging.

Due to the fixed structure of .proto-files and due to the need of an intermediate compile step to generate the (de)serializer stubs from the .proto-files, protocol buffers lack of flexibility. Besides, protocol buffers are developed by an industry company, currently without any standardization efforts, eg. W3C, IETF.

XML together with an XSD grammar is the most generic data-structure allowing the serialization of any kind of character data in a structured and machine readable way. Sender and receiver agreeing on the structure can even maintain type-safety. Many tools allow optimized search and modification of XML elements.

The generality of XML comes at the price of more overhead. In [25] a quantitative comparison between JSON and XML has been performed, showing that for large datasets the overhead becomes non-negligible.

Our aim for the gateway architecture is to use as many open standards as possible honoring the efforts for standardization and inter-operability. Although protocol buffers seem to be an interesting format for future use, we renounce their usage for now, as we believe that standardization is more important for the Web interoperability than performance.

To save bandwidth during the transmission of data from one computer to another, we chose JSON. We think, that JSON offers a good tradeoff between size and human readability.

To encode complex data-structures within the gateway (Section 4.4.1), we select XML as the most generic representation.

```proto
message Friend {
  required string firstName =1;
  required string lastName =2;
}
```

```text
Friend {
  firstName: "Vlad"
  lastName: "Trifa"
}
```
3.3 Events and Messaging

Distributing messages between several communication partners within a large community (like chats, live radio-stations, TV, sensor data) is a notoriously hard problem. We first present the basic terminology used and then exemplary highlight a few architectures for event distribution.

3.3.1 Event notification

According to [24] there are four basic mechanisms of event notification: Channels, Subject-Based Filtering, Type-Based Filtering and Content-Based Filtering - each with different strategies to determine the type of the event.

**Channels:** A publisher selects a named channel and publishes its notifications into this channel. Consumers select the channel they like and receive all the notifications created by any producer. This approach can somehow be compared to television, where the consumer tunes into a specific channel.

**Subject-Based Filtering:** Each notification is annotated with a subject string that denotes a rooted path in a tree of subjects. D-INFK could for example publish notifications in the subject-tree /University/Switzerland/ethz/dinfk. Consumers can register at several subjects in the tree.

**Type-Based Filtering:** Notifications are modeled in a type hierarchy with multiple inheritance explicitly allowed. The filtering process checks subtype inclusions and tries to use XPath expressions on the type hierarchy to determine the notification.

**Content-Based Filtering:** Evaluates the notification as a whole by applying XPath expressions, matching templates, arbitrary programs and mobile code.

3.3.2 XMPP

The Extensible Messaging and Presence Protocol (XMPP) [15] is an XML-based instant messaging protocol, initially designed for the transport of text messages. Today XMPP has been extended, such that voice-over-IP packets and the transfer of files are supported as well. XMPP is decentralized by design and there is no central master server orchestrating the network. Clients within the XMPP network maintain a unique identification ID such that they can be addressed from everywhere. However there is no direct client to client communication - servers provide a multi-hop routing delivering messages from one client to the other.

XMPP was originally developed as the core of the Jabber instant messaging service but today many other projects use XMPP as well as transport protocol (eg. Openspime³, Palantiri⁴, OpenFire [orig. Wildfire]⁵, Tigase⁶).

³http://www.openspime.org
⁴http://www.palantirisystems.com
⁵http://www.igniterealtime.org/
⁶http://www.tigase.org/
### 3.3.3 MQTT

The MQ Telemetry Transport protocol (MQTT)\(^7\) is a messaging protocol designed by IBM to enable messaging from tiny devices, such as sensors and actuators. MQTT uses a channel based eventing scheme (Section 3.3.1), where devices publish their events together with a topic (the channel):

```java
client.publish(
    "fireTopic",
    "floor1_OnFire".getBytes(),
    1, true);
```

In the background a message broker (example IBM MQ WebSphere\(^8\)) receives the events and distributes them to all the clients that subscribed to events of the generated topic.

### 3.3.4 JMS

The Java Message Service (JMS)\(^9\) provides message oriented middleware, connecting producers and consumers by means of queues. Connections between producer and consumer can be either point-to-point (producer knows consumer and delivers the message directly to it) or publish/subscribe (with an intermediary JMS Provider). In pub/sub the producer selects a channel (JMS topic) and delivers the message to the message provider. It is then the task of the message provider to dispatch the message to all the clients that registered for this channel.

There are many implementations of the message provider available (commercial and open source) that provide additional QOS (delivery guarantee, persistence, transactions, ...).

### 3.3.5 OSC

Open Sound Control (OSC)\(^10\) is a protocol for real-time communication among computers, sound synthesizers, and other multimedia devices. OSC tries to provide an open-source alternative to the highly successful MIDI protocol\(^11\). OSC-endpoints deliver notifications according Subject-Based Filtering (Section 3.3.1), allowing to send a message to a group of listeners. The following code snippet shows an example of an OSC-listener registered in the subject-tree at the leaf `enter` in the subtree `tag`:

```java
OSCListener enterListener = new OSCListener() {
    public void acceptMessage(Date time, OSCMessage message) {
        System.out.println("tag enter event.");
    }
};
oscReceiver.addListener("/tag/enter", enterListener);
```

Data can be exchanged in the `OSCMessage` as an `Object` array allowing efficient binary data transfer for any kind of data (eg. as protocol buffer or JSON).

---

\(^7\) [http://mqtt.org](http://mqtt.org)
\(^10\) [http://opensoundcontrol.org](http://opensoundcontrol.org)
3.3.6 Summary

XMPP, JMS and MQ are mature projects that have proven great success. XMPP is used in the Jabber instant messaging framework as transport protocol mainly for text messages (encoded as XML documents). Originally limited to character data, today XMPP is capable to encode binary data into XML as well, introducing some artificial overhead. JMS and MQ circumvent the issues with binary data by allowing any kind of data in the payload (basically both introduce a dedicated object structure that maintains the payload decoupled from the routing information).

All three messaging paradigms support many additional QOS, moving work from the client into the dedicated infrastructure. This helps clients to build lightweight, loosely coupled distributed applications. Thanks to the standardized interfaces, it is easy to interconnect different applications of the same messaging framework. Sadly, this is not the case when mixing the different technologies. Connecting a JMS message provider to a Jabber server needs a dedicated intermediary that translates the message format from one encoding to the other. In other words, this means that Web resources would have to implement their translators, in order to be capable of consuming events.

For our embedded gateway approach, a full broker (JMS or MQ) or a complete XMPP server is too heavy weight (the few light-weight implementations are not open-source). Especially Jabber servers require a fine-grain configuration with many runtime-critical parameters, making it hard to start the server in a dynamic environment. We want from the messaging server a ”zero-configuration”, such that the gateway can be started without prior configuration.

OSC provides a flexible event distribution protocol for point-to-point communication lacking multi-cast distribution, to target several computers at the same time. This fact renders OSC unusable in a large distributed environment. However, we still use it for device to gateway communication (Section 5.2).

In our messaging implementation (Section 4.3.4), we focused on the seamless integration into the Web. From JMS, XMPP and MQ we borrowed the idea of intermediary nodes that dispatch messages between users and other intermediaries. The intermediate nodes are modeled as Web resources linked via URI, simplifying (re)use and at the same time providing a loose coupling. The interface between the intermediaries follows the REST principles by using HTTP. Messages are encoded as JSON objects allowing the storage of any kind of data. Web resources can directly use the standardized data format and create mash-ups for the Web.
Chapter 4

System design

"A day without laughter is a day wasted."
Charlie Chaplin (1889-1977)

The gateway architecture was designed with four main goals in mind: simplicity, extensibility, modularity and openness. Simplicity and extensibility to enable users to extend and customize the gateway to their needs. Modularity so that internal components of the gateway can interact only through small interfaces, thus allowing the evolution and exchange of individual parts of the system. By using only publicly accessible standards and components that are free to use (libraries) the gateway can be used and enhanced by everyone.

Our gateway implementation is written mainly in Java with several driver components in C/C++, Python and Perl. The architecture is composed out of three major layers - the presentation layer (Section 4.2), the control layer (Section 4.3) and the device abstraction layer (Section 4.4) - each responsible for a well defined set of tasks. Figure 4.1 shows a high level overview of the gateway.

4.1 Core

The core specifies the "kernel" of the gateway providing utility classes for all the components of the architecture. Appendix F.3 shows the class diagram for the core and the most important components.

To ensure that there is only one core available at runtime, the "Singleton-pattern" [12] is used where the core can be accessed through the static method:

public static Core getInstance();
At startup, different tasks get executed by the core. Configuration files get loaded, the HTTP server for the presentation layer (Section 4.2) gets started and the plugins for the control layer (Section 4.3) get initialized. At shutdown, the core ensures that the current configuration of the gateway gets stored back to the configuration files and that all the resources occupied get freed again.

### 4.1.1 Configuration

The core provides a central configuration management where components can retrieve and store settings. At startup the configuration file is loaded into memory. All the properties are stored as key/value pairs that can be queried:

```java
String key = "myConfigurationKey";
Configuration cfg = Core.getInstance().getConfig();
Object value = cfg.get(key);
```

When the gateway is instructed to shut down, the configuration is stored back to the configuration file. Components that implement the interface `ConfigurationStorable` can register at the configuration management in order to be saved together with the core configuration:

```java
public MyComponent implements ConfigurationStorable {
    public MyComponent() {
        // register to be stored at shutdown/snapshot
        Core.getInstance().getConfig().register(this);
    }

    public Map<String, Object> getConfiguration() {
        // provide the config when asked for it
        Map<String, Object> cfg = new HashMap<String, Object>();
        cfg.put("myConfigurationKey", "myConfigurationValue");
        return cfg;
    }
}
```
The configuration management can be instructed to perform a "snapshot" whenever a component changes its configuration. A snapshot is a copy of the current configuration stored to a temporary file. If the gateway crashes unexpectedly because an illegal operation has been performed, the latest working copy of the configuration can be loaded at startup and the gateway recovers.

### 4.1.2 Structure Maintenance

Gateways can be linked into tree structures (Figure 4.2). This allows grouping of devices hierarchically according to some metrics (e.g., their location in a building, or the types of services provided by the devices). The structure maintenance ensures that all the control information needed to maintain the grouping gets passed between the different gateways.

**Location**

In the current implementation, each device has a well-defined symbolic location in the tree hierarchy starting at the root of the tree. When a new device is attached to the gateway, this device inherits the location of the root and adds itself as a new root at the end of the location (e.g., the root has the location `/rootLocation`). A new device `sunSpot1` is added. The device inherits `/rootLocation` as location in the tree and adds itself into the tree under the location `/rootLocation/sunSpot1`). The same process is applied for gateways that are structured into a tree. Figure 4.2 shows an example of a tree with three gateways.

The structure of the tree directly implies that there are two ways to address and access a gateway in the tree:

1. **Direct addressing:** You need to know the IP address of the gateway you want to interact with (Can be obtained by performing a search for the gateway (Section 4.3.5)). Then you can directly call the gateway on its IP.
   
   Example how to access `child2` from Figure 4.2: Just perform an HTTP request to `http://3.4.5.6/`.

2. **Hierarchical tree addressing:** You need to know the location of any gateway in the tree and the IP address of the root gateway. Then you can call the root (or any intermediate gateway) by direct addressing, and from there follow the symbolic location.

   Example how to access `child2` in Figure 4.2:

   (a) Perform an HTTP request to the root with the location of the child relative to the root appended: `http://1.2.3.4/child2`.

   (b) The root extracts the target `child2` from the request and forwards the call directly to the respective node.

**Keywords**

Each device also provides a set of keywords that allow the classification of the device (Example: A temperature sensor would possibly serve the keywords `temperature` and `heat`). Keywords are used to distinguish different events (Section 4.3.4). Initially, keywords have been propagated from the bottom of the tree up to the root, each intermediate node taking the union of all the keywords of its subordinate devices. This way, each intermediate gateway knew all the keywords of the subtree which allowed an optimized eventing mechanism (basically restrict the
event registration only on the subtrees that contain the keyword). Different tests revealed two performance problems. Because of that, the initial approach was rejected:

1. When \( n \) gateways are structured in a chained list then it takes \( n \) heart beat rounds (see below) until all the keywords get propagated to the root. In some lethal case, this could take several minutes. As opposed to the location, which is mostly static, keywords often change (because devices arrive/leave) and therefore such long delays are not acceptable.

2. When many devices with many different keywords are attached to the tree, the root gateway ends up with a huge table holding all the keywords. This requires a lot of memory and computational power which is opposite to the idea of a lightweight and distributed gateway.

As a solution to the problem, the keywords are kept locally on gateway level (devices propagate their keywords to their gateway) and gateways do not propagate their keywords to their direct parents.

![Diagram of three gateways organized in a tree structure together with attached devices.](image)

**Figure 4.2:** Three gateways organized in a tree structure together with attached devices. Solid arrows indicate the propagation direction for location information, double arrows (=) how keywords are propagated and the bended arrows (\( \sim \)) the direction of the name.

### Registration

At startup the child gateway receives a URI pointing to the parent gateway. If this URI does not equal to null (meaning that the gateway is not the top root gateway in the tree) the registration phase is initialized by the structure maintenance. The child sends three mandatory parameters to the parent:

1. The name of the child gateway.

2. The name of the class that will provide the service driver for the communication from the parent to the child.

3. A URI where the child can be reached.

The parent extracts the parameters and generates a driver through reflection using the device abstraction (Section 4.4). As soon as the driver is up and running, the parent switches to the
heartbeat protocol sending the child its location (Listing 4.1).

```
function register
    if parent not null
        registerAtParent(name, driver, myuri)
        if ok then
            switch to heartbeat
        end
    end
end
```

Listing 4.1: “Pseudo-code for the registration protocol.”

**Heartbeat**

The structure maintenance tries to reach the parent at regular intervals and sends the name of the child as identification string (Figure 4.3). When the parent gateway is reachable, it responds with the current location of the parent in the tree together with the URI of its parent (the grand parent of the child). The child updates its state (means updates its location) and stores the grand parent. In the case where the parent is not reachable, the child tries to install an alternate route directly to the grand parent which remains active as long as the parent is not reachable. Listing 4.2 illustrates the process in pseudo-code.

```
function contactParent
    repeatPeriod = REPEAT_PERIOD
    while true
        // receive the location and the grandparent
        location, grandpaAddress = sendHeartBeat(parentAddress, myName)
        // could not contact the parent
        if error then
            install fallback route to grand parent
            // retry in double the time
            repeatPeriod = 2 * repeatPeriod
        else
            stop fallback route to grand parent
            repeatPeriod = REPEAT_PERIOD
        end
        sleep(repeatPeriod)
    end
end

function contactGrandParent
    repeatPeriod = REPEAT_PERIOD
    while route to parent down
        location, grandpa = sendHeartBeat(grandpaAddress, myName)
        sleep(repeatPeriod)
    end
end
```

Listing 4.2: “Pseudo-code for the heartbeat protocol.”
If a child starts the heartbeat by accident without being registered at the parent, the parent notifies the child with a HTTP error status code and the child then runs a registration first. A nice side-effect of this protocol is an implicit tree-stabilization that should keep the tree structure connected.

![State diagram of the structure maintenance (Without fail-over).](image)

**Figure 4.3:** State diagram of the structure maintenance (Without fail-over). The system starts in state Unregistered. This state is maintained until a registration has been performed at the parent (Unless as the gateway is the top gateway). As long as the heart beat protocol successfully runs the state Registered is maintained, otherwise the state machine falls back to Unregistered.

### 4.1.3 System timers

Many processes within the gateway have to be executed at regular time intervals (check whether devices are still alive, generate events, etc.). Most programmers solve this by using threads to trigger special methods. To avoid code duplication (by repeating the same code over and over again) and to avoid errors (thread programming is notoriously error prone) the core provides a central timer service - the `PollingService`. Objects that need to execute tasks at regular intervals implement the interface `Pollable` and can then register at the polling service to be notified whenever a certain timer interval has elapsed. Multiple registrations with different time intervals is possible as well (Listing 4.3).

```java
Pollable p = new Pollable {
    public void onPoll(long interval) {
        if (1000 == interval) {
            System.out.println("polling each second");
        } else if (60000 == interval) {
            System.out.println("polling each minute");
        }
    }
};
Core.getInstance().getPollingService().register(p, 1000);
Core.getInstance().getPollingService().register(p, 60000);
```

**Listing 4.3:** "Creation of a pollable object that registers at the polling service to be polled in two intervals (1000ms and 60000ms):"
4.2 Presentation

The presentation layer makes the gateway components accessible to the Web. On top of the control layer (Section 4.3), it manages requests from clients through a REST interface implementing the "Facade Pattern" [12]. The facade pattern tries to decouple clients and complex subsystems by introducing a clean higher level interface (Figure 4.4).

![Figure 4.4](image)

**Figure 4.4:** The image on the left shows how clients access a complex subsystem without facade. The image on the right introduces a facade that decouples clients from the subsystem (Images taken from [12]).

We use the Restlet Framework[^1] to provide a RESTful HTTP-server. Restlet uses **Resources** or **Restlets** to represent resources on the Web. We reused those two building blocks to implement our facade (Figure 4.5). Resources implement all the necessary interfaces to provide REST operations towards the Web, and at the same time link those operations with the internal subsystems, effectively decoupling the presentation layer from the model.

![Figure 4.5](image)

**Figure 4.5:** Implementation of the facade pattern in the gateway using the Restlet framework. Different resources (eg. DeviceResource/SearchResource) provide the facade towards the Web.

All the resources on the gateway can currently be retrieved in three different formats (XML, HTML, and plain text) allowing clients to choose their preferred format. Restlet delivers with each request a data-structure that contains all information specified in the client request. From these data-structures, the server can retrieve the desired resource representation. Translators

[^1]: [http://www.restlet.org](http://www.restlet.org)
To simplify browsing on the gateway, the presentation layer additionally performs a mapping from device names to URL. All devices attached to the gateway can be accessed by their device name (e.g. http://gw_IP/myDevice). A dedicated resource extracts the device name from the request and redirects the call to the responsible device driver. The handling of the request is explained in detail in (Section 4.4).

### 4.3 Control

The control layer maintains the application logic for the gateway. The control layer is split into several (independent) components - we call Plugins (Section 4.3.2). Figure 4.1 on the right (Internal API) illustrates the concept graphically.

#### 4.3.1 Translator Management

In order to support any kind of MIME-type representation the concept of Translators has been introduced. A translator is a piece of software that is capable of translating a representation of an object into another representation (e.g. translate a XML document into a HTML document). The translator manager maintains all the available translators and makes them accessible to the gateway.

To save memory, translators are loaded lazily. At startup, only a descriptor, describing the translator will be loaded. For that, the classpath is explored for specific "marker-files" at initialization of the translator management (Appendix E.1). As soon as a specific translator is requested from any of the gateway components, the translator management dynamically class-loads the required software parts (using the descriptor) and makes them available.

**Translator**

The current implementation provides a set of translators all using an XSLT style-sheet to transform the device representation (Section 4.4.1) into the desired target format.

Users can write their own translators and place them (packed as a jar-file) into the classpath of the gateway. The class SampleTranslator provides a sample implementation showing the basic procedure when writing a new translator:

```java
public class SampleTranslator extends Translator {
    public SampleTranslator(TranslatorDescriptor descriptor) {
        super(descriptor);
    }

    public Object translate(Object o) throws TranslatorException {
        throw new TranslatorException("!! sample translator does nothing!!");
    }
}
```
4.3. Control

The jar-file has to contain a "marker"-file props/translator.xml that contains one or more descriptor(s) for the translator management. A sample descriptor has the form:

```xml
<Translator ID="ch.ethz.inf.vs.gateway.translator.core.XML2HTML"
Version="1.0.0" From="text/xml" To="text/html"/>
```

The ID identifies each translator uniquely and is used at runtime to classload the translator as soon as it is used. The two fields From and To describe the source/target MIME-type handled by this translator. At runtime each translator makes the meta-data (descriptor file) accessible through the final attribute descriptor.

4.3.2 Plugin Management

A plugin is a software component that is loaded at startup of the gateway through the core (Section 4.1). Users can write their own plugins and place them (packed as a jar-file) into the classpath of the gateway. The jar-file has to contain a "marker"-file props/plugins.xml that contains a descriptor for the plugin with the mandatory fields Version, ID, and Dependency (IDs of plugins that need to be loaded before this plugin) - the classpath is explored for these "marker-files" at initialization of the plugin management (Appendix E.1). The "marker" contains the class name that will be used later for the class-loading. A sample descriptor has the form:

```xml
<Descriptor>
  <ID>ch.ethz.inf.vs.gateway.plugin.devices.Devices</ID>
  <Version>1.0.0</Version>
  <Lazyload>true</Lazyload>
  <Dependencies/>
</Descriptor>
```

Plugins are allowed to depend on other plugins (eg. the eventing plugin depends on the device management plugin). To keep a loose coupling between plugins, a lightweight synchronization mechanism is implemented with the "Observer-pattern" [12]. The observer (plugin depending on some other plugin) will be registered on the observable. As soon as the observable changes its state, an internal synchronization method is invoked informing all the observers about the change (eg. the devices plugin calls this method whenever a device is removed or added).

4.3.3 Device Management Plugin

The Device Management plugin maintains a high level view on devices registered at the gateway by using the device abstraction (Section 4.4). Each device is identified uniquely by its name. The plugin exposes a REST interface towards the Web allowing devices to be added and removed through the URI http://gw_IP//device.

Add device: To create a new device, only two mandatory parameters are required: the class-name of the device driver and a unique name for the device. The device registration can then be performed in two independent ways:

- From within the gateway via Java: All the required parameters have to be packed into a HashMap as key/value pairs.
- From the Web via HTTP: The parameters are passed within a Web-form with a PUT/POST-request. The registration extracts the parameters and stores them into a HashMap.
The code-snipped shows how to create a new device from within the gateway:

```java
Map<String, Object> params = new HashMap<String, Object> ();

// mandatory parameters
params.add(Constants.CREATE_DEVICE_PARAM_NAME, "myDevice");
params.add(
    Constants.CREATE_DEVICE_PARAM_CLASS,
    "ch.ethz.inf.vs.gateway.mydevice.MyDevice");

// optional parameters
params.add("vendor", "eth zurich");

Devices mgmt = Core.getInstance().getDeviceManagement ();
mgmt.invokeAdd (params);
```

**Device heartbeat:** On each device driver, a heartbeat protocol is installed which requires the device to keep its status alive. This means that at regular intervals, a watchdog is invoked and iterates over all the drivers and invokes the method isAlive():boolean. If a driver does not respond with true the driver gets removed from the device list. Note that the implementation does not poll the real device for aliveness - the aliveness test within the driver towards the physical device is usually performed using an optimized manner depending on the device.

**Remove device:** To remove a device, a DELETE request has to be sent to the registration together with the name of the device to be deleted. The management then force removes the device. This means that we remove the device whatever state it maintains.

### 4.3.4 Eventing Plugin

Many sensors read their state at a regular interval and a programmer usually has to check periodically whether this state has changed or not. When using sensors over the Internet, polling is inefficient and creates unnecessary load that can be avoided by using an asynchronous publish/subscribe model.

From the clients perspective it suffices to send a simple POST request to the event registration URI with three mandatory parameters - lease time to specify how long the registration shall be valid, the keyword specifying the type of event the registration is for, and the callback giving an address where to deliver the events (Listing 4.4).

Consider the tree structure from Figure 4.6. The client registers at the top node (floor1) for a fire-event triggered by any of the sensors in the subtree of the floor1-gateway. As soon as a fire-event is triggered on the gateway, the client will be notified about the event through the callback address provided in the registration request. In our case, the floor1-gateway is also connected to other gateways in its subtrees. It therefore has to register on these gateways as well for fire-events. This registration works in exactly the same manner as with the client registration, but this time with the floor1-gateway as client on the different room-gateways.
Listing 4.4: Example for a client event registration. The registration is valid for 6 seconds (6000ms). Events of type fire will be sent to the callback client_callback_address.

```
POST /_eventing/registration HTTP/1.1
Host: ip_of_the_registering_client
leasetime=6000
callback=client_callback_address
keyword=fire
```

Figure 4.6: Sample gateway hierarchy with a gateway at the top of the tree (floor1) with two gateways in the subtree (room1 and room2). The solid arrows indicate the path the registration message takes through the gateway. The pointed arrows indicate the path the event message follow.

In the current implementation we use this simple eventing approach. However, more advanced approaches would be more appropriate for large scale implementations (eg. MQTT (Section 3.3.3) or XMPP (Section 3.3.2)).

4.3.5 Search Plugin

Searching for devices that meet a certain criteria has become an important aspect of every modern distributed architecture. The focus of this thesis has not been on creating a new search algorithm, therefore the presented search algorithm is very basic and should be extended in future work.

We implemented a simple radius-growth algorithm that runs on the tree hierarchy (Listing 4.5). In the radius-growth search you start searching for matching devices at your direct neighbors (the gateways that can be reached within one hop). If a matching device is found then the algorithm stops. Otherwise the search radius is doubled. This procedure is repeated until a maximum radius is reached or a matching device is found. In the gateway hierarchy the search follows the tree structure up (from children to the parent) and down (from parent to children).

The difficulty of such a search algorithm in a totally asynchronous distributed system is, how to set the maximum radius (TTL) and how to set the timeout between two subsequent search rounds. Both questions are not answered in the current work and require further research.
Listing 4.5: "Pseudo-code for the radius-growth search algorithm."

To determine which devices match a search request, the search routine performs two strategies: ID-based search and search with XPath expressions. Both search routines return a JSON array with all the devices found together with their access-point (IP address of the responsible gateway and the ID of the matching device).

**ID-based search**

Within the gateway, each device maintains a unique identifier (Section 4.4). ID-based search compares the search request with those device IDs and returns at most one device. An ID-based search request for a device with the name `sensor1`: `GET http://gwIP/search/sensor1`.

**XPath expression search**

The gateway maintains an XML representation for each device (Section 4.4.1). XPath is a language, that allows to navigate and search within XML documents. The XML document is transformed into a tree with nodes and leaves. An XPath expression then formulates a constraint to be matched onto the structure of the tree.

Consider the XML from Listing 3.2. An XPath expression to search for all friends with `firstName Vlad` has the form: `/friends/friend[firstName='Vlad']`. The result of the search is:

```xml
<friend>
  <firstName>Vlad</firstName>
  <lastName>Trifa</lastName>
</friend>
```

---

2[http://www.w3schools.com/XPath](http://www.w3schools.com/XPath)
4.4. Device Abstraction

In a first step, the gateway performs a few health-checks (with regular expressions) on the search string to ensure, that it encodes a valid XPath expression:

```java
private boolean isXPath(String searchExpression) {
    // expressions that contain [,],|,(,),=,@,{,} are
    // considered to be XPath expressions.
    // notice ( is escaped by \\
    Pattern pattern =
        Pattern.compile(
            ".*[\[|\]|\(|\)|=|\@|\{|\}]+.*") ;
    Matcher m = pattern.matcher(searchExpression);
    if (m.matches()) { return true; }
    return false ;
}
```

Then the expression is compiled and applied to the XML-representation of each device.

```java
// compile the expression
XPath xpath = XPathFactory.newInstance().newXPath();
XPathExpression expr = xpath.compile(searchExpression);

// create the document builder
DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();

// iterate over all the devices.
for (Device d : mgmt.getAll()) {
    // get the XML representation.
    ByteArrayInputStream bin = new ByteArrayInputStream(d.asXML().getBytes());
    Document doc = builder.parse(bin);

    // evaluate the expression
    String s = expr.evaluate(doc);
    ...
}
```

The search algorithm collects and returns all the devices, where the XPath search resulted in a non empty result.

4.4 Device Abstraction

Like in almost any modern operating system, the gateway provides an abstraction for devices. For higher level applications, any type of device looks the same, even if the underlying implementation differs.

The device abstraction is illustrated on the left part of Figure 4.1. Different specialized drivers are used to communicate through their ”proprietary” protocol with the respective physical device (eg. SunspotDriver communicates through IEEE 802.15.4 with the SunSpot). From a higher level perspective all the drivers implement the abstract class Device allowing the device management plugin to treat all the devices in the same manner. Appendix E.2 contains an
example of a fully implemented device driver.

For devices that are already Web-enabled, the driver implementation can simply forward requests from the presentation layer (Section 4.2) to the physical device. However, when the device is not Web-enabled (e.g. SunSpots do not have a IP stack), the driver is responsible to translate the Web request into a protocol understood by the physical device. The code snippet shows how a device driver can handle a request from the Web:

```java
public String handle(Response response, Request request) {
    List<String> segments = request.getResourceRef().getSegments();

    // only proceed if device name matches
    if (getDeviceName().equals(segments.get(0))) {
        // remove the device name
        segments.remove(0);

        if (segments.size() > 0) {
            // there are more segments to be processed
            String m = segments.get(0);

            if ("myDeviceResourceName".equals(m)) {
                // do something
                ...
                return null;
            }
        }
    } else {
        // default case
        return asXML();
    }
}
```

A crucial feature of a device driver is the capability to act as a resource representation (or proxy) for the underlying physical device. Consider a temperature sensor changing it’s temperature seldomly. Instead of polling the device every time a client requests the temperature, the driver can store the temperature and return the value directly, thus minimizing the actual communication with devices. This caching mechanism is extremely useful for shared access to quasi real-time sensor data collected with low-power devices.

### 4.4.1 XML Representation

Internally (aside the device abstraction representation) each device is represented as an XML data-structure following a XSD style-sheet (Appendix G). The XML is generated lazily and cached for subsequent requests (and invalidated in case the device changed its state). The XML can be used as a basis for many higher level services:

**MIME type translation:** From XML many different representations can be generated on-the-fly using XSLT style-sheets in translators (Section 4.3.1).

**Search:** Searching a specific device programmatically can be tedious and cumbersome. The gateway provides a simple XPath based search routine allowing internal and external clients full access to the registered drivers (Section 4.3.5).
4.5. Firefox integration

**Ontologies:** Thanks to the well-formatted structure of the XML, ontologies can be developed on top of the representation allowing automation and "reasoning".

### 4.4.2 Contiki and TinyOS

In his Master Thesis [20], Andreas Kamilaris implemented a device level abstraction in Contiki and TinyOS for Tmote. Together, we developed a device driver to bridge the smart gateway to the Contiki/TinyOS motes.

The interesting research question of this work is how to bridge the two systems that follow two orthogonal communication principles. The gateway provides a synchronous communication to the Web (request-reply), whereas the communication to the Tmotes is performed in a totally asynchronous way - the client posts a message into a message queue, the reply will arrive at some unspecified time and is displayed in the command line).

We introduced synchronizers, to "translate" the asynchronous behavior into a synchronous one. Restlet creates a new thread per request from the Web. In the handle-code (Section 4.4), the thread obtains a unique message-token (message-id), sends the asynchronous request together with the token to the Tmote driver and obtains a lock on a unique synchronizer token (identified by the message-token). As soon as the response to the message-token arrives, the synchronizer lock is searched and the waiting thread is awakened. Figure 4.7 shows the sequence diagram and Appendix E.3 shows parts of the implementation.

![Figure 4.7: Synchronizer for asynchronous to synchronous communication](image)

1. The client obtains a unique message token. 2. Together with the token, the request is sent asynchronous to the Tmote driver. 3. The client obtains (with the message token) a lock on the Thread-Lock and puts itself to sleep. 4. The message arrives from the Tmote. The gateway awakes the client thread together with the message-token, and the synchronous execution continues.

### 4.5 Firefox integration

Once the gateway hierarchy is installed and ready to run, one key issue still remains: how can a user with a mobile device (be it a laptop or a mobile phone) discover the gateway in its vicinity? When there are several gateways around, which one is the most suitable?

---

3[^contiki]: http://www.sics.se/contiki/
4[^tinyos]: http://www.tinyos.net
• A possible solution is to scan a RFID or a bar-code with a mobile phone to retrieve the URI associated with a location [27, 21]. However, this approach requires specific software and/or hardware.

• Another idea could be to write the URI of the nearest gateway on the wall and the user then has to copy this URI into the browser address-bar. This approach does not use any additional hardware, however it requires user intervention, which should be omitted as much as possible.

• With a lightweight software agent on the client side a discovery protocol could search for the gateways in the vicinity and present those to the client (eg. Apple Bonjour). Advantage of this approach is that there is no user intervention, however the client needs to install additional software.

We chose the third approach with the lightweight discovery protocol. The disadvantage of the requirement for additional software is compensated by the fact, that there is no need for expensive additional hardware and that there is no user intervention after the software has been installed.

4.5.1 Discovery protocol

The discovery protocol uses multi-cast [33] to call the gateways in the local (wireless)-network. The protocol uses two different messages (the value in brackets denotes the length in bytes and (X) denotes variable length):

• Discovery request message:
  
  \[
  \begin{array}{c}
  \text{msg\_type} \ (1) \\
  \end{array}
  \]

  \text{msg\_type set to 0x1 (request message).}

• Discovery answer message:
  
  \[
  \begin{array}{cccc}
  \text{msg\_type} \ (1) & \text{nameLen} \ (2) & \text{uriLen} \ (2) & \text{name} \ (X) & \text{uri} \ (X) \\
  \end{array}
  \]

  \text{msg\_type set to 0xd (answer message).}

  \begin{itemize}
  \item \text{nameLen: length of the name-field in byte.}
  \item \text{uriLen: length of uri-field in byte.}
  \item \text{name: encodes the name of the gateway as a byte array.}
  \item \text{uri: encodes the URI of the gateway as a byte array.}
  \end{itemize}

  Each gateway instance maintains a multi-cast server socket, that listens for incoming multi-cast requests. When the gateway receives a discovery request message from a client, it returns a discovery answer message containing the name of the gateway (eg. ifwGateway) and the URI where it can be accessed (eg. http://www.ifw.eth.ch:8218).

4.5.2 Implementation

On the gateway side the implementation is straightforward and relies on the the Java Multi-castServerSocket that runs on a separate thread. On the client side a more sophisticated approach had to be found (as the Java security model does not allow browser applets to execute threads or sockets without proper policy-files written by the user itself). As a proof of concept, we chose
4.5. Firefox integration

To write a Firefox extension\(^5\). Firefox provides a simple but powerful extension mechanism that allows users to install additional software components as plugins into their browser with one single click. Figure 4.8 shows a screen-shot of the plugin in action.

**Figure 4.8:** The firefox discovery plugin in action. It presents three gateways in the local network to the client. On the bottom right you can see the button that activates the discovery plugin (red D within highlighted box).

**UI**

The graphical user interfaces in all Mozilla derivatives have to be written in XUL\(^6\) which turned out to be a straight-forward task. XUL is a XML-like markup language especially designed to create and extend user interfaces. The XUL-Code from Listing 4.6 shows how the statusbar of the browser can be extended with a little button linked to the discovery service oDiscovery.onDiscover().

```xml
<?xml version="1.0"?>
<overlay id="gatewaydiscoveryStatusBarID"
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
<script type="application/x-javascript"
src="chrome://gatewaydiscovery/content/discovery.js"/>
$statusbar id="status-bar">
<statusbarpanel id="gateway-statusbar">
<toolbarbutton
    image="chrome://gatewaydiscovery/skin/icon.png"
oncommand="oDiscovery.onDiscover()"/>
</statusbarpanel>
</statusbar>
</overlay>
```

**Listing 4.6:** XUL-Code that extends the browser status bar with a "clickable" button.

---

\(^5\)http://addons.mozilla.org

\(^6\)http://www.mozilla.org/projects/xul/
Backend

In order to gain full access to system resources (like sockets for multi-cast and threads for concurrency) and to stay platform independent (Windows, OSX, Linux), we chose to implement the extension in C++ using XPCOM (Cross Platform Component Object Model)\(^7\) and NSPR (Netscape Portable Runtime)\(^8\). XPCOM is comparable to Corba\(^9\), and the interface of the application is formulated in a IDL (Interface Description Language)\(^10\). From the interface, application stubs for the respective implementation language (in this case C++) are obtained, by using an interface compiler available for different platforms. NSPR gives a uniform access to system components (comparable to Java VM) making the extension platform independent. The IDL-Code from Listing 4.7 shows the different methods that are accessible (and scriptable by JavaScript) from external.

```c
[scriptable, uuid(6f359100-d704-41a6-9026-4b0fc7b1ff00)]
interface IGatewayDiscovery : nsISupports
{
    /** starts a discovery process. */
    void startDiscovery();

    /** stops a discovery process. */
    void stopDiscovery();

    /** restarts a discovery process. */
    void restartDiscovery();

    /** the gateways found by the discovery round. */
    readonly attribute nsIArray gateways;
}
```

**Listing 4.7**: IDL-Code of the interface to the discovery extension.

To link the C++ implementation to the XUL user interface, a JavaScript component had to be implemented. The JavaScript starts the discovery process and provides the user interface with the answers from the gateways.

\(^7\)http://www.mozilla.org/projects/xpcom/
\(^8\)http://www.mozilla.org/projects/nspr/reference/html/
\(^9\)http://en.wikipedia.org/wiki/CORBA
\(^10\)http://en.wikipedia.org/wiki/Interface_description_language
To show the capabilities of the gateway and to accent the advantage of using Web technologies in based on RESTful principles, we developed several prototypes. They illustrate the flexibility and ease of use of our gateway implementation when used with a variety of different input and output devices under very different scenarios.

5.1 Distributed Energy monitor

The distributed energy monitor was developed for a research paper [34] submitted to the LOCA conference. It illustrates how the gateways work by implementing a mobile, location-aware energy meter. The prototype is an ambient device that changes its color according to the level of energy consumption of the place it is currently located in. This prototype illustrate how simply location-aware applications can be built by using a Web-based infrastructure. Moreover, it provides a good example of what we call physical mash-ups for the Web of Things, since it is a composite application based on three types of devices that are integrated to the Web using REST.

5.1.1 Scenario and Interaction

As shown on Figure 5.1 (circle 1) the device starts in room D43.1. Upon entering and while it stays in this room, it gets the energy consumption of the devices in the room from connecting to a number of energy sensors (white boxes next to the laptop and the kettle). Depending on the total amount of energy consumed, the Ambient Meter changes its color from green when the
Figure 5.1: The prototype deployment on the floor of the IFW building, with three physical gateways in two different rooms and one in the hallway. Two energy meters are connected to the corresponding gateways. As the Ambient Meter moves from one room to the other its color changes according to the level of energy consumption in this room (1-3).

amount of energy consumed in the room is low, to red when a lot of energy is currently consumed in this place. The meter retrieves this information by automatically connecting to the closest gateway and asking it for the `energyMonitor` resource (GET http://gwIP/energymonitor).

The Ambient Meter is then taken to room D48.1 (circled 3 on Figure 5.1). On the way to this place it asserts the energy consumption of the South Wing of Floor D by connecting to the corresponding gateway while still asking for `energyMonitor`, and changes its color accordingly to the value it receives. Upon arrival in room D48.1, it monitors the consumption of a lamp and a desktop computer located in D48.1 again by connecting to the gateway of that room (circled 3 on Figure 5.1), and dynamically updates the display at a fixed time interval. Note that the ambient device does not actually have to deal with any specific location. It actually should not since it only wants to assert the energy consumption of the current location it is in. What the current location corresponds to will depend on the closest gateway (or the one with the best connectivity).

In the last part of this scenario (circled 4 on Figure 5.1) a user enters room D48.1. Using the Firefox plugin his browser (Section 4.5) automatically discovers the closest gateway. The user selects the one corresponding to the room he is currently in: D48.1. He gets back a Web page containing links to either related locations (e.g. D48.2, etc.) or to resources in the current room. By clicking on the D48.1/energyMonitor/ he gets the same information the ambient energy meter could retrieve about this room, namely the amount of energy currently consumed by the lamp and the desktop computer. As the Ambient Meter is also currently located in this room the user can click on its link and access its services. For instance this link can be used to retrieve the temperature currently sensed by the Ambient Meter for room D48.2. This part of the prototype illustrates how users can leverage the gateways structure, to browse (and bookmark) their physical location as they would browse Web pages.
5.1.2 Deployment and Implementation

As shown on Figure 5.1 the prototype was deployed in our building at ETH. The location information is provided by one logical and three physical gateways. The first gateway is logical (i.e. can be deployed on any device or computer, not necessarily at the location it manages) and covers the ETH IFW building as well as the D Floor. The second gateway is a NSLU NAS device supplemented with further communication capabilities (by using Bluetooth or IEEE 802.15.4 Wireless dongles), and covers the South Wing of the building. The two other gateways are deployed on two computers respectively in office D43.1 and D48.1. All these gateways are running the software described in (Chapter 4) and can communicate with the Ambient Meter over a wireless link (IEEE 802.15.4).

The Ambient Meter is a SunSPOT Java sensor node. In order to bring the SunSPOT into the Web, we deployed a REST architecture and a Web server on the nodes and its gateway [18]. As a consequence, all the functionalities of the SunSPOTs are made available through URI-identified resources accessible through a RESTful interface by using HTTP requests. This enables the mobile user to access the sensors’ resources simply by entering their URI in any Web browser.

As SunSPOTs do not have energy metering capabilities, the energy consumption of electric appliances can be monitored using a Plogg. Ploggs are sensor nodes that combine an electricity meter plug and a data logger that can be accessed using Bluetooth. As for the SunSPOT, we had to build another gateway that was needed to access the Ploggs through a RESTful interface. The RESTful structure we have built on top of both the SunSPOT platform and the Ploggs reduces the integration work to a mere URL redirection. For example, when the D48.1 gateway receives a request in the form of IFW/D48.1/energyMonitor, it just needs to redirect the request to the corresponding device’s Web server, in this case the Ploggs’ gateway.

5.2 Temperature sensor

The prototype was developed for a research paper [35] submitted to the DCOSS conference. The goal of the prototype is to illustrate how low-level devices like the Tikitag RFID reader can be made accessible to the Web and how it can be ”mashed-up” into higher level devices. Additionally it shows how Web-enabled devices (in this case the SunSpot) can easily access and control the gateway citizens. Figure 5.2 shows the setup graphically. The gateway is installed on a laptop, where a SunSpot base station for IEEE 802.15.4 communication and a Tikitag RFID reader are attached. As SunSpots per se do not have an IP interface, a reverse proxy [18] is used to multiplex 802.15.4 communication streams so that the SunSpots can access Web content directly through HTTP.

5.2.1 RoomState

RoomState is a virtual device (Web resource) that provides a high-level abstraction to read or change the status of a real room. For this demonstration RoomState simulates the temperature of a room that can be measured, increased or decreased using two interfaces:

1. Through a REST interface via the Web:
   The RoomState driver exposes a simple REST interface to Web clients that allows the

\[http://www.sunspotworld.com/\]
\[http://www.plogginternational.com\]
\[http://www.tikitag.org\]
Figure 5.2: Prototype for the DCOSS conference. The gateway maintains a virtual RoomState device. The Tikitag-reader is attached through the Tikitag driver. The SunSpot connects to the gateway with HTTP through the reverse proxy.

retrieval of the current value and the update to a new value.

GET http://RoomState/temperature will return the current temperature of the room. The actual implementation is hidden from the user and can change depending on the devices available in the room.

POST http://RoomState/temperature/X will change the temperature of the room to match the desired value. This could be implemented by sending a specific (proprietary) command to a climate control device.

2. By being notified through a particular event in the system:
   The idea is that each time an RFID tag is put on the Tikitag reader, a specific event will be generated. The RoomState resource can then register for events sent by the Tikitag driver. You can use different tags to either increase or decrease the temperature (their ID will correspond to the desired temperature).

5.2.2 Tikitag

An RFID reader is only a sensor and cannot provide visualization. Every time a new tag is read, the Tikitag reader sends an OSC message (Section 3.3.5) that contains the tag ID and the reader ID to the TikitagDriver, which will then notify the subscribers of RFID messages (in this case the RoomState).

5.2.3 SunSpot

The SunSpot periodically polls the RoomState device on the gateway using a GET on the URI of the RoomState resource to read the current value. The value is encoded as a json string and is then displayed using different led colors (eg. blue when the temperature is below 10 degrees, green when between 10 and 25 degrees and red when over 25 degrees). Additionally, the SunSpot can also be used as an input to control the room temperature, for example by using the acceleration sensor on-board (shaking the SunSpot for 2 seconds on the x-axis will issue a POST request on the RoomState resource decreasing the value whereas shaking on the y-axis increases the value).
5.3 House Keeping

The prototype has been developed for the presentation of [19] at ETHZ. The prototype shows the integration of the smart simulator into a hybrid setting with physical devices interacting with virtual sensors. The scenario models a hotel with one central breakfast room and several guest rooms on different floors. When guests are breakfasting, the housemaid receives a message from the respective gateway indicating her to clean the room. Five guests and one housemaid are emulated by the simulator. Additionally one guest and one housemaid are physically represented as RFID-tags (e.g., placing the tag of the guest onto the RFID-reader at the breakfast gateway means that the guest is breakfasting).

Three gateway instances are started in a tree structure (Figure 5.3). The top gateway provides the virtual breakfast room together with a Tikitag reader (Section 5.2.2). The two child gateways (A and B) have virtual rooms attached. B in addition provides a second Tikitag reader and a special guest room consuming the RFID events (e.g., guest or housemaid in the guest room).

The prototype has shown that the gateway enables higher level applications (in this case the smart simulator) to build mash-ups on top of the gateway architecture. First, the virtual guests searched the nearest breakfast room using the built-in search routine. Secondly the Tikitag reader generated real events that then transparently could be consumed by the virtual simulation agents. Third, the virtual rooms exchanged events between the gateway instances to start/stop the “cleaning” of the virtual guest rooms - all on top of Web standards!

![Diagram](image)

**Figure 5.3:** House keeping scenario. TT: Tikitag RFID Reader, BFR: breakfast room, GR: guest room, SGR: special guest room consuming events from TT. BG, A and B represent 3 physically deployed gateway instances.
5.4 Summary

Devices embedded into the Web allow to build applications consuming sensor output. Our prototypes show, how easy this task becomes when devices can be addressed and accessed with Web technology, enabling mash-ups using sensor data (e.g. Section 5.2) by simply patching different URI (representing the sensors). Using the Web requires no prior programming skills and therefore the devices become accessible and usable for anyone.

Integration of different applications usually takes a considerable amount of time. Incompatible interfaces and messages force programmers to develop adaptors and bridges to enable information exchange between the communication partners. However, with our Web-based approach this statement does not yield. Building on top of a rich ecosystem of Web-enabled devices, higher level applications can be patched together in no time. The integration of the location-aware energy monitor (Section 5.1) required two programmers to invest two hours of work to successfully deploy the scenario. Even with the more complex house keeping scenario (Section 5.3) we invested one day of two mens work in total.

Although our prototypes are simple applications built with the Web in mind, they indicate that the Web provides powerful tools for the integration of different middlewares and devices into higher-level services. From an economic view, this reduces costs and allows programmers to focus on building applications instead of integration. Likewise a much broader audience can be addressed for free.
Chapter 6

Evaluation

"The highest reward for man’s toil is not what he gets for it, but what he becomes by it."

John Ruskin (1819-1900)

To find out bottlenecks in the implementation of the gateway, several tests have been performed in a larger test bench. We show the different tests together with a short discussion of the test results. In Section 6.5 a short summary is presented.

6.1 Setup

In order to test the gateway latency and not the network latency, all the computers were attached on a 1Gbit full-duplex fast Ethernet Switch. The average latency of the network (round trip time) was 0.1ms.

- The client for the test beds was a Gentoo GNU Linux with sun-jre-1.5.0.17, 2 x 2.13GHz, 8GB RAM, 1Gbit NIC.

- To simulate different server conditions two servers were used:
  1. Gentoo GNU Linux with sun-jre-1.5.0.17, 1.1GHz, 2GB RAM, 1Gbit NIC.
  2. Debian GNU Linux with sun-jre-1.5.0, 400MHz Pentium-II, 32MB RAM, 100Mbit NIC.

The test client is much faster than the servers. This is on purpose as we want to test the server (and not the client). The slower server (10 years old, 400MHz) has approximately the same computational power, as many of todays embedded computers, but is much easier to program. The fast server (5 years old, 1100MHz) tries to model possible, future computational power for embedded systems and therefore acts like a "forecast model".
When measuring latencies between computers within a network, it is important that the clocks of all the computers are carefully synchronized. One of the computers therefore provided a NTPv4 time-server\(^1\). Within local networks this protocol is exact enough to keep the computers in sync (According the protocol, the drift is within 0.1ms on 100Mbit LAN).

In the eventing module (Section 4.3.4) a congestion control protocol is installed to avoid over-flooding of the gateway. All the parameters in the test benches have therefore been chosen in such a way that no congestion occurred.

For each test-run, we computed the ratios between two subsequent load profiles (eg. number of clients, number of devices, or number of events). The ratios as a dimension-less factor, give us hints, how the implementation scales with increased load.

## 6.2 Web Access

The test tries to evaluate the performance of the gateway, when many devices are attached and a varying number of clients access the presentation layer via HTTP.

At randomized intervals, virtual devices triggered maintenance routines on the gateway to simulate real device behavior. The test client started several concurrent threads that accessed the gateway and its devices randomly (in order to simulate real clients accessing the gateway). The test is split into two different settings:

- (I) 4000 devices, three test runs with 100 clients, 50 clients and 25 clients.
- (II) 1000 devices, three test runs with 1500 clients, 750 clients and 375 clients.

The test was performed against the 1100MHz server. Figure 6.1 shows the results as CDF together with the ratios.

**Discussion:** The results from (I) are surprising and encouraging. Even with a huge number of devices running internal maintenance routines and aliveness protocols, the gateway is capable to handling concurrent requests to a moderate number of Web clients with reasonable speed. With a constant number of devices, doubling the number of accessing clients results in approximately twice the latency. (II) shows that a large number of the requests can be satisfied within reasonable speed (eg. with 1500 clients 80% of the requests can be handled within approximately 3000ms). However for a few requests the delay is enormous (approximately 20000ms). On average the performance shows promising results and we can deduce, that the latency grows linear (doubling the number of clients results in twice the latency).

This test bench shows clearly the limits of our gateway implementation - although those limits are, from our point of view, surprisingly high. We conclude that an optimal system performance can be gained with a moderate number of accessing clients and attached devices.

\(^1\)http://de.wikipedia.org/wiki/Network_Time_Protocol
6.3 Event Delivery

To test the different aspects of the eventing, three completely different independent tests have been performed.

6.3.1 Event Subscription

The event subscription plays a crucial part in the whole eventing architecture. The benchmarks aim at testing the mechanism under different load profiles and tries to test whether the current implementation is flexible enough to support many clients at the same time (scalability).

Three runs with different numbers of concurrent clients (modeled by threads on the test machine) posted subscriptions at irregular intervals leading to a high churn rate (clients leaving or entering). For the 1.1GHz server the load profile was 325 clients, 650 clients and 1300 clients. For the 400MHz server the load profile was 100 clients, 200 clients and 400 clients. Figure 6.2 shows the measurements performed in a CDF together with their ratios.

Discussion: (I) With little subscribers the subscription process is very fast - doubling the number of subscribers from 100 to 200 leads to an increased processing time of almost 14 times on average. Another doubling increases the the processing time by four times. (II) The large number of subscribers allows to look at the quantitative scalability. As expected doubling the number of subscribers doubles the processing time for both tests (325 to 650 and 650 to 1300).

Figure 6.1: (I) Response time to deliver a request from a gateway with 4000 devices attached. 100 clients (triangle), 50 clients (circle) and 25 clients (box). (II) Response time to deliver a request from a gateway with 1000 devices attached. 1500 clients (triangle), 750 clients (circle) and 375 clients (box). (III) Ratios of the different measurements.
Figure 6.2: (I) Event subscription response times measured for 400 clients (triangle), 200 clients (circle) and 100 clients (box) on 400MHz server. (II) Event subscription response times measured for 1300 clients (triangle), 650 clients (circle) and 325 clients (box) on 1100MHz server. (III) Ratios of the different measurements for many subscriptions. The table on the left shows the results for the 400MHz server, the table on the right shows the results for the 1100MHz server.

The tests indicate that the registration process is scalable for a large number of concurrent clients (keep in mind, that there are websites with 100 requests a day). Especially with little subscribers, the system performs well. We think that in a real deployed case only a limited number of clients will access the gateway concurrently, therefore the current implementation is more than sufficient for most requirements.
6.3.2 Many Receivers

This test bench tries to benchmark the eventing module, when many clients installed a subscription for events and a event is triggered by one of the devices.

The test client started a event sink that received events on 200 different ports for the first test run, on 100 ports for the second test run and on 50 ports for the third test run. For each port a event subscription was posted to the gateway leading to 200, 100 or 50 registrations. The gateway then generated artificial events (containing the generation time) that were delivered to all the subscribed ports. The test client measured the arrival time and from that computed the delay for each arriving event. Figure 6.3 show the measurements performed in a CDF together with their ratios.

![Figure 6.3](image)

**Figure 6.3:** (I) Event delivery times measured for 200 subscribers (triangle), 100 subscribers (circle) and 50 subscribers (box) on 400MHz server. (II) Event delivery times measured for 200 subscribers (triangle), 100 subscribers (circle) and 50 subscribers (box) on 1100MHz server. (III) Ratios of the different measurements for many receivers. The table on the left shows the results for the 400MHz server, the table on the right shows the results for the 1100MHz server.

**Discussion:** The test results indicate that doubling the number of subscribers to a certain topic doubles the load while delivering the event. We deduce a linear scalability from that. Interesting to notice is that the slower server always delivered better minimum latencies than the faster server (upon repeated tests and after careful timer synchronizations). We don’t know the reason yet. Possible explanations may be, the scheduling algorithm of the operating system, or the implementation of the ethernet-card driver (bad handling of many parallel connections).

During this test, an unexpected failure behavior of the Restlet framework occurred that made it impossible to benchmark with more subscribers. The framework does not close used
network sockets properly (fast enough) leading sooner or later into a security limits violation by the Java VM (on both server and client). Although the tests in Section 6.3.1 have been performed with much more concurrent subscribers (many requests to one server socket) the same setup with many listening server sockets cannot be used. This test results should therefore be interpreted with a grain of salt.

6.3.3 Many Events

This benchmark tries to measure the gateway performance when many different events are triggered and have to be delivered to subscribers.

A special benchmark device generated different events (containing the time the event was fired on the gateway) at regular intervals. The test client received the events and computed the delay between arrival time and generation time. For the 400MHz server three test runs with 200 events, 100 events or 50 events have been executed. For the 1100MHz server 600 events, 300 events or 150 events have been generated. Figure 6.4 shows the results as CDF together with their ratios.

![Figure 6.4](image)

**Figure 6.4:** (I) Response time to deliver events to a client. 200 events (triangle), 100 events (circle) and 50 events (box) on 400MHz server. (II) Response time to deliver events to a client. 600 events (triangle), 300 events (circle) and 150 events (box) on 1100MHz server. (III) Ratios of the different measurements for many events. The table on the left shows the results for the 400MHz server, the table on the right shows the results for the 1100MHz server.

**Discussion:** The tests in (I) and (II) both indicate, that the event delivery mechanism scales linearly with the number of events. Doubling the number of events in both test sets resulted in approximately twice the response time. The maximum delay is well bounded even for many
6.3. Event Delivery

We hope that the number of events handled gracefully by the gateway suffices our requirements. However, further research can be done in lowering the minimum response time that is, for our taste, quite high (around 80ms). The current implementation uses a HTTP based event delivery mechanism on top of the restlet HTTP connector. A possible solution might be to exchange Restlet by a faster NIO-based connector (Grizzly, Jetty, ...). NIO\textsuperscript{2} is the acronym for \textit{New Input/Output} and in essence tries to speed up transmissions by exchanging synchronous, blocking streams with asynchronous, non-blocking buffers.

6.3.4 Many Devices

The benchmark tries to simulate the scenario, when there are many different devices attached to the gateway. Those devices generate events with random wait times between two subsequent events in the interval 1000ms to 5000ms.

The benchmark was run on the 1100MHz server with 200 devices, 100 devices and 50 devices attached. From the event arrival and generation time, the delivery delay was computed. Figure 6.5 shows the results together with the ratio as a CDF.

![Figure 6.5](image)

<table>
<thead>
<tr>
<th></th>
<th>50 d.</th>
<th>100 d.</th>
<th>200 d.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>29ms</td>
<td>30ms</td>
<td>418ms</td>
</tr>
<tr>
<td>Max</td>
<td>127ms</td>
<td>284ms</td>
<td>1087ms</td>
</tr>
<tr>
<td>RT 50%</td>
<td>41ms</td>
<td>124ms</td>
<td>641ms</td>
</tr>
<tr>
<td>RT 80%</td>
<td>71ms</td>
<td>174ms</td>
<td>733ms</td>
</tr>
<tr>
<td>Mean</td>
<td>52ms</td>
<td>125ms</td>
<td>699ms</td>
</tr>
</tbody>
</table>

**Figure 6.5:** (I) Response time to deliver events to a client with many devices attached. 200 devices (triangle), 100 devices (circle) and 50 devices (box) on a 1100MHz server. (II) Ratios of the different measurements for many events.

**Discussion:** The results in (I) indicate that for a moderate amount of devices (50 and 100), the gateway is able to handle the events and the required maintenance work. The ratios in (II) show that doubling the number of devices results in a approximate doubling of the latencies. With 200 devices the performance drops significantly. The ratios grow to four times the latency.

There are two factors that might be responsible for the performance loss. First, we guess that the current implementation is not able to handle many concurrent requests to the event module gracefully, and Second, the HTTP based event distribution mechanism takes to long

\textsuperscript{2}http://www.jcp.org/en/jsr/detail?id=203
for the connection setup and payload delivery. A short term solution to the second issue could be to collect several events, with the same target, and send them as one payload (With the risk to introduce artificial delay due to the grouping).

### 6.4 Caching

When a client requests a resource on the gateway in a format other than XML, then the representation has to be compiled/translated into the target format first. This compilation procedure takes time and is memory and CPU intensive. The idea is to keep the compiled version in a local cache in order for subsequent requests to the same resource to be delivered from the cache, without the need for the compilation step.

To test caching, 100 sequential requests were performed and the response-time (RT) was measured for each request. Figure 6.6 shows the measurements performed in a cumulative density function (CDF) with cache enabled versus cache disabled.

![Figure 6.6](image)

<table>
<thead>
<tr>
<th></th>
<th>no cache</th>
<th>with cache</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>93ms</td>
<td>8ms</td>
<td>11.62</td>
</tr>
<tr>
<td>Max</td>
<td>605ms</td>
<td>85ms</td>
<td>7.11</td>
</tr>
<tr>
<td>RT 50%</td>
<td>135ms</td>
<td>11ms</td>
<td>12.27</td>
</tr>
<tr>
<td>RT 80%</td>
<td>191ms</td>
<td>50ms</td>
<td>3.82</td>
</tr>
<tr>
<td>Mean</td>
<td>180ms</td>
<td>23ms</td>
<td>7.82</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>no cache</th>
<th>with cache</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>27ms</td>
<td>4ms</td>
<td>6.75</td>
</tr>
<tr>
<td>Max</td>
<td>277ms</td>
<td>78ms</td>
<td>3.55</td>
</tr>
<tr>
<td>RT 50%</td>
<td>40ms</td>
<td>4ms</td>
<td>10</td>
</tr>
<tr>
<td>RT 80%</td>
<td>70ms</td>
<td>9ms</td>
<td>7.78</td>
</tr>
<tr>
<td>Mean</td>
<td>68ms</td>
<td>10ms</td>
<td>6.8</td>
</tr>
</tbody>
</table>

Figure 6.6: (I) Response times measured with cache (triangle) and without cache (circle) on 400MHz server. (II) Response times measured with cache (triangle) and without cache (circle) on 1.1GHz server.

**Discussion:** (I) 50% of the requests (RT 50%) have been delivered within 11ms with cache versus 135ms without cache. This equals to a speedup of approximately 12 times. For 80% of the requests (RT 80%) still a speed up of approximately 4 times could be gained (50ms versus 191ms). On average the cached version was approximately 8 times faster than the non cached version. (II) 50% of the requests (RT 50%) have been delivered within 4ms with cache versus 40ms without cache. This equals to a speedup of 10 times. For 80% of the requests (RT 80%) a speed up of approximately 8 times could be gained (9ms versus 70ms). On average, the cached version was approximately 7 times faster than the non cached version.

The measurements give rise to the conclusion that caching can be used to reduce the response time of the gateway. This is especially true for reading data from physical devices that seldomly change their state. However, in the case where the cache entry is invalidated right
after generation by a state change on the device, the cache maintenance overhead could degrade overall performance of the gateway.

Another advantage when using cached values is that the representation of the device can be generated directly (and transparently) on the gateway without communication to the real device. This lowers the overall load for the physical device even further, thus increase the battery lifetime of sensor nodes.

6.5 Summary

The performance results reveal surprising and interesting results that motivate further optimizations and enhancements.

In the stress test (Section 6.2) we tried to find the limits of the implementation under extreme conditions. The results show that the gateway scales with the hardware, or formulated more sloppy: for a given gateway, more computational power allows to attach more devices.

Our eventing module seems to behave as expected. The subscription process and the event delivery scale linearly with the number of clients/events. Especially when there are only a few clients, we measured low delays between the time, the event was triggered and the time when the event arrived at the consumer (On average between 100ms and 300ms).

Caching (Section 6.4) indicates that on average, a caching mechanism can reduce response time and helps omitting repeated computations over and over again. We can save spare resources by introducing a tiny bit of overhead to the gateway (the caching maintenance work). However, the presented caching-test benchmarks only the caching-properties of the presentation layer and we therefore need to formulate a more elaborate test scenario (eg. with the help of the contiki/TinyOS subsystem (Section 4.4.2)).

We envision the runtime environment for our gateway to be installed mainly on home routers where the number of attached devices is moderate (eg. a WiFi, mobile phone, ...). For these appliances the performance of the system should therefore be sufficient.
Chapter 7

Discussion and Future Work

"Art is never finished, only abandoned."

Leonardo da Vinci

We developed a stand-alone server component, that runs on resources constrained computers with limitations on CPU power and memory. The performance evaluations (Chapter 6) show promising results even on a 10 years old computer, encouraging us to further investigate work. The successful deployment of the prototypes (Chapter 5) on a Linksys NSLU (133MHz CPU with 16MB RAM) indicates, that the smart gateway can even operate on embedded home routers. We look forward to equip wireless routers or modems with our gateway software.

With the device abstraction (Section 4.4) we demonstrated a powerful mechanism that allows the integration of any kinds of devices into the gateway. The abstraction makes no prior assumptions on the type of the devices (be it a physical or a virtual device), therefore enabling the integration of different communication protocols/paradigms. With the integration of the Contiki/TinyOS subsystem (Section 4.4.2) we can show, that even synchronous and asynchronous subsystems can co-exist and interact due to the simplicity and flexibility of the chosen device abstraction.

The gateway takes into account, that devices can be mobile or erroneous and therefore by nature join/leave at random. With the error masking algorithms (Section 4.3.3), missing devices get automatically removed from the devices list, requiring no administrative work on the user.

We model every device as a Web resource with different capabilities that can be exposed via XML, plain-text or HTML to the clients. This helps at runtime, to perform security tests on the requests from the Web. At the same time, clients can use the information to search and lookup devices that match certain criterias, enabling higher-level services to be built on top of the gateway infrastructure. We therefore provide reusable, extensible, and "mashupable"
building blocks for a real *Web of Things*.

We propose a simple, yet powerful, asynchronous eventing mechanism (Section 4.3.4) supporting higher-level events, that can be used in business application scenarios. Using Web technologies for the event propagation and presentation, we provide device-level events to a much broader audience on the Web. We therefore support straight-forward integration of sensor-events into existing Web applications.

Thanks to the RESTful design, complex integration tasks can be broken into smaller and simpler ones, that can be solved efficiently and quickly (Section 5.1). Linking the different building blocks reduces integration to no more than patching several URIs together. Based on our experience, a RESTful architecture helps building loosely coupled, reusable, distributed applications that are scalable, evolvable, and extensible.

### 7.1 Future Work

We developed and tested the gateway in a limited environment, with only a few different devices attached. Large scale deployments with many heterogeneous devices (real and simulated) have to show, whether our system is capable of dealing with a large number of such citizens, interacting and (re)using gateway components concurrently. Extended performance and usability tests have to discover and address factors, that affect the overall score of the system as a whole.

The cornerstone of the gateway architecture have been set and a few architectural decisions have been made. However many unaddressed issues remain to be solved. We present a brief account of interesting topics for future exploration and research:

**Discovery** Our implementation assumes that administrators carefully prepare the runtime environments for the gateway by providing necessary input, such as the parent gateway in a distributed setup. This approach might fit as long as we are considering a small setup with a limited number of gateways interacting, but for larger deployments a more automated solution has to be found (eg. DPWS, Bonjour).

**Load balancing** The success of todays Web lies within its support to effectively distribute the load onto several servers - transparently for Web clients. The gateway tries to address load distribution through the tree structure, where administrators can dynamically inject new server instances. New devices can then register at the new gateways. However an automatic transfer from a highly loaded gateway to a less crowded one is missing.

**Security** Although HTTP addresses security between server and client by using HTTPS, security for a complete distributed architecture using Web technology remains a central issue. By design, HTTPS is "only" capable to secure the channel between two direct endpoints and if there are intermediate routing hops (eg. an intermediate gateway), each hop has to be trusted for a secure channel. To provide end-to-end security the gateway needs a technology comparable to WS-security on top of the existing architecture.

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

Search  The presented search infrastructure lacks in terms of efficiency and ease of use. Applications might need to search in the direct neighborhood for devices and therefore need a mechanism that controls the search radius (there’s no need to search through the whole tree when the neighbors are sufficient).

Future work might integrate existing Web-search algorithms directly into the Web-architecture of the gateway.

Enterprise Eventing  The current implementation supports a Web-based event management that offers great flexibility for a lightweight event interaction using Web standards. A future extension might include a subset of a XMPP server or even implement a lightweight MQTT broker in Java, to extend the event mechanism to a much broader audience.

Auto assembly  An interesting feature is "auto-assembling" device drivers, that, using some kind of user defined rules, automatically try to find gateway components (first locally, then distributed) to build mash-ups. For example, a client can define a weather forecast device just by instructing the gateway to ”patch” a humidity, a pressure and a temperature sensor together. The system will then take care of the implementation and the connections between the different sensors - As opposed to manually hand-code the driver.

Mobile Code  When a new device registers at the gateway we assume that the gateway already maintains a copy of the device driver required to interact with the physical device. In a worldwide setting with a vast number of devices, all using different device drivers, this is not feasible. Mobile code refers to code that can be passed and executed between different applications, could be a promising solution to circumvent this bottleneck.

7.2 Conclusion

The smart gateway could be a central element to build a loosely coupled architecture built upon Web technologies. Heterogeneous physical and virtual objects are integrated into a network of inter-operable components, that can be found and used by both machines and humans. By using a resource oriented approach, rather than WS-*, we reuse existing and well established standards which results in a flexible and scalable application that can be integrated seamlessly into the existing Web. A careful design following the REST principles ensures that the resources represented on the gateway maintain a narrow and clean interface that can be explored through existing Web browsers. Therefore it lowers the barrier for people to use and develop applications on a much wider ecosystem of interconnected devices. With the hierarchical concept of symbolic locations, basic location-aware applications can be implemented for physical objects, using design principles from the Web (URI). In the implementation of the location-aware energy monitor, we have shown that even our simple location concept suffices to break complex tasks into smaller blocks that can be solved and handled by resource constrained embedded devices (SunSpots). Our high-level, HTTP-based event architecture provides a common ground for all kinds of device-level events, masking away the complexity of the underlying technology. Together with the gateway, whole sensor networks can be "bridged" to the Web, by providing
the gateway with device-level events from the sensor nodes.

I believe that the vision of a *Web of Things* can become reality. The smart gateway addresses a few of the many problems on the road towards our goal, therefore helping to build the future of distributed sensor-based computing.

### 7.3 Acknowledgments

I would like to say thanks: to Vlad Trifa for sharing his creative mind with me and for always giving me insight and guidance whenever I was stuck like a donkey in front of a big mountain, to Michael Hänni for the countless discussions about how to choose the right technology, to Andreas Kamilaris for describing me the nice weather in Greece and the funny hours in IFW-laptop-room, to Dominique Guinard for his enthusiasm when playing around with energy-meters and to all the others that helped me to finish this work successfully.
Appendix A

Maven Cheat Sheet

The gateway project uses Apache Maven 2\(^1\) to maintain the source code, to build the runtimes and to generate project reports (Javadoc, NCSS, PMD, ...). This guide shall give a short overview to the most important commands available.

The guide assumes that you have a running Java environment (\(\geq 1.5\)), a running copy of Maven (The Website has an excellent guide how to install Maven for almost any operating system) and a checkout from the gateway source code.

If you are not familiar with Maven please read the whole guide before you start. Then proceed step by step from the start.

Cheet Sheet The most important commands. For more details read the whole document.

- **mvn clean**: Clear temporary folders.
- **mvn compile**: Compile the source code.
- **mvn package**: Compile source code and package jar files.
- **mvn install**: Package and install into local repository.
- **mvn site:stage -DstagingDirectory=DIR**: Compile Website.
- **mvn eclipse:eclipse**: Generate eclipse projects.
- **mvn eclipse:clean**: Remove eclipse projects.
- **MAVEN\_COMMAND -o**: Execute in offline mode (faster).
- **MAVEN\_COMMAND -U**: Force Maven plugin updates.

Verify Maven Please run the command *mvn -version* before you proceed. Make sure that it generates a console prompt comparable to the one below:

```bash
$ mvn -version
Maven version: 2.0.9
Java version: 1.6.0_11
OS name: "linux" version: "2.6.25-hardened-r10" arch: "i386" Family: "unix"
```

\(^1\)http://maven.apache.org/
Compilation and Packaging  Open a console and change to the top directory where you have the source code.

Example:
$ cd ~/workspace-master/gateway

Execute `mvn compile` to compile the sources, `mvn package` to package the compiled classes into jar files together with their dependencies or `mvn install` to install the packages into the local Maven repository (Usually stored in your home directory under `.m2/repository`). If you run Maven for the first time run `mvn install`.

If everything compiles/installs well maven will prompt you the success:

```
[INFO] ---------------------------------------------------------
[INFO] ---------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ---------------------------------------------------------
[INFO] Total time: 43 seconds
[INFO] Final Memory: 35M/396M
[INFO] ---------------------------------------------------------

ATTENTION: It is possible that Maven does not find some needed dependencies (MQTT, OSC, Jade). This is because those jar-files are not hosted on any Maven repository. You need to install these files manually. Download the required jar-files to a location on your computer (Example: /tmp) and execute the command listed to the console. Maven copies the jar-files into its local repository and from then on you will no more be bothered about those missing dependencies.

Example how the missing JadeTools can be installed manually:
$ mvn install:install-file -DgroupId=com.tilab.jade -DartifactId=jadeTools \
   -Dversion=3.6-SNAPSHOT -Dpackaging=jar -Dfile=/tmp/jadeTools.jar

NOTICE: `install` implies `package` and `package` implies `compile`.

Clean  During the compilation and installation phases Maven generates a subfolder `target` with the compiled classes and temporary files. You can clean out those folders automatically:

$ mvn clean

Website  Maven can auto-generate a project Website from the whole source trunk. This Website includes an aggregated Javadoc containing all the sub-projects. For each sub-project additionally a sub-Website is generated that provides useful information about different code metrics (NCSS, PMD, Duplicate code, Tagging (FIXME, TODO, ...)).

Example: Compile the Website and store the result to the folder "/tmp/":
$ mvn site:stage -DstagingDirectory="/tmp"

If Maven complains about the missing `site`-plugin append `-U` to the end of the command (instruct Maven to search for plugin updates on the Maven repository).
**Eclipse integration**  From the `pom.xml` Maven is able to generate Eclipse projects for you. The projects can then be imported by Eclipse. After the import Eclipse will complain about the missing classpath variable `M2_REPO` and the projects will not compile at all. Open the global Eclipse preferences dialog, search for `CLASSPATH` and add a new classpath variable `M2_REPO` that points to the local Maven repository (Usually stored in your home directory under `.m2/repository`). Refresh the workspace.

Generate eclipse projects:

$ mvn eclipse:eclipse

Remove the eclipse projects:

$ mvn eclipse:clean

If Maven complains about the missing `eclipse-plugin` append `-U` to the end of the command (instruct Maven to search for plugin updates on the Maven repository).
Appendix B

Build Firefox Extension

There are many guides available in the Web that give you hints about how to build new Mozilla plugins and how you can compile the Mozilla suite from source. Many of them are badly written, deprecated or just plain wrong (with a few good exceptions!). If you want to create a new plugin/extension on your own i recommend to read the following tutorials/documentations carefully before you start.

- [https://developer.mozilla.org/En/Simple_build](https://developer.mozilla.org/En/Simple_build):
  Short tutorial for the impatient how to setup a simple build environment. If you just want to compile the firefox plugin for the gateway, use this tutorial to prepare your build environment.

  Complete tutorial/documentation how to setup a complete build environment.

- [http://www.iosart.com/firefox/xpcom/](http://www.iosart.com/firefox/xpcom/):
  Good starting point for first experiences. For advanced building and packaging not suitable.

- [https://developer.mozilla.org/en/Creating_Custom_Firefox_Extensions_with_the_Mozilla_Build_System](https://developer.mozilla.org/en/Creating_Custom_Firefox_Extensions_with_the_Mozilla_Build_System):
  EXCELLENT tutorial how to setup your build environment and how to start a new plugin/extension. Please notice that i used this tutorial as basis to setup my build environment.

Additional references:

  Reference API for the XUL language.

  Reference API for the Netscape Portable Runtime (NSPR).

  XPCOM Documentation and API Reference.

  If you plan to extend the gateway firefox extension read this tutorial first!
**Preparations**  Before you can start compiling the gateway firefox extension, you should first have a running build environment. I recommend the first tutorial from the list above. When you are compiling for Windows, installing the correct libraries is very important, so make sure that you install **ALL** the listed requirements! Below you can find my `.mozconfig`. I recommend you to copy this file as it is!

```
$topsrcdir/browser/config/mozconfig
mk_add_options MOZ_OBJDIR=@TOPSRCDIR@/ff-opt
ac_add_options --disable-tests
ac_add_options --enable-extensions=default
```

Make sure that you can compile the whole source trunk successfully.

```
# compiles the mozilla source code
$ make -f client.mk build
```

**Compile the gateway firefox extension**  Download the source code of the gateway firefox extension and copy the folder `gatewaydiscovery` into the folder `MOZILLA_SRC_DIR/extensions`. After that you need to tell the build system to compile your extension together with the mozilla source code. Modify the `.mozconfig` as following (Notice the additional extension added):

```
$topsrcdir/browser/config/mozconfig
mk_add_options MOZ_OBJDIR=@TOPSRCDIR@/ff-opt
ac_add_options --disable-tests
ac_add_options --enable-extensions=default,gatewaydiscovery
```

Now run `make -f client.mk build`. If you compiled the sources already in advance, this task will take approximately 10 seconds to finish (Otherwise 10 minutes).

The `.mozconfig` specified the object dir to be `ff-opt`. The build system will store the xpi (the packaged mozilla extension) into this folder under `dist/xpi-stage`. Aside the xpi you will additionally find an exploded copy of the discovery extension.
Appendix C

Gateway Debug Tools

During the implementation of the gateway a few debugging helpers have been developed. Those tools will shortly be presented together with a few useful debug tools from third party developers.

C.1 My own tools

All my tools have been grouped into gateway-core. You will find them in the Java package ch.ethz.inf.vs.gateway.utils.

EventReceiver is a basic event sink receiving and displaying gateway events. At startup the sink accepts an integer value specifying the port where to listen for events. The sink always displays the newest event at the top.

default sink URI: http://localhost:9994/_eventing/event

PutEvent helps posting events onto a gateway.

The tool allows to specify the keyword, the gateway where to put the event, the lease-time for the event and the callback where to deliver the events to.

RequestContentByMime is comparable to Poster (C.2). As an improvement the tool fetches several copies of the target URI in the desired MIME-types and displays them (as opposed to one).

TikitagSim simulates a physical Tikitag RFID reader. The tags presented to the reader simulator are delivered via OSC to the Tikitag gateway driver. At startup one can specify the IP
and the port of the target driver on a gateway together with the name of the local user interface (this helps to distinguish several copies of the simulator). To enhance usability the simulator remembers all the tags in a drop-down.

RemoteControlUI allows to send multi-cast events to the gateway (4.5). This greatly simplifies live when debugging the multi-cast interface of the gateway. As a nice feature you can shut down the gateway instance remotely by sending the correct access token to the gateway.

C.2 Third party tools

Poster is a great Mozilla extension that gives your Web browser the capability to perform real RESTful interaction. Handcrafted HTTP requests (header, body, parameters) can be sent to any kind of Web-service.


Firebug is a Mozilla extension allowing on site HTML, JavaScript and CSS editing and debugging. Fast JavaScript debugging made easy...


HTML Validator is a Mozilla extension that performs on the fly HTML validation. HTML and JavaScript errors will be highlighted and sample fixes are presented.

Appendix D

Smart Gateway as Stand-alone Dependency

During the integration with the smart simulator, a major "issue" with the gateway became visible. As we modelled the Core as a singleton, it is not possible to start several instances of the gateway out of the same process context (eg. by executing a dedicated thread). We therefore need to run the different gateway instances as sub-processes, using the Java runtime environment.

To simplify the delicate handling of the in- and out-streams, we implemented a wrapper, that maintains the starting/stopping of the underlying gateway process. Listing D.1 shows the partial code of the server wrapper. The code below shows, how you can start the server wrapper:

```java
public class MyServerExecutor implements Observer {

    public void myServerStartMethod() {
        String identifier = "myServerWrapperIdentifier";
        ServerWrapper sw = new ServerWrapper(identifier);
        sw.addObserver(this);

        URL myParent = new URL("http://myParentIP:ParentPort");
        String myIP = "localhost";
        int myPort = 8080;
        String myGatewayName = "myGateway";

        sw.execute(
            GATEWAY_JAR_LOCATION , myParent ,
            myIP , myPort , myGatewayName
        );
    }

    public void update(Observable o, Object arg) {
        if (o instanceof ServerWrapper) {
            ServerWrapper sw = (ServerWrapper) o;
            if (sw.isError()) {
                ...
            } else {
                ...
            }
        }
    }
}
```
Listing D.1: ServerWrapper.java: Wrapper to maintain gateway processes.

```java
package ch.ethz.inf.vs.gateway.server.wrapper;

/**
 * little helper around the server program that allows the server
 * to be run from within another application more conveniently.
 * If you register as an observer then you will be notified when
 * the application crashed or finished.
 * @author sawielan
 */
public class ServerWrapper extends Observable implements Runnable {

    ... CODE THAT WAS OMITTED ...

    /**
     * default constructor. ONLY use this constructor if you know
     * what you are doing!
     */
    public ServerWrapper() {
    }

    /**
     * creates a server wrapper.
     * @param id the identifier that identifies this server wrapper.
     */
    public ServerWrapper(String id) {
        this.id = id;
    }

    ... CODE THAT WAS OMITTED ...

    /**
     * execute the server.
     * @param jarLocation the location where we find the server.
     * @param parent the location of the parent.
     * @param myIP the ip where to listen.
     * @param port the port where to bind.
     * @param myName the name of the server instance.
     * @throws IOException upon error...
     */
    public void execute(String jarLocation, URL parent, String myIP, int port, String myName) throws IOException {
        String pString = "null";
        if (null != parent) {
```

Smart Gateway as Stand-alone Dependency
pString = parent.toString();
}
long t = 0;
synchronized (token) {
    t = token ++;
    accessToken = String.format("%d", t);
}

cmd = String.format("java -jar %s -g %s -a %s -p %d -n %s -t %d",
    jarLocation,
    pString,
    myIP,
    port,
    myName,
    t
);

if (sc) {
    terminal = new Terminal(id);
}
    new Thread(this).start();
}

/**
* stops a server instance by sending the quit command.
* @throws IOException upon error...
*/
public void terminate() throws IOException {
    MulticastSocket socket = new MulticastSocket();
    BroadCastRequest r = new BroadCastRequest(BroadCastRequest.REMOTE_CONTROL_REQUEST);
    String command = accessToken + ", quit";
    r.setBytes(command.getBytes());
    byte[] b = r.encode();
    DatagramPacket packet = new DatagramPacket(b,
        b.length,
        InetAddress.getByName(Constants.GROUP),
        Constants.PORT);
    socket.send(packet);
    socket.close();
}

/**
* execute the command line.
*/
public void run() {
    try {
        Process p = Runtime.getRuntime().exec(cmd);
try {
    BufferedReader inputStreamReader = new BufferedReader(  
        new InputStreamReader(p.getInputStream()));
    BufferedReader errStreamReader = new BufferedReader(  
        new InputStreamReader(p.getErrorStream()));

    StringBuffer output = new StringBuffer();
    StringBuffer error = new StringBuffer();

    for (String line; (line = inputStreamReader.readLine()) != null;) {
        output.append(line);
        if (null != terminal) {
            terminal.append(line);
        }
    }
    for (String line; (line = errStreamReader.readLine()) != null;) {
        error.append(line);
    }
    if (error.length() > 0) {
        this.errorString = error.toString();
        this.error = true;
    }

    p.waitFor();

    } catch (InterruptedException e) {
        this.error = true;
        e.printStackTrace();
    } catch (IOException e) {
        this.error = true;
        e.printStackTrace();
    }

    this.finished = true;
    setChanged();
    notifyObservers();
}

... CODE THAT WAS OMITTED ...
Source Code

E.1 File Finder

The file finder is a helper utility, that allows you to search for a given marker in the whole classpath. All the occurrences are stored into a linked list of URL.

Listing E.1: FileFinder.java: Helper utility that searches for all occurrences of a given marker file in the classpath.

```java
package ch.ethz.inf.vs.gateway.commons.finder;

import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;
import java.util.LinkedList;
import org.apache.log4j.Logger;

/**
 * you can pass the file finder a filename and the finder will
 * search all files in the classpath that conform to this filename.
 * @author sawielan
 *
 * /
 * public class FileFinder {

    /** Log4j instance. */
    private static Logger log = Logger.getLogger(FileFinder.class);

    /** the list holding all the files found. */
    private LinkedList<URL> list = null;

    /**
     * constructor. nothing more, nothing less.
     */
    public FileFinder() {
    }
```
/**
 * searches for all occurrences of files with name fileName.
 * @param fileName the name to search for.
 * @return a linked list holding all conformant files.
 */
public LinkedList<URL> find(String fileName) {
    log.debug("Searching for file(s): " + fileName);

    list = new LinkedList<URL>();

    try {
        ClassLoader cl = Thread.currentThread().getContextClassLoader();
        Enumeration<URL> urls = cl.getResources(fileName);

        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            list.add(url);
            log.debug(String.format("found possible file: %s", url.toString()));
        }
    } catch (IOException e) {
        log.error("caught exception when searching for files");
        e.printStackTrace();
    }

    log.debug(String.format("found %d possible file(s)", list.size()));
    return list;
}

/**
 * give the user the possibility to retrieve the search results without the need for searching.
 * @return the search results, null if no search has been done previously.
 */
public LinkedList<URL> get() {
    if (list == null) {
        log.debug("warning, returning null pointer as no search has " +
                "been performed before get()");
    }
    return list;
}
E.2 Device Driver Example - Redirector Driver

The redirector device driver redirects a request to another Web resource transparently for the client. The URL for the remote resource has to be passed at startup of the device driver.

Listing E.2: Redirect.java: Implementation of a device driver.

```java
package ch.ethz.inf.vs.gateway.plugin.devices.drivers;

import java.io.UnsupportedEncodingException;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLDecoder;
import java.util.Map;

import org.apache.log4j.Logger;
import org.restlet.Client;
import org.restlet.data.Protocol;
import org.restlet.data.Request;
import org.restlet.data.Response;

import ch.ethz.inf.vs.gateway.Core;
import ch.ethz.inf.vs.gateway.plugin.devices.Device;

/**
 * redirects a call to another location transparently for the client.
 * @author sawielan
 */

public class Redirect extends Device {

    /** the name of the URL parameter in the parameters map. */
    public static final String URL_PARAM = "url";

    /** the client to use for the HTTP request/redirect. */
    private Client client = null;

    /** the name of the device. */
    protected String name;

    /** the URL where to redirect to. */
    protected URL url;

    /** log4j instance. */
    private static Logger log = Logger.getLogger(Redirect.class);

    public Redirect(String name) {
        super(name);
        this.name = name;
    }

    @Override
```
public String getDeviceName() {
    return name;
}

@Override
public String handle(Response response, Request request) {
    // create a new instance of the client.
    if (null == client) {
        client = new Client(
            Core.getInstance().getServer().getRestApplication() .getContext().createChildContext(),
            Protocol.HTTP
            );
    }

    // get the remaining part
    String str = request.getResourceRef().getRemainingPart();
    Request newRequest = new Request(request.getMethod(),
        url.toString() + str);
    log.debug(String.format("make call to %s", url + str));

    newRequest.setReferrerRef(Core.getInstance().getServer().
        getHostURI().toString());
    newRequest.setClientInfo(request.getClientInfo());
    client.setConnectTimeout(10000);
    client.handle(newRequest, response);
    return null;
}

@Override
public void init(Map<String, Object> params) {
    try {
        String str = (String) params.get(URL_PARAM);
        str = URLDecoder.decode(str, "UTF-8");
        this.url = new URL(str);
        //this.url = new URL((String)params.get(URL_PARAM));
        log.debug("will use URL: " + url.toString());
    } catch (MalformedURLException e) {
        log.debug("url is malformed: " + url.toString());
    } catch (UnsupportedEncodingException e) {
        log.debug("url is not correctly encoded: " + url.toString());
    }
}

@Override
public boolean isAlive() {
    return true;
}
### E.2. Device Driver Example - Redirector Driver

```java
@Override
public void setAlive() {
}

@Override
public void setDeviceName(String deviceName) {
    this.name = deviceName;
}
```

E.3 Asynchronous/Synchronous Execution Synchronization

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

Listing E.3: Implementation of the synchronizer between asynchronous and synchronous behavior.

```java
package ch.ethz.inf.vs.gateway.plugin.devices.drivers.motes;

public class MoteDevice extends Device {

    // locking stuff...
    /** the next free token. DONT MODIFY DIRECTLY, USE getToken() */
    private static Long msgToken = new Long(1);

    /**
     * @return a token for the message id.
     */
    public static synchronized long getToken() {
        synchronized (msgToken) {
            return msgToken++;  
        }
    }

    /**
     * dispatch a response to the synchronizer.
     * @param r the low level response.
     */
    public static synchronized void dispatchResponse(Response r) {
        AsyncToSync lock = synchronizer.get(r.getRequestID());
        if (null == lock) {
            return;
        }

        synchronized (lock) {
            lock.setResponse(r);
            lock.notifyAll();
        }
    }

    /**
     * helper class to synchronize the async communication to contiki/tinyos.
     * @author sawielan
     */
    public class AsyncToSync {

        /** my token. */
        private final long token;

        /** the response onto my request. */
```
private Response response = null;

/**
 * constructor.
 */
public AsyncToSync() {
    token = MoteDevice.getToken();
}

/**
 * @return my token.
 */
public long getToken() {
    return token;
}

/**
 * @return the response
 */
public Response getResponse() {
    return response;
}

/**
 * @param response the response to set
 */
public void setResponse(Response response) {
    this.response = response;
}

/** a hash map containing the synchronizer objects. */
private static Map<Long, AsyncToSync> synchronizer =
        new ConcurrentHashMap<Long, AsyncToSync>();

// \ end of synchronizing

... CODE THAT WAS OMITTED ...

/* Handles Responses from Smart Device by forwarding them to the appropriate Internet Client who made the Request */
public void handleResponse(Response r) {
    System.out.println("Handling normal Response for service:"+r.getServiceName());
    dispatchResponse(r);
}

/* adds Request r in Request Message Queue */
public void addRequest(Request r) {
    this.msgQueue.addRequestMessage(r);
}
@Override
public String handle(org.restlet.data.Response response,
         org.restlet.data.Request request) {

    ... CODE THAT WAS OMITTED ...

    Request re = new Request(deviceID,resourceName,method,params,
        values,false,0);
    Response r = waitSynchronous(re);
    if (null == r) {
        response.setEntity(new StringRepresentation(Constants.NACK));
    } else {
        log.debug(r.getResult());
        response.setEntity(
            new StringRepresentation(r.getResult().toString()));
    }
    return null;
}

public Response waitSynchronous(Request request) {
    // handle a request with lock wait...
    AsyncToSync lock = new AsyncToSync();
    synchronizer.put(lock.getToken(), lock);
    try {
        log.debug("wait on lock.");
        request.setRequestID(lock.getToken());
        addRequest(request);

        synchronized (lock) {
            lock.wait();
        }
        log.debug("leaving lock");
    } catch (Exception e) {
        e.printStackTrace();
        synchronizer.remove(lock);
    }
    log.debug("left lock");
    synchronizer.remove(lock.getToken());
    return lock.getResponse();
}

    ... CODE THAT WAS OMITTED ...
}
Appendix F

Class Diagrams

F.1 Class Diagram - Plugin Management

Figure F.1: Class diagram for the plugin management.
F.2 Class Diagram - Translator Management

Figure F.2: Class diagram for the translator management.
Figure F.3: Class diagram for the core and the most important modules provided by the core.
Figure F.4: The diagram shows the class hierarchy of a device driver together with its control structures. On the bottom five different device driver implementations are plotted.
Figure F.5: Class diagram for the eventing plugin (at the bottom) and its representation in the presentation layer (on top).
XSD-Schema for XML Device Representation

Figure G.1: Graphical representation of the devices representation Ontology.
Listing G.1: XSD Ontology for the devices representation.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:element name="device">
    <xs:annotation>
      <xs:documentation>
        A device describes either a gateway device or a physical device attached at a gateway.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element ref="location"/>
        <xs:element ref="devices"/>
        <xs:element ref="keywords"/>
        <xs:element ref="resources"/>
        <xs:element ref="information" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="devices">
    <xs:annotation>
      <xs:documentation>
        wraps all devices (both locally attached and gateways).
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gateway" minOccurs="0" maxOccurs="unbounded" type="xs:string"/>
        <xs:element ref="local"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="gateway">
    <xs:annotation>
      <xs:documentation>
        wraps all gateway devices.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="device" minOccurs="0" maxOccurs="unbounded" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="local">
```
<xs:annotation>
  <xs:documentation>
    wraps all locally attached devices.
  </xs:documentation>
</xs:annotation>
<xs:complexType>
  <xs:sequence>
    <xs:element name="device" minOccurs="0" maxOccurs="unbounded" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="keywords">
  <xs:annotation>
    <xs:documentation>
      wraps all the keywords.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="keyword" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="location">
  <xs:annotation>
    <xs:documentation>
      wraps all different locations.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="symbolical"/>
      <xs:element ref="gis" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="symbolical">
  <xs:annotation>
    <xs:documentation>
      describes the symbolical location of the device.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="current" type="xs:string"/>
      <xs:element name="parent" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="gis"/>
<xs:annotation>
  <xs:documentation>
    describes the geo location of the device.
  </xs:documentation>
</xs:annotation>

<xs:complexType>
  <xs:sequence>
    <xs:element name="elevation" type="xs:string"/>
    <xs:element name="longitude" type="xs:string"/>
    <xs:element name="latitude" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="resources">
  <xs:annotation>
    <xs:documentation>
      wraps all the resources available by the device.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="resource" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="resource">
  <xs:annotation>
    <xs:documentation>
      a resource is a capability of a device (eg. getTemperature).
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element ref="methods"/>
      <xs:element ref="mimetypes"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="methods">
  <xs:annotation>
    <xs:documentation>
      how a resource can be called (eg POST or GET).
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="method" type="xs:string"/>
<xs:element name="mimetypes">
  <xs:annotation>
    <xs:documentation>
      wraps all the mime types.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="mimetype" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:complexType>
</xs:element>

<xs:element name="information">
  <xs:annotation>
    <xs:documentation>
      describes the device specifics
      (eg vendor, hardware information etc..).
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vendor"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
**Benchmark Configuration Files**

**H.1 400MHz Server**

Listing H.1: Configuration File for the testbenches.

```plaintext
### DEFAULT CONFIGURATION FILE

##### if you want to configure a configuration file on your needs,
##### copy this file and add the file path to the startup of the benchmark.

#### global system configuration

# where to store the graphics
ch.ethz.inf.vs.gateway.benchmark.Benchmark.graphPath=/tmp/
ch.ethz.inf.vs.gateway.benchmark.Benchmark.numTest=3
ch.ethz.inf.vs.gateway.benchmark.Benchmark.1.class=ch.ethz.inf.vs.gateway.benchmark.Benchmark
ch.ethz.inf.vs.gateway.benchmark.Benchmark.2.class=ch.ethz.inf.vs.gateway.benchmark.Benchmark
ch.ethz.inf.vs.gateway.benchmark.Benchmark.3.class=ch.ethz.inf.vs.gateway.benchmark.cache.BenchmarkCache
ch.ethz.inf.vs.gateway.benchmark.Benchmark.0.class=ch.ethz.inf.vs.gateway.benchmark.cache.BenchmarkCache

## subscriber churn test.
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.numTest=3
# the callback
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.cb=http://192.168.99.5:9999/_eventing/event
# how long we keep the subscription
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.st=1000
# run the whole test for 10000 ms
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.rt=20000
# sample the resulting graph all 10 ms
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.sr=1
# the host to test
# how many simultaneous registrations
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.0.sub=400
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.1.sub=200
ch.ethz.inf.vs.gateway.benchmark.subscriber.SubscriberChurn.2.sub=100

## cache tests
# the host to test
# the name of the benchmark device
ch.ethz.inf.vs.gateway.benchmark.cache.BenchmarkCache.bmd=cm

## many events test
# how many different load profiles
ch.ethz.inf.vs.gateway.benchmark.events.ManyEvents.num=3
# the different load profiles
ch.ethz.inf.vs.gateway.benchmark.events.ManyEvents.0.num=200
ch.ethz.inf.vs.gateway.benchmark.events.ManyEvents.1.num=100
ch.ethz.inf.vs.gateway.benchmark.events.ManyEvents.2.num=50
# the host to test
# the URI of the event sink
ch.ethz.inf.vs.gateway.benchmark.events.ManyEvents.sink=http://192.168.99.5:9999/_eventing/event
# the benchmark runtime
ch.ethz.inf.vs.gateway.benchmark.events.ManyEvents.bmt=20000

## many receivers test
# how many simultaneous registrations
ch.ethz.inf.vs.gateway.benchmark.events.ManyReceiversBenchmark.num=3
# the callback
# the host to test
# how many simultaneous registrations
ch.ethz.inf.vs.gateway.benchmark.events.ManyReceiversBenchmark.0.sub=200
```

**Appendix H**

**Benchmark Configuration Files**

**H.1 400MHz Server**

Listing H.1: Configuration File for the testbenches.
### DEFAULT CONFIGURATION FILE

#### if you want to configure a configuration file on your needs,
#### copy this file and add the file path to the startup of the benchmark.

#### global system configuration

# where to store the graphics
ch.ethz.inf-vs.gateway.benchmarkBenchmark.graphPath=/tmp/

# the benchmark runtime
ch.ethz.inf-vs.gateway.benchmark ManyReceiversBenchmark.num=100
ch.ethz.inf-vs.gateway.benchmark ManyReceiversBenchmark.num=50

## subscriber churn test.
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.numTest=3
# the callback
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.cb=http://192.168.99.5:9999/_eventing/event
# how long we keep the subscription
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.rt=20000
# run the whole test for 10000 ms
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.st=1000
# sample the resulting graph all 10 ms
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.st=1
# the host to test
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.host=http://192.168.99.2:8080
# how many simultaneous registrations
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.0.sub=1300
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.1.sub=650
ch.ethz.inf-vs.gateway.benchmarkSubscriberChurn.2.sub=325

## cache tests
# the host to test
# the name of the benchmark device
ch.ethz.inf-vs.gateway.benchmarkBenchmarkCache.bmd=bc

## many events test
# how many different load profiles
ch.ethz.inf-vs.gateway.benchmark.events.ManyEvents.num=3
# the different load profiles
ch.ethz.inf-vs.gateway.benchmark.events.ManyEvents.0.num=600
ch.ethz.inf-vs.gateway.benchmark.events.ManyEvents.1.num=300
ch.ethz.inf-vs.gateway.benchmark.events.ManyEvents.2.num=150
# the host to test
# the URL of the event sink
ch.ethz.inf-vs.gateway.benchmark.events.ManyEvents.sink=http://192.168.99.5:9999/_eventing/event
# the benchmark runtime
ch.ethz.inf-vs.gateway.benchmark.events.ManyEvents.bmt=20000

## many receivers test
ch.ethz.inf-vs.gateway.benchmark.events.ManyReceiversBenchmark.num=3
# the callback
# the host to test
# how many simultaneous registrations
ch.ethz.inf-vs.gateway.benchmark.events.ManyReceiversBenchmark.0.sub=200
ch.ethz.inf-vs.gateway.benchmark.events.ManyReceiversBenchmark.1.sub=100
ch.ethz.inf-vs.gateway.benchmark.events.ManyReceiversBenchmark.2.sub=50
# the benchmark runtime
ch.ethz.inf-vs.gateway.benchmark.events.ManyReceiversBenchmark.bmt=20000

## many devices test
ch.ethz.inf-vs.gateway.benchmark.md.ManyDevices.num=3
# the callback
# the host to test
# how many simultaneous registrations

---

3.3 1100MHz Server

### Listing H.2: Configuration Files for the testbenches.
H.2. 1100MHz Server

```
ch.ethz.inf.vs.gateway.benchmark.md.ManyDevices.0.sub=200
ch.ethz.inf.vs.gateway.benchmark.md.ManyDevices.1.sub=100
ch.ethz.inf.vs.gateway.benchmark.md.ManyDevices.2.sub=50
# the benchmark runtime
ch.ethz.inf.vs.gateway.benchmark.md.ManyDevices.bmt=20000

## web access test
# test run with many devices
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.num=3
## the host to test
## how many simultaneous clients
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.0.sub=100
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.1.sub=50
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.2.sub=25
## the benchmark runtime
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.bmt=20000
## the number of devices on the test host
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.nd=4000

# test run with little devices but may access
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.num=3
# the host to test
# how many simultaneous clients
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.0.sub=1500
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.1.sub=750
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.2.sub=375
# the benchmark runtime
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.bmt=20000
# the number of devices on the test host
ch.ethz.inf.vs.gateway.benchmark.access.WebAccess.nd=100
```
Bibliography


