Aalto University
School of Science
Degree Programme of Computer Science and Engineering

HUANG, Fuguo

Web Technologies for the Internet of Things

Master’s Thesis
Espoo, July 7, 2013

Supervisors: Professor Jukka K. Nurminen
Instructor: Dr. Mario Di Francesco
Traditional web communication technologies, such as the Hypertext Transfer Protocol (HTTP), provide a uni-directional link and a request/response message exchange model. This solution can be troublesome in web-based applications involving a large number of different interconnected devices, such as in the Internet of Things (IoT). In this thesis, we design communication patterns suitable for scalable and low-latency web-based IoT applications. We further evaluate different technologies to realise such applications. Among them, we focus on the WebSocket protocol, that provides bi-directional communication mechanism through a single socket, thus reducing unnecessary communication overhead and redundant TCP connections. We also consider the WebSocket Application Messaging Protocol (WAMP), a WebSocket subprotocol, to implement some of the considered communication patterns – namely, Publish/Subscribe and Remote Procedure Call – in the IoT. As a case study, we implement web-based IoT applications for the scenarios represented by home automation and social sensing. We further evaluate the performance of WAMP over WebSocket against other protocols commonly used in IoT applications.

**Keywords:** Web, Technology, WebSocket, WebSockets, WAMP, Internet of Things, IoT, sensor, actuator, light, switch, JavaScript, HTML5

**Language:** English
Acknowledgements

Foremost, I would like to express my sincere gratitude to Prof. Jukka K. Nurminen for supervising the thesis work. I especially wish to thank my instructor, Dr. Mario Di Francesco, for his patience and motivation at all stages of the thesis.

I am grateful to Nico Kaiser for his contribution on the development of WAMP library. I am also thankful to my friends for their encouragement, and for helping me stay sane through the whole difficult period.

Most importantly, nothing would have been possible without the love and patience of my family.

Thank you!

Espoo, July 7, 2013

HUANG, Fuguo
Abbreviations and Acronyms

Ajax Asynchronous JavaScript and XML
CoAP Constrained Application Protocol
CORE Constrained RESTful Environments
HTTP Hypertext Transfer Protocol
ID Identification
IP Internet Protocol
IPv4 Internet Protocol Version 4
IPv6 Internet Protocol Version 6
JSON JavaScript Object Notation
MBWS MessageBroker WebSocket Subprotocol
MQTT Message Queuing Telemetry Transport
Pub/Sub Publish/Subscribe
RPC Remote Procedure Calls
REST Representational State Transfer
RTT Roundtrip Transfer Time
SOAP Simple Object Access Protocol
STOMP Simple (or Streaming) Text Orientated Messaging Protocol
TCP Transmission Control Protocol
UDP User Datagram Protocol
URI Uniform Resource Identifiers
WAMP WebSocket Application Messaging Protocol
XML Extensible Markup Language
WSN Wireless Sensor Network
## Contents

Abbreviations and Acronyms

1 Introduction
   1.1 Background and Objectives .............................................. 8
   1.2 Contributions ............................................................... 10
   1.3 Structure ................................................................. 11

2 Web Technologies
   2.1 Internet-based Communication Protocols .......................... 12
   2.1.1 TCP, UDP and IP ....................................................... 12
   2.1.2 HTTP ................................................................... 13
   2.1.3 HTML5 .................................................................. 14
   2.2 Traditional Communication ............................................. 15
   2.3 WebSocket ................................................................. 17
     2.3.1 WAMP ................................................................. 18
     2.3.2 Other Registered Subprotocols ................................. 19

3 Internet of Things
   3.1 Things: Bridging the Real and the Virtual World .............. 21
   3.1.1 NFC and WSANs ......................................................... 22
   3.1.2 Spime and Smart Items .............................................. 22
   3.2 Sensors and Actuators .................................................... 23
   3.3 Web Services for the IoT ................................................. 24
     3.3.1 REST ................................................................. 26
     3.3.2 SOAP ................................................................. 27
     3.3.3 CoAP ................................................................. 28
4 Web-based IoT Applications

4.1 Patterns for Web-based IoT Applications
4.1.1 RPC
4.1.2 Publish/Subscribe
4.1.3 Database-centric Approach

4.2 Smart Lighting and Control Application
4.2.1 Components
4.2.2 Implementation of The Backend
4.2.3 Implementation of The Front-end
4.2.4 Overall Structure
4.2.5 The Role of Messaging Patterns

4.3 Social Sensing
4.3.1 Features
4.3.2 Stack and Architecture
4.3.3 Retrieving Data

5 Evaluation

5.1 Reference Scenario
5.2 Tools and Setup
5.3 Simulation
5.4 Measurements
5.5 Scalability
5.6 Security

6 Conclusion

Appendix A Related Protocols and Formats
A.1 JSON
A.2 URI
A.3 XML
Appendix B Implementation Details

B.1 Web-based IoT Applications Screenshots . . . . . . . . . . . . 62
B.1.1 Smart Lighting and Control Application . . . . . . . . 62
B.1.2 Social Sensing Application . . . . . . . . . . . . . . . . 62
Chapter 1

Introduction

1.1 Background and Objectives

After the web turning from 1.0 to 2.0 where the network as platform, spanning all connected devices. Unlike Web 1.0, applications are treated as continually-updated services that get better the more people use it. This is done by consuming and fusing data from multiple sources, including individual users, while providing their own data and services in a form that allows fusing by third parties [29].

Web browsers are now commonly installed in smart electronic equipments such as computers, smart phones, and tablets which can connect to the Internet. Typical contents, presented by web browsers, are text and multimedia such as images, audio and video. These contents are stored in other computers connected by the Internet. Our work started at this point: a browser directly communicates with an Internet-enabled device. To make the work interesting, devices should have good interoperability i.e., they should be able to communicate with sensors and actuators, or even directly with other devices.

Usability is another important element of Web 2.0. Web pages are not just pages that have simple colours, logos, and animations, but they are more like desktop applications. Thus, web pages are turning to web applications [24].

With the radical improvement of content usability, Web applications con-
CHAPTER 1. INTRODUCTION

tents have been presenting more dynamic. Data can be retrieved from multiple sources in real time and assembled on a single web application or page [24]. A web browser, one of the user agents that retrieves contents identified by Uniform Resource Identifier (URI) [23], is the gateway for users to the Internet.

With the purpose of presenting rich content to users, a web browser communicates with a web server to fetch the data. The interaction between a browser and a web server are, traditionally, based on HTTP requests. That is, a browser sends HTTP requests to a web server, specifying the desired data; the server, replies to the browser in HTTP responses with the data, if available. At the application level on the browser side, an HTTP request is usually sent by using techniques, such as, polling or long-polling over XMLHttpRequest (or Ajax). When using Ajax, data are retrieved from a remote server by sending an HTTP request asynchronously; for more other data, the web browser sends other HTTP requests. In this thesis, we address scalability and performance of web applications involving Internet enabled devices. Specifically, Ajax increases the frequency of requesting the web server, with repeatedly connections and disconnections; or rises the workload of the web server to keep the requests open. The problem becomes severe in the scenario of Internet of Things (IoT) with a large amount of interconnected devices.

However, this approach can radically be improved by removing the redundancy of HTTP requests. Then we transfer all data over a single TCP connection, which is what we called WebSockets. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers without relying on opening multiple HTTP connections [33].

In effort to solve the Ajax abuse, bidirectional and real-time communication problems in IoT, we provide a solution using WebSockets with its subprotocol, the WebSocket Application Messaging Protocol (WAMP). In the thesis, we simulated and visualised the behaviours of IoT devices over an HTML5 application using publish/subscribe (Pub/Sub) and Remote Procedure Call (RPC) patterns, providing by WAMP. These two patterns cover the two most important interaction in IoT: notification and control. Additionally, we present another pattern, the database-centric approach, for
web-based IoT applications.

1.2 Contributions

In the thesis, we defined different patterns for web-based IoT applications suitable for time-sensitive and low overhead communications. Amongst, we focused on Publish/Subscribe (Pub/Sub), Remote Procedure Call (RPC) and database-centric approach, as they cover the most desired features that the web-based IoT applications demand. Furthermore, we implemented the web-based IoT applications in two relevant scenarios, namely, home automation and social sensing. Specifically, we combined the first two patterns (Pub/Sub and RPC) with one HTML5 web technology, the WebSocket, and applied them to a home automation scenario, enabling the web-based IoT application cross-platforms (such as mobiles and desktops). For the database-centric approach, we presented this pattern in a social sensing scenario, ensuring that the whole web-based IoT application is scalable, as the IoT data may increase very quickly.

For the purpose of ensuring real-time communication in the web-based IoT applications, whilst reserving scalability and security, we developed a scalable and robust architecture for each scenario of the aforementioned applications, based on the corresponding requirement. Through the architectures, we managed to achieve scalability and security in our implementation result. Moreover, we presented a way to visualise IoT data. This is an essential aspect to improve usability for the end users to inspect, review and manage IoT data.

We finally compared and evaluated different technologies and protocols for IoT-based communications. Specifically, we evaluated the performance of potential web technologies and protocols. This comparison efficiently highlights the protocols that are not suitable for web-based IoT applications. For the home automation scenario, we further focused on comparison message overhead and Roundtrip Transfer Time (RTT). The obtained results show that our approach has a better performance than other approaches commonly used in the IoT.
1.3 Structure

The rest of the thesis is organised as follows. Chapter 2 introduces the background about web technologies, including the WebSocket and its subprotocols. Chapter 3 discusses the background about IoT and its most relevant aspects; it will also overview relevant Internet-based communication protocols. Chapter 4 presents the design and implementation of the web-based IoT applications, while Chapter 5 provides performance evaluation of protocols suitable to IoT applications. Finally, Chapter 6 concludes the thesis.
Chapter 2

Web Technologies

In this chapter, we will introduce the basic elements for communication in the Internet. We will start with Internet-based communication protocols, then move onto web communication protocols. Furthermore, we will compare traditional communication technology with modern HTML5 communication technologies. Among the HTML5 communication technologies, we focus on the WebSocket protocol and its relevant subprotocols.

2.1 Internet-based Communication Protocols

2.1.1 TCP, UDP and IP

The Transmission Control Protocol (TCP), one of the two major transport-layer protocols, provides a reliable bi-directional communication [7]. TCP addresses the communication problem between applications with the following major features:

- Transferring Data over a connection;
- Connecting two endpoints;
- Data is received the same as it is delivered;
- Data is sent in sequence;
- Reliable startup and shutdown;
The User Datagram Protocol (UDP) is the other major transport-layer protocol used in the Internet [7]. Compared with TCP, UDP is less complex and thus less reliable. For example, UDP is connectionless, meaning a message can be sent between two endpoints without prior arrangement; UDP sends and receives individual messages, meaning it does not divide a message into multiple parts.

The Internet Protocol (IP) provides a communication mechanism for computers or devices [30]. IP distinguishes each computer or device in network by a unique number known as Internet Protocol Address (IP address). TCP or UDP packets are usually encapsulated into IP packets.

### 2.1.2 HTTP

Hypertext Transfer Protocol (HTTP) is an application layer protocol exchanging or transferring contents, such as hypertext, in request/response pattern [14]. HTTP is assigned to port number 80 over either TCP or UDP connection [32]. HTTP communication, however, usually is established over TCP/IP connections with the default port TCP 80. HTTP can be also established other protocol, such as unreliable UDP.

When a browser requests resources from a server, usually the browser sends a HTTP request to the server. Then the server answers with an HTTP response. Before HTTP 1.1, a pair of request/response is, usually, sent in a single TCP connection. Generally, a TCP connection is expensive to create, while most HTTP 1.0 or older connections use TCP at least efficiency, which leads to congestion and unwanted overhead [38]. To improve this, HTTP provides connection reuse mechanism [13] by using Keep-Alive in general headers. This mechanism has been further improved in HTTP 1.1, and thus all connections are set to persistency by default. This mechanism aims for reducing CPU and memory usage, pipelining, reducing network congestion, reducing latency the frequency of TCP opening handshakes and error reported improvement. With pipelining, the performance of HTTP 1.1 can be largely improved compared with HTTP 1.0 [28].

At this point, HTTP persistent connections seem to be perfect and the WebSocket protocol is unnecessary. The HTTP persistent connection, how-
ever, is restricted by its practical consideration. Firstly, Servers are, usually, maintained by setting time-out values. Consequently, the mechanism kills idle connections. E.g. Apache 2.0 server will set the timeout at 15 seconds by default\(^1\), while Apache 2.2 reduces the amount down to 5 seconds\(^2\). Additionally, this implies that both clients and servers need to be capable to recover from any closed events and subsequently indicates delay in communication. Secondly, any client that uses persistent connections is instructed to limit the number of simultaneous connections that they maintain to a given server at the maximum of two, for the purpose of improving HTTP response time and avoiding congestion.

For a browser, in order to synchronise the information with a web server, normally, there are polling or long-polling over Ajax. When we use Ajax, we retrieve data from a remote server by sending an HTTP Request; and if we would like to retrieve more data, we need to send other HTTP Requests. HTTP request/response will, finally, confront with the fact that it increases the frequency of requesting the web server, with repeatedly connections and disconnections; or increases the workload of the web server to keep the requests open. The problem becomes severe when it happens in the scenario of Internet of Things (IoT) with a large amount of devices interconnected. Figure 2.1 illustrates sequence diagram where a client (browser) fetches data from a server using HTTP.

### 2.1.3 HTML5

HTML5 is hypernym that consists of a large number of web technology \[44\]. HTML5 covers the following area: new semantics, new CSS styling, multimedia, Graphics and 3D, Device Access, Performance, offline storage and connectivity. Amount all, the connectivity consists of communication technology such as Web Real-Time Communications (WebRTC), Server-Sent Events (SSE), Cross-Document messaging and WebSocket. WebRTC\(^3\) provides real-time multimedia communication on web without requiring plugins; SSE provides an API for a browser to receive pushing notification from a server via

\(^{1}\)http://httpd.apache.org/docs/2.0/mod/core.html#keepalivetimeout
\(^{2}\)http://httpd.apache.org/docs/2.2/mod/core.html#keepalivetimeout
\(^{3}\)http://www.webrtc.org/reference/architecture
an HTTP connection [20]; While Cross-Document messaging or Web Messaging largely expands the possibility and capability of communication of web browsers with least side-effect on security and privacy. Cross-Document messaging is designed to overcome the same-origin policy.

The WebSocket Protocol is also a part of HTML5 and will be discussed in Subsection 2.3.

2.2 Traditional Communication

The client/server model is one of the network architectures to connect computers or applications, where the server provides resources and the client requests for the resources. Usually, the client/server model is designed as a request/response model, where the server does not actively communicate with clients. The client could be a computer, or an application, such as a web browser. The server is, usually, a (powerful) computer that keeps some applications running as constantly as possible. Those applications could be a database, a web server (application), or a printer server (application) that links to a physical printer.
The HTTP protocol is a request/response protocol [14], where a client, or rather a user agent (UA) communicates with a server through a request and a response message(s). In other words, a client retrieves resources from a server in a request message, while the server hands over the contents in a response message. This is a general model that a client retrieves data from a server. A classic web page must reload, when a web browser updates the content, whose procedure generates a large amount of additional overheads. With Ajax, a technique allowing web pages to be updated asynchronously, web pages can be updated partially as needed, without reloading the whole page. This procedure exchanges less amount of data compared with the classic web pages updating.

Generally, there are three ways to retrieve data asynchronously[25]: polling, long polling and streaming. All these approaches are uni-directional.

With polling, a client sends a request to a server, e.g. a web browser sends a request with Asynchronous JavaScript and XML (Ajax) [16] to a web server and expects data. This is an efficient solution, if the client knows exactly the time when the data is available. However, the server might not have the content when the client requests, or the content may be temporarily unavailable. If this situation would happen in practise, the client would have repeated polling for data. This will result in a large redundant headers generated by HTTP protocol, and a high number of TCP connections. Consequently, the approach leads a heavy workload to the server and will not be suitable in the scenario, such as, some Internet scenarios with a large number of nodes.

Another method, Long polling, is an improved approach, with respect to the frequency of request. Compared with polling, long polling does similarly, expect that the server holds the client’s request for a certain period until timeout. Within this period, the server will send a response(s) back to the client, if there is data available. This method reduces the frequency of HTTP requests, but it increases the workload of a server, such as, ports handling. Generally, an HTTP request is initiated over TCP/IP connections (but any other protocols that guarantee a reliable transport can be used) [14]. In TCP/IP connections, a port number is 16 bits [31], which means the maximum number of ports is $2^{16} = 65536$. However, one characteristic in
the aforementioned Internet scenarios is scalability, which means the number of nodes could yield a larger amount than that of the ports. Even though all ports are dedicated to communication, it is insufficient to support the schema with a large number of Internet-based nodes.

As for the HTTP streaming, or known as HTTP server push, a server sends a response and keeps the connection open and alive, where the connection is triggered by a client’s request. The drawback of this approach is that streaming is still encapsulated in HTTP, and thus buffer mechanism might increase the latency of messages delivering. Moreover, buffer is not an approach for an energy constrained device.

2.3 WebSocket

The WebSocket Protocol enables full-duplex communication through a single socket between a client and a remote server. For this reason, this technology decreases the amount of opening ports on server sides, comparing with the traditional mean of retrieving resource. Moreover, this protocol is not just an enhancement of the current HTTP protocol; it, however, represents enormous advance, especially for real-time, event-driven communication [25].

Moreover, the WebSocket protocol supports data frame including text and binary. Once the WebSocket connection has been successfully established, generally, data frames transfer between a client and a server. However, in consideration of optimising WebSocket connections between a client and a server, the client can request the server to use a specific sub-protocol by including the “Sec-WebSocket-Protocol” filed in its handshakes [33]. A sub-protocol is a application-level protocols layered over the WebSocket protocol [33]. A subprotocol could be implemented by the servers and can be also versioned. The server chooses one or none subprotocol.

The goal of WebSocket technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g. using XMLHttpRequests and long polling)\(^4\). Older web technology, like HTTP,

\(^4\)http://www.websocket.org
remains inefficiencies problems [44]. Firstly, it is not natively designed for nowadays Web Applications with rich contents and interactive demands; Secondly, overheads and unnecessary information cumulate swiftly as the interaction between the client and server continues. WebSocket improves communication problems by solving the above problems, while keeping simplicity. Both WebRTC and WebSocket are in effort to enhance the real-time communication. WebRTC provides APIs that allow web browsers communicate with each other directly, while WebSocket provides the real-time communication mechanism between a client and a server. Figure 2.2 illustrates sequence diagram where a client (browser) fetches data from a server using WebSockets.

2.3.1 WAMP

WebSocket Application Messaging Protocol (WAMP)\(^5\) is a subprotocol officially registered in WebSocket Protocol Registries\(^6\), which gives structure to messaging by providing two high level messaging patterns: Pub-
lish/Subscribe (Pub/Sub) and Remote Procedure Calls (RPC).

WAMP fills in the blanks in the WebSocket protocol: a low-level specification which originally provides raw messaging communication (text or binary). Moreover, WAMP does not make any extra standard, but it bases on established Web standards: WebSocket, JSON and URI. The WebSocket standard is the default transport channel where WAMP is binding. With the WebSocket Standard, WAMP is assumed to be given a reliable, ordered, and full-duplex message channel.

### 2.3.2 Other Registered Subprotocols

Apart from WAMP, there are MessageBroker WebSocket Subprotocol (MBWS), Simple Object Access Protocol (SOAP) and Simple Text Oriented Messaging Protocol (STOMP) can be used as subprotocols, according to the WebSocket Protocol Registries.

MBWS is used by messaging clients to send messages to, and received messages from, an Internet message broker. The Internet message broker is also called a message broker which queues messages, sent by clients, for asynchronous deliveries to clients [19]. Like WAMP, MBWS provides full-duplex and reliable transport; unlike WAMP, MBWS does not obviously provide higher level messaging patterns. MBWS, however, defines a binary message frame starting with an MBWS binary header, and a text message frame starting with an MBWS text header. MBWS provides two additional features. Firstly, it supports connection recovery, which has not defined by the WebSocket protocol. Secondly, it supports metadata in MBWS message headers. MBWS has a lightweight version, called MessageBrokerLight WebSocket Subprotocol (MBLWS) which only supports Message Metadata [19]. However, MBWS is still a draft.

SOAP is a lightweight protocol intending for exchanging structured information in a decentralised and distributed environment. SOAP uses XML technology to exchange data [17]. Microsoft, additionally, defines SOAP Over WebSocket Protocol Binding (MS-SWSB)\(^7\), a subprotocol for the WebSocket Protocol. The subprotocol involves a Web Services Description Language

(WSDL) and supports message exchange patterns (MEPs). With MEPs, SOAP supports one-way, two-way and request/response message exchange patterns, which involves Pub/Sub and RPC.

Simple (or Streaming) Text Orientated Messaging Protocol (STOMP)\(^8\) is a frame-based protocol. It refers to a STOMP frame includes a command, optional headers and a body. STOMP is text-based with UTF-8 encoding by default, but it, optionally, supports binary messages. As for message patterns, STOMP version 1.2 supports Pub/Sub. An example of Pub/Sub model using STOMP with WebSocket is discussed in [44].

Table 2.1 is a comparison summary among WAMP, MBWS, SOAP and STOMP:

<table>
<thead>
<tr>
<th></th>
<th>WAMP</th>
<th>MBWS</th>
<th>SOAP</th>
<th>STOMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Representation</td>
<td>JSON</td>
<td>binary and text with MBWS headers</td>
<td>XML</td>
<td>binary and text with headers</td>
</tr>
<tr>
<td>Message Pattern</td>
<td>Pub/Sub, RPC</td>
<td>not specified</td>
<td>MEP</td>
<td>Pub/Sub</td>
</tr>
<tr>
<td>Connection Recovery</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Metadata</td>
<td>currently no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Released</td>
<td>yes</td>
<td>draft</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2.1: WAMP and other WebSocket subprotocols

\(^8\)http://stomp.github.io
Chapter 3

Internet of Things

In the concept of Internet of Things (IoT), the networked objects or things, called devices, are deployed worldwide and connected over the internet. Moreover, the IoT device is individually addressable, so that each device is interconnected and can be accessed through the standards of the web. In this chapter, we will introduce the basic elements in the IoT. Then, we will move onto web services for the IoT, where we will discuss the most relevant protocols for the web services.

3.1 Things: Bridging the Real and the Virtual World

The term, Internet of Things, was firstly mentioned by Kevin Ashton in once his presentation in 1999 [1].

Semantically, Internet of Things means “a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols” [21].

The “things” refers to the “objects” in the semantical explanation. The “things” are heterogeneous objects. The initial idea of the “things” started with Radio-Frequency IDentification (RFID) tags by Auto-ID Labs[^1], a worldwide network of academic research laboratory in the field of networked RFID

[^1]: http://www.autoidlabs.org
and emerging sensing technologies. Hereafter, other concepts of “things” emerged: NFC tags, Wireless Sensor and Actuator Networks (WSANs), Spime and Smart Items.

### 3.1.1 NFC and WSANs

NFC is a proximity technology that provides contactless communication by using ISO/IEC 14443 and ISO/IEC 18092. NFC tags utilise Reader/Writer mode, one of the modes that a NFC devices should supports [26], to store data and interact with users. A typical scenario is that a NFC tag acts as a smart poster, where a user could place a NFC based mobile phone near a smart poster and thus would retrieve further information. For instance, if a smart poster stores movie poster data, it can forward a user to the movie trailer or its home page. Additionally, there are card emulation mode and Peer-to-Peer mode. Card Emulation Mode e.g. turns a mobile phone into a smart card such as a credit or debit card, and the NFC reader cannot distinguish the difference between the mobile phone and smart cards. Thus, with this features, it can implement a new way of payment system called Mobile Payments. NFC Peer-to-Peer (P2P) mode is designed based on the standards ISO 18092, which allows two NFC devices communicate mutually [26].

WSANs are composed of large numbers of minimal capacity sensing, computing, and communicating devices and various types of actuators. WSANs constitute an important and exciting new technology with great potential for improving many current applications. It also creates new revolutionary systems in areas such as global-scale environmental monitoring, precision agriculture, home and assisted living medical care, smart buildings and cities, industrial automation, and numerous military applications [40].

They are, together with RFID, are recognised as the atomic components that links the real world with the digital world [41].

### 3.1.2 Spime and Smart Items

Spime and Smart Items are also considered to be the “things”. Spime is a neologism created by Bruce Sterling, a science fiction writer, in 2005. Sterling
stated that there will be a new kind of object – user-alterable, baroquely multi-featured and programmable – in the future, for which Sterling invented the term “Spime” [41]. Sterling described that the Spime is one of the objects in Internet of Things since Spime enables a thing to be searchable. i.e., A thing can be searched by a search engine from a virtual world to a physical world [41]. Spime is a bit theoretical, but we can find some of its prototype in the real world, and we call them Smart Items [2].

IoT is also derivative from the vision of “Internet”, which Vasseur and Adam described in [42]. A Smart Object is an item equipped with a form of sensor or actuator, a tiny microprocessor, a communication device and a power source. The sensor or actuator gives the smart object the ability to interact with the physical world. Smart Objects are interconnected using the Internet Protocol (IP).

According to IPSO Alliance, the IP stack is a light protocol that already connects a huge amount of communicating devices and runs on tiny and battery operated embedded devices. This guarantees that IP has all the qualities to make IoT a reality [2].

3.2 Sensors and Actuators

A sensor detects or measures the physical environment (such as temperature, speed, and location etc) and converts it to a form (such as a digital form) which can be read by human or used for other purpose. For example, a light sensor senses the brightness of the environment, then the sensor converts the brightness into a digital form (data). The data can be, further, transferred to other receivers.

Sensors can interconnect wirelessly, and form a network, which is called wireless sensor network (WSN). WSN monitors the environment or other physical data distributively. According to [9], WSN can be apply to a wide range of uses: monitoring space, things, and the encompassing space. Particularly, the first category, space, which is related to the thesis, covers light, temperature, humidity and other environment data.

Sensors perform tasks more than measuring environments. The second category applies on things or objects. Sensors, that being the case, are
applied more in area of such as, the detection of structures breakages [4]. However, things and the environment are sometimes interconnected and interactive. This is the third category when sensors are used e.g. monitor the wildlife habitats [9] and the interaction between traffic queuing and the frequency of traffic lights switching. In short, the WSN provides a good source of data for Internet of Things.

Sensors collect and transfer data for further use, but an actuator provides controlling functions to a system e.g. a light switch can turn on and turn off a light; Here the light switch is an actuator. Sensors and Actuators work mutually can optimise processes. For example, sensors are installed in chemical production to bring much greater granularity to monitoring, where the sensor collects data and transfers them to computers which subsequently analyse the data and signal actuators to adjust the processes, such as ingredient mixtures, temperatures, or pressures [5]. The whole process can then be automated.

### 3.3 Web Services for the IoT

The currently architecture of IoT consists three layers [45]. From top to bottom, they are: application layer, network layer and perception layer. The perception layer consists of the unquietly addressable objects. The network layer connects the objects and connects objects to computers. The application layer makes the IoT data meaningful, useful or extensible.

TCP/IP and UDP ensure the objects communicable in IoT. Moreover, they ensure that the objects can be communicated outside the IoT network. Subsequently, the data collected by sensors are usable. For example, the data collected by sensors can be transferred to a data centre and can be further analysed.

Another issue about IP, Internet Protocol Version 4 (IPv4) is the dominant IP protocol in the Internet currently. IPv4, however, has exhausted [36] and thus could not meet the ‘unquietly addressable’ requirement in IoT. IP Version 6 (IPv6) is its successor. “IPv6 enables end-to-end communication, in which any IPv6 ‘smart things’ can connect to any other IPv6 devices or system from any place and at any time. It enables the possible extension of
the Internet to any device, sensor or actuator, which can become real nodes of the future Internet.” [43]. It primarily provides the following features [10]:

- Expanding addressing capabilities to support a greater amount of addressable nodes.
- Simplifying header format to reduce the cost of packet handling.
- Improving support for extensions and options to allow greater flexibility in the future.
- Flowing labelling capability to label packets and thus to handle, e.g., real-time services.
- Authentication and privacy capabilities.

HTTP and WebSocket ensure that IoT can be used or extended by applications. For example, a web interface that can visualise temperature data. Subsequently, IoT network can be directly or indirectly connect to end users.

A Web Service “is a network accessible interface to application functionality, built using standard Internet technologies” [37]. In other words, an application can be accessed by another application, client or user over the Internet, via a web technology such as HTTP or XML. A Web Service, usually, provides services based on standards at the layer between service requesters and service providers. This means, it does not matter how are the services implemented, e.g. using C/C++, Java, Python or Javascript, or based on whatever platforms, e.g. Windows, Max OS or Unix.

When a request is sent to a web service layer, this layer will start a procedure which further requests the service provider. When the procedure is finished, the web service layer will response to the sender results (the response is sometimes optional and it relies on how is the web service implemented). The communication, between the service requesters and the web service layer, is usually based on standard protocols, which aims at cross-platform interoperability.

In the following, we will talk about a sequence of protocols that can be used in Web Services.
3.3.1 REST

Representational State Transfer (REST) is introduced by Roy T. Fielding in his doctoral dissertation in 2000 [15]. REST is a web architectural style. It is, however, not a standard or protocol.

An architectural style is “a coordinated set of architectural constraints that restricts the roles and features of architectural elements, and the allowed relationships among those elements, within any architecture that conforms to the style. Thus, a style provides a name by which we can refer to a packaged set of architectural design decisions and the set of architectural properties that are induced by applying the style.” [15]

REST was developed in parallel with HTTP protocols with the basic four constraints: resources identified by URI; resources manipulated by standard HTTP methods; resources with various representations; and stateless communication. Resources are data that can be addressed by URIs, which further implies resources are differed by URIs. A representation is a way to present resources. For example, a text, an Image or an HTML document. The representation, moreover, consists of metadata describing the data and, occasionally, metadata describing metadata.

All REST communications are stateless. Thus, each request involves all the necessary information that a receiver needs to understand the request. An example of stateless design is shown in Figure 3.1 [34].

As it is illustrated, the request specifies which page the client fetches, i.e., page 2; while the response message contains what the current page is, what
the next page is and also the requested page.

On the contrary, stateful design, as shown by Figure 3.2 [34], the server
saves each client status and responses the corresponding status to the clients
who request them. In Figure 3.2, the client need not to specify which page it
fetches, but just call the corresponding procedure. The server computes the
result and returns the page to the client.

The replies are in XML format in Figure 3.1 and 3.2, but the REST
reply is not restricted in XML document. It could be JSON or other data
structure. Usually, REST is implemented relying on HTTP and other web
technologies.

3.3.2 SOAP

SOAP, a protocol designed for web services, specifies, firstly, a XML-based
envelope to transfer information; Secondly, a set of rules for translating ap-
plication and platform-specific data types into XML representations [37].

SOAP is largely based on XML standards and provides a standard way to
structure XML messages. Thus, SOAP has two related applications: Remote
Procedure Call (RPC) and Electronic Document Interchange (EDI). RPC is
one way to do distributed computing, where one programme calls a function
or method (called procedure) on another, optionally with parameters and
receiving return value. EDI is used in automated transactions.

A SOAP message contains an optional SOAP header and a required
SOAP body. The SOAP header consists of several information blocks and
each of the block is a header. Each header explains how the message is to be parsed. The message body is the actual message in XML syntax.

SOAP provides a mechanism to handle errors which is called SOAP Faults. A SOAP fault consists of fault code, fault string, the fault actor and the fault details. The fault code has been standardised in namespace belong to http://www.w3.org/2001/06/soap-envelope; The fault string is an explanation of the error for human to read, while the fault detail is a further explanation about the error. The actor is a concept in SOAP Message Exchange Patterns (MEP), a pattern of message exchanged between SOAP nodes, which has been standardised at [3].

When a SOAP message is sent from a provider to a receiver, it passes one or more web services. Each of the web service is called intermediary. All these intermediaries that the SOAP message travels through is called a message path. We call each intermediary on a message path an actor. Thus, the fault actor is the unique identifier of message processing node where the error occurs.

SOAP does not specify the means of SOAP message transports. Generally, SOAP can be transferred through HTTP, FTP, BSD sockets and SMTP etc.

3.3.3 CoAP

Constrained RESTful Environments (CORE) is a working group under Internet Engineering Task Force (IETF) forming in 2010\(^2\). CORE has been contributing the major standardisation work for Constrained Application Protocol (CoAP).

CoAP is a protocol (in draft) used in constrained nodes and constrained networks and designed for machine-to-machine (M2M) applications [35]. A constrained node, usually, has a lower capacity controller with a small amount of ROM and RAM. Constrained networks refer to the networks, generally, with high packets error rates and a low throughput.

CoAP applications or endpoints interact via a request/response model. It also supports built-in services and resources discoveries. With CoAP, appli-

\(^2\)https://datatracker.ietf.org/wg/core/history/
cations or endpoints can easily interact with an HTTP based web architecture [35].

Unlike HTTP, CoAP stack is designed for constrained environment, while CoAP, additionally, differ from HTTP in terms of power consumption and overhead. Generally, CoAP power consumption bytes per-transaction are lower than those of HTTP, which implies longer battery lifetime. Figure 3.3 illustrates a comparison between HTTP and CoAP stack. CoAP consists of the concept of HTTP, but it has been re-designed with the respect to the low processing power and energy consumption constraints [6].

CoAP relies on UDP. Moreover, CoAP cannot communicate with a browser directly without a gateway which translates CoAP to HTTP. The gateway can be an application, e.g. Copper\(^3\), a firefox plug-in, currently, is available for users to browse and interact with Internet of Things devices.

\(^3\)https://addons.mozilla.org/de/firefox/addon/copper-270430/
3.3.4 MQTT

Message Queuing Telemetry Transport (MQTT)\(^4\) is a machine-to-machine (M2M) / IoT connectivity protocol. There are two main specification for MQTT: MQTT v3.1 and MQTT-S (MQTT for Sensor) v1.2. We will compare MQTT V3.1 with WAMP. Since MQTT-S does not work on TCP/IP network, it is not suitable for the IoT thus will not be further discussed.

MQTT is designed as an extremely lightweight publish/subscribe messaging transport; while MQTT-S is a publish/subscribe messaging protocol for wireless sensor networks (WSNs) and is originally designed for non-TCP/IP networks.

MQTT V3.1 and WAMP are both working on TCP/IP network. Different from WAMP, MQTT V3.1 assumes the communications channel is more likely to be unreliable.

MQTT V3.1 has a two bytes message header and a UTF8 encoded payload. Message type, such as Pub/Sub, is defined in the header. Unlike WAMP, MQTT V3.1 does not yet support RPC. Additionally, MQTT V3.1 provides three levels of Quality of Services (QoS) control.

Level 0 is called “At most once delivery”, when the message is delivered highly dependent to the underlying TCP/IP network. There is no any expectation that the recipient will sent back a response and the message arrives at the recipient at once or it just loses.

Level 1 is called “At least once delivery”, when a response is expected to be sent back from the recipient, otherwise, the sender will re-send the message.

Level 2 is called “Exactly once delivery”, when the message will be received and only received once by the recipient. More helper messages will be exchanged at this level to ensure the quality of delivery, and hence there is an increase in the network traffic [22].

WAMP, however, does not support such QoS control. Finally, WAMP has been officially registered in the WebSocket Subprotocol Name Registry, while MQTT is not a member.

\(^4\)http://mqtt.org
3.3.5 Comparison

In terms of the transport layer, CoAP works on UDP, while WAMP and MQTT work on TCP. As for the means of fetching resources, the three protocols differ from each other. WAMP retrieves resources over WebSockets, while CoAP retrieves via HTTP requests and MQTT retrieves over BSD Sockets. WAMP is the only protocol that is compatible with browsers directly, while CoAP and MQTT can not interact with browsers directly. For example, CoAP can be applied on a Firefox browser via a specific plugin. In terms of communication, all the three protocols need servers. For example, WAMP needs a WebSocket server supporting WAMP. In terms of messaging patterns, WAMP is the only protocol that supports RPC, while CoAP is still using the request/response model. However, all the three protocols support Pub/Sub. As for proxy compatibility and requirements, WAMP supports proxy via the WebSocket protocol, while CoAP built-in supports proxy. MQTT needs an external proxy to translate the protocol, which implies delay in communication. Table 3.1 is a summary of the comparison between WAMP, CoAP and MQTT.
<table>
<thead>
<tr>
<th></th>
<th>WAMP</th>
<th>CoAP</th>
<th>MQTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport Layer</td>
<td>TCP</td>
<td>UDP</td>
<td>TCP</td>
</tr>
<tr>
<td>Get Resource</td>
<td>over WebSockets</td>
<td>HTTP Requests</td>
<td>BSD Sockets</td>
</tr>
<tr>
<td>Browser</td>
<td>yes, HTML5 support</td>
<td>no, only via Firefox plugin currently</td>
<td>no, not directly</td>
</tr>
<tr>
<td>Server</td>
<td>WebSocket Server with WAMP support</td>
<td>CoAP Server</td>
<td>MQTT Server</td>
</tr>
<tr>
<td>RPC</td>
<td>yes</td>
<td>request/response model (through REST APIs)</td>
<td>no</td>
</tr>
<tr>
<td>Pub/Sub</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Proxy Support</td>
<td>through WebSockets</td>
<td>built-in</td>
<td>external proxy to translate protocol (implies delay)</td>
</tr>
</tbody>
</table>

Table 3.1: WAMP, CoAP and MQTT comparison
Chapter 4

Design and Implementation of Web-based IoT Applications

In this chapter, we will start with the patterns applied for web-based IoT applications. Then we will discuss the design and implementation of our web-based IoT applications which match the pattern.

4.1 Patterns for Web-based IoT Applications

In this section, we will, firstly, discuss Remote Procedure Call (RPC), a pattern that can be used for sending commands and controlling configuration of IoT devices. Secondly, we will discuss Publish/Subscribe (Pub/Sub) pattern, a pattern that can be used for transferring data in web-based IoT applications. Finally, We will cover database-centric pattern, a pattern that designed for scaling in the scenario where IoT data expand dramatically.

4.1.1 RPC

Remote Procedure Call (RPC) is one of the Inter-Process Communication methods for data exchange among threads or processes on different hosts. Unlike a local application case [39], where the code runs all locally, RPC solves the problem of excusing codes of the whole application which is distributed cross different networks or the Internet.
RPC can be further developed into a massage pattern involving two roles, client and server. Usually, RPC pattern is initiated by a client, which sends an request to a remote server; The server processes based on the request, then responses to the client. An example of RPC pattern is shown in Figure 4.1. The client starts its procedure sequence when the programme is running. The procedure sequence starts with Procedure 1. The client, afterwards, needs the result from Procedure 2 which locates on the Server. Thus, it sends a request to the server. The server handles the logic when the request comes, and thus it starts Procedure 2. At the moment, the Client is waiting for the result until the server responses to the client. With the result, the client can now continue to execute the next procedure, Procedure 3.

RPC is commonly used in real world scenarios. For example, Facebook, a social network web platform, provides open APIs that allows third party applications to access its data. Here, Facebook plays the role of the server(s), while the third party applications play the role of the client. When a third

\[ \text{http://wamp.ws/faq#rpc} \]
party application (client) sends a request to Facebook asking, e.g. a list of the user’s friends, Facebook servers process the request. Facebook servers, then, response to the client, when the result, either success or error, is ready. The client can then use the list of friends to perform further procedures.

From the perspective of IoT, RPC pattern covers one of the two most important interaction in the IoT application architecture, i.e., sending commands or configuration control. In the case of IoT, nodes could locate on different locations from cities to countrysides. Commanding is helpful to configure a node remotely, e.g. setting the parameters (device identification, timeout and reporting rate). As for an actuator, RPC pattern can control its action. e.g. a light actuator, or called a light switch, has the actions of on and off. With commanding, the light can, thus, be switched on or off remotely.

4.1.2 Publish/Subscribe

Publish/Subscribe (Pub/Sub) is one of the message exchange patterns. Messages are exchanged based on publishing and subscribing. A sender of a message is called a publisher. Usually, each message contains a topic. A receiver filters and receives a message by the message topic. The receiver who subscribes topics is called a subscriber.

Pub/Sub architecture consists, usually, at least publishers, subscribers and a broker(s). The broker maps the right messages to the right receivers and exchanges messages among publishers and subscribers. Generally, there is, on one hand, no restricted boundary between a publisher and a subscriber. This means, a publisher can also be a subscriber and receives messages from other publishers; while a subscriber can also be a publisher and publishes messages with topics. On the other hand, there is no order between publishing and subscribing. This means, a publisher can publish a message with a topic without subscribing any topic; or even no one subscribes the topic. The subscriber does not have to subscribe any topic either. This refers to, if the aforementioned situation happens, then messages are, e.g., simply discarded. All the message exchanging logics, however, are handled by the broker(s). The broker(s) should specifies and implements corresponding functions to
An example of the Pub/Sub pattern is shown in Figure 4.2. Client 1 plays roles in both subscriber and publisher in different stages. i.e., When it subscribes topic 1, client 1 is treated as a subscriber. Client 1, however, changes to a publisher when it publishes a message with the topic (topic 1). The broker analyses the message and maps the topic to the subscribers who are interested in the topics. In this case, both Client 1 and Client 2 are interested in the topic. Hence, the broker replies the message to both of them. Client 1, and thus, changes its role to a subscriber.

The concept of the Pub/Sub pattern has been used in real world cases. Twitter, for example, can be treated as a real world scenario, even though the implementation might be different in Twitter. When a new user, User 1, posts a message in Twitter, usually, it will not popup on any other users’ home page, but it will popup on User 1’s homepage or a similar page that represents the user. This can be understood that User 1 subscribes himself, but no one subscribes him.

User 1 can follow other users based on his interest. For example, User 1 follows User 2. User 1 will, then, receives User 2’s post on User 1’s homepage. Here, User 2 becomes a topic that User 1 subscribes; User 1 is the subscriber and User 2 is the publisher. Twitter, acts as a broker, handles the logic how
the messages sent by User 2 delivers to User 1.

From the perspective of IoT, Pub/Sub pattern covers the other one of the two most important interaction in the IoT application architecture, i.e., notification. With notification, data can be sent interconnected, such as among nodes or from nodes to data centres. For example, one node, Node A, collects temperature data and humidity data. Node A then submits the collected data to the other node or data centre (called Node B) for the purpose of assembling. One strategy, which organises and manages the data sent by Node A, is to set the topic of the each message to the data type (i.e., temperature or humidity). Subsequently, Node B can easily categorise the data for further processing.

### 4.1.3 Database-centric Approach

Database-centric is a software architecture where database plays critical role. Generally, database provides fault tolerant and reliable transactions. Database-centric Approach, in IoT services, subsequently, provides a dependable mechanism to record a large amount of IoT data.

An example of implementing a database-centric IoT service has been discussed in [11]. The approach merges the features of database-centric approach with the demands of IoT services, and provides customised features for IoT services. Thus, the approach, firstly, supports heterogeneous and multimedia contents; Secondly, it provides standard APIs for standard web services, and thus the IoT data can be utilised at optimally maximum; Finally, live notification is supported and security features are applied.

In this pattern, data are transferred through REST APIs with HTTP Requests. This means, there is an intermediate layer between a client and the database, which translates the raw data from database to a format that the client requests. As for real-time notifications, most database has already natively provided notifications for added or updated data. However, the real-time notification passage between the database and the client needs to be defined in the intermediate layer. For example, Ajax techniques (such as polling and long polling) can be applied. That is, the intermediate layer handles the opening connections over HTTP between itself and the client,
for purpose of informing the client as needed. Some database, like CouchDB, has built-in intermediate layer. Thus, the CouchDB handles the connections directly. Figure 4.3 shows the logic of intermediate layer.

4.2 Smart Lighting and Control Application

In this section, we will discuss on our home automation implementation, the Smart Lighting and Control Application. We will start with its components. Then we will explain the components interconnection in backend and front-end. We will also cover the role of the aforementioned patterns in this application.

4.2.1 Components

In the thesis, we built an HTML5 based application to demonstrate a smart lighting and controlling IoT service. The concept of the sample service consists the following elements: a light, a switch, a sensor and the sun. The light is a component that visualises the light status. The switch is a component that represents an actuator that can control the light. The sensor is a component that senses the environment brightness. The sun represents the environment. The change of the luminance from the sun affects on the sensor which then affects on the light. The luminance of the light will be, subsequently, adjusted. The service allows the light to keep the brightness of a space constantly, or to be adjusted by the switch: on, off or to dim. The relationship among the components is shown in Figure 4.4.
4.2.2 Implementation of The Backend

With the concept above, we implemented an embedded web server based on a NodeJS. We also attached a WebSocket server to the NodeJS, and a WAMP server on top of the WebSocket server. The embedded web server, then, understands the WebSocket protocol and the WAMP subprotocol. The component layer structure is shown in Figure 4.5.

We added a generic component layer on the embedded web server. The generic component is an abstract layer that describes information and generic configuration of a device, e.g., component type (temperature sensor, humidity sensor or light actuator etc), device ID, returnable, timeout, data sent rate and some other hardware parameters.

The generic component also implements a function that controls the component events. For example, if the sensor should send the data in some fixed interval or the data should be sent only by signalled. In the thesis, the generic component covers sensor and actuator. Thus, we added another component layer onto the generic component. All the layers interconnected will, subsequently, play the role of sensor or actuator.

We start with the sensor. As a sensor is built on top of a generic component which handles parameters configuration and event abstraction, a sensor is capable of responding to configuration and events. As for a light sensor, the events are the followings:

- **Get current sensor data.** This function triggers a sensor to retrieve data from its environment. In the sample, the sensor senses light brightness data.
• Configuration. To configure a sensor with parameters. Sensor’s action, however, is usually hardcoded and cannot be configured remotely. This is due to a security reason, that the currently implementation does not allow an action parameter (such as a function) transmitted remotely.

• Reset. To reset the configuration of a sensor to default.

As for actuator, it is built at the same layer as that for sensor, and thus, it also inheres features from generic sensors as the sensor does. In the sample, we configure an actuator into a light switch, and it can be described as the followings:

• Switch on. To switch on a light and regardless the light current state.

• Switch off. To switch off a light and regardless the light current state.

• Switch a light. To switch the light to its opposite state. This means, if a light is on, it, then, will be switched off and vice versa.
• Adjust luminance. To adjust a light luminance to a given value which should be in the range of a light acceptable value.

• Configuration. Similar to that of a sensor.

• Reset. Similar to that of a sensor.

In effect with simulation, we added an virtual component that simulates a light. Thus, a light status (on or off) can be check through this component. Summary so far, the interface of a light sensor/switch (sensor/actuator) can be illustrated in Figure 4.6. in a generic way.

To view the architecture, at a higher level, which we have exposed so far, the light sensor and the light switch are connected to a component with an embedded web server, or separately. This architecture is shown in Figure 4.7.

The light sensor and the light switch are linked to a component respectively. The two components differ themselves by the topmost layer which we
showed in Figure 4.5. With the embedded web server, the light sensor and the light switch can communicate with each other and other components (such as the simulated component, light) over WebSockets. We can think this way, the component, which the light switch connected to, enhance the functionality that the light switch can do. For example, it controls the light via the interface switch on/off.

### 4.2.3 Implementation of The Front-end

However, the current system should be accessible from outside, such as by end users. We implemented this via a web front-end interface.

The structure is layered at front. Concerning the connection is over WebSockets, a WebSocket layer is arranged at the bottom. The WebSocket layer handles the connection with the embedded web server.

The communication coming from the embedded web server, is however, using WAMP protocol. Thus, we added another layer, WAMP layer, on top of the WebSocket layer. With these two layers, the front-end and the backend can, now, communicate.

We also would like to control the backend, and thus we added another layer, APIs layer, on top of the WAMP layer. The APIs layer contains light sensor APIs and light switch APIs. As the name implies, programmers can
control the backend through the APIs that this layer provides. For example, with the APIs, we can make a web interface that visualises the light switch which can response to users click event and subsequently control the light on and off; we can also visualise the light status on a web interface, so we can check the light status remotely.

This architecture is shown in Figure 4.8.

### 4.2.4 Overall Structure

We have covered end users, front-end structure and backend structure. Now, we can link them together. This architecture is shown in Figure 4.9.

The APIs layer connects users and the interface provided by the backend. WAMP Client talks with WAMP server through WebSocket Client which talks with WebSocket Server.

### 4.2.5 The Role of Messaging Patterns

Previously, we have discussed Pub/Sub and RFC. These two patterns, handled by WAMP layer, cover the two most important interactions in Internet of Things: notification and command. Here, we discuss how are they integrated in the sample. This architecture with message patterns is shown in Figure 4.10.
As the front-end is built with HTML5 technology, the sample is capable with any browser that supports HTML5 feature (WebSocket). Thus, the architecture is designed in a way that end users can control the light switch and view the light status via a browser.

When a end user control the light switch, such as turning on the light; turning off the light; or adjusting the luminance of the light, it sends the command using RPC patterns. RPC pattern is also applied in component initialisation and reset. When the user view the visualisation of the light status, it communicates with the components via Pub/Sub patterns.

The interaction between the light switch and the light is using Pub/Sub pattern. The light component subscribes a topic from the light switch. When the user e.g. turns on the light, the switch component publishes a message with the light on value (such as, true) to the light component. The light component, then, receives the message, processes the message and turn on the light. Luminance adjustment is similar.

The interaction between the light sensor and the light is, also, Pub/Sub
pattern. The light component subscribes a topic from the light sensor, as opposed to that from the light switch. When the light sensor updates its collected data, the light receives the data and subsequently adjust the light luminance according to its setting.

4.3 Social Sensing

In contrast to the WebSocket based sample: Smart Lighting and Control Application, we provided another sample IoT service, Social Network IoT Sensing Application, based on data centralised model using HTTP for communication.

4.3.1 Features

The sample application provides a way to visualise data collected by IoT devices remotely in nearly real-time. For example, a user deploys IoT devices...
CHAPTER 4. WEB-BASED IOT APPLICATIONS

at home. The devices, then, collect temperature, humidity and multimedia (image, video, and audio) data at home. The purpose of the design enables the user to inspect those data in a visualised way from a browser. Visualisation means, for example, a graph chart of the temperature of the user’s home within the past 24 hours. Real time means, for example, the web interface updates corresponding to live notification, once the collected data have been processed.

Furthermore, the sample application is extensible by integrating third party APIs. For example, the sample could visualised IoT devices geolocations and geographic structures, with Google Map APIs; Notification could be extended into the area of social network, other than the owner exclusively.

The underlying infrastructure of application handles scaling such as distributing and clustering. This feature satisfies the same demand in IoT services.

4.3.2 Stack and Architecture

Social Network IoT Sensing Application is a CouchApp\(^2\), a Javascript and HTML5 application served directly from CouchDB. CouchDB is a documented-oriented database. Data are stored in JSON documents and are accessed with a web browser via HTTP. CouchDB is, also, designed for data distribution and replication. In short, CouchDB provides scalability and flexibility. These features are meaningful to the challenges \([11]\) – heterogeneity and scale – in IoT services. Its layer stack is shown in Figure 4.11.

CouchDB acts as an HTTP server, serving HTML directly to a browser, while manages heterogeneous data and horizontal scaling; CouchDB, moreover, provides live notifications and handles over the notifications to upper layers. External data, such as collected by IoT devices, connect to CouchDB via a gateway. The collection of the data and the purpose of the gateway have been discussed in \([11]\).

CouchApp inherited these features from CouchDB. Furthermore, CouchApp is extensible. Thus, we have a third layer to extend the functionality and

\(^2\)http://couchapp.org/page/index
customise the usability. For instance, we used BackboneJS, a Model-View (MV) Javascript library, to structure our application. Third party APIs, additionally, can be integrated into the third layer and thus enable, e.g. social network or geography functions. End users interact with the system via the third layer.

The architecture of the sample application is shown in Figure 4.12.

4.3.3 Retrieving Data
The architecture, which end users directly interacting with, is based on a HTTP web server, CouchDB. Thus, all connections are HTTP based. In order to keep notification alive, the browser has to maintain HTTP connections, negotiate with the web server, update HTTP connections and keep HTTP connection alive.
Chapter 5

Evaluation

5.1 Reference Scenario

We planned to build an IoT service with web technology that differed from the previous solution, database-centric approach, which largely relies on Ajax for data exchange.

The prevision of the thesis was constructed in two parts: web technology and IoT; Its solution should integrate the two parts into a coherent service. Following the consideration of HTML5 connectivity features, we focused on the WebSocket protocol which fills the part of web technology. As for IoT, which covers a wide area, we selected one scenario – smart lighting control – with a sensor and an actuator from many IoT visions such as NFC applications, smart housing and phones connectivity.

Compared with other HTML5 communication technologies, the WebSocket protocol is more mature, having ready made auxiliary libraries with a feasible amount of selections. The WebSocket protocol, additionally, supports subprotocols where WAMP fills the blank and provides the messaging patterns which furthermore covers the two most important interaction in IoT service.

The Sensor and actuator are two of the most common “things” in IoT architecture. The sensor collects data, while the actuator emits commands. We evaluated these two components matches the web technology, WebSockets with WAMP. We, further, collected several IoT scenario assumptions - high
lever descriptions of what we thought might work. For example, we drafted using sensors in a phone, collected data and reported to data centre; Applying a searching function in the real world; or smart building. We evaluated the phone case is a bit far away from web technology and the implementation might involve more mobile technology rather than web technology. As for the searching function, it might involve more hardware configurations and physical experiments.

Among of all, thus, we trimmed down the assumptions and focused on the smart building scenario. We proposed several hypotheses – more granular descriptions of the assumption – which target on the following specific areas: sensor deployment, data collection, data transmission, endpoint visualisation and others, such as business model level cases. Moreover, we continuously decreased the amount of areas to work on. Data transmission is related to the thesis scope where it matches Pub/Sub and RPC patterns; While endpoints visualisation provides a human friendly interface to review data, and it, besides, performs simulation, such as, simulating the environment.

Finally, we started to experiment on the smart lighting and control application, an IoT service about a sensor and an actuator with the WebSocket.

5.2 Tools and Setup

Before we started the thesis project, we assumed the front-end environment would largely rely on web browsers. As a result, we expected to build it with JavaScript. For this reason, we found AutobahnJS\(^1\), an open source JavaScript networking library.

AutobahnJS supports WAMP at a higher level, while supports the WebSocket protocol at lower level. Thus, it is feasible to create a real-time enabled HTML5 applications with the library. Moreover, AutobahnJS has - another advantage of AutobahnJS - clear tutorials and documents.

As for the backend deployment, it relatively has more options where a list of existing library has been being appended to support WAMP. From the perspective of programming languages, there are implementation on

\(^1\)http://autobahn.ws/js
Python, PHP, JavaScript, Java and others. Our implementation is based on WAMP.IO, a NodeJS library supporting WAMP. WAMP.IO does not handle the WebSocket protocol, but its dependency does.

At the beginning, there were some technical problems in WAMP.IO, leading an initial connection failure with a WAMP client. Thus, we moved to Ratchet, a backend implementation using PHP, until the developer fixed those problems. Compared WAMP.IO with Ratchet, we found out Ratchet is more verbose, from the perspective of development: there are some features we did not need. WAMP.IO, however, targets the desired features - Pub/Sub and RPC - straightforward. With its simple architecture, we could easily customise the library to match demands more closely. Moreover, its developer, behind the library, actively identified and fixed bugs, making the whole development reliable.

5.3 Simulation

We mentioned about the light, the switch/dimmer, the sensor and the sun in the thesis. Considering the thesis scope, we decided to focus on the software level, rather than the hardware part. Thus, we, at the current stage, simulated the light, the switch, the sensor and the sun.

For the simulation, we ported the most relevant features – directly related to the implementation – to the software level. Meanwhile, we analysed its possibility bringing those features and the results back to physical objects.

For example, a light bulb emits light, where the light is the key link connecting the virtuality to the reality. We mapped light luminance into digital numbers at the software level. The luminance can thus be controlled by adjusting digital numbers. The light status – on and off – similarly, can be simulated by e.g. boolean value.

As for the switch/dimmer, a switch turns on or off the light, while a dimmer adjusts the luminance of the light at a certain range. The action of turning/adjusting is the key part that we simulated. The simulation of turning/adjusting, is, being the case, to connect the light component and the switch/dimmer component at the software level, and making the light response to the action of the switch/dimmer. This is, actually, how a
A sensor, generally, converts physical quantity into digital signal which can be further read or analysed by an observer or an instrument. Thus, a sensor is, essentially, a programme that provides data at the software level. Consequently, we simulated the sensor through a programme piping the output of the sun to the light.

As for the sun, we also ported it to the software level. To do so, we made a programme that generated a sequence of numbers periodically increasing to a peak then recurring.

5.4 Measurements

We measured about overheads and Roundtrip Delayed Time (RTT) comparing WAMP with MQTT. Figure 5.1 illustrates the message overheads on WAMP and MQTT.

WAMP shows its lightweight design better than that of MQTT. For each 20 bytes of data sent, WAMP will have 1.25 times overheads larger than the message itself, while MQTT will have around 2.4 times overheads generated.
Figure 5.2: Message RTT

Figure 5.2 illustrates the message RTT on WAMP and MQTT.

We measured the RTT from a local workstation to an Amazon micro instance in Ireland. And the result we got is, generally, MQTT has a longer RTT than that of WAMP; WAMP RTT starts to increase after sending 10k bytes, while MQTT stays constantly. The sudden increment could be caused by the programme design, the workstation or the Amazon instance we used. As the package size increased, the processing and queueing time might consequently enlarge.

5.5 Scalability

In Pub/Sub and RPC Patterns for web-based IoT applications, the broker(s) plays a key role managing the interconnection of clients. Each client differs from others by its ID. Thus, the nature of scaling in these patterns is the implementation of addressing clients. Further communications and interactions, among users and backends, rely on the successful navigation of the broker(s).

In the scenario of the smart lighting and controlling application, the scal-
ing issue generated by the enlarging amount of lights, switches and light sensors. Considering the limited resources that an embedded web server are assigned, the architecture might not suit for a large scale without other resources support such as extra brokers with power sources.

As for the database-centric pattern, the CouchDB solution naturally provides a reliable and flexible way to scale horizontally and to replicate, with the growing amount of data. The database acts as a hub between IoT devices and end users, while it can be further developed into cloud-based architecture which maximises its scalability.

5.6 Security

The WebSocket protocol supports connections over Transport Layer Security (TLS). TLS provides communications security over the Internet [12]. Furthermore, the WebSocket protocol provides applicable security considerations as the followings [33]:

- **Non-browser** Client resistant. The WebSocket protocol prevents malicious running inside a web browser.
- **Origin Considerations.** The WebSocket protocol provides a way for servers that do not intend to process input from web pages from other origins.
- **Masking.** All data from clients to servers are masked, so that the attacker does not have knowledge about how the data being sent.

The security considerations on database-centric pattern implemented by CouchDB has been discussed in [11]. As for database, one of the most important security issue is authentication and access control. It is notable that CouchDB, by default, does not turn on access control for different users. This means, everybody has the same permission as the administrator. CouchDB authenticates users in many ways, such as HTTP basic authentication, cookie authentication and OAuth [18]. CouchDB has started to support Secure Sockets Layer (SSL) since CouchDB version 1.1, and thus provides communication security over the Internet.
Chapter 6

Conclusion

In this thesis, we applied web technologies to Internet of Things scenarios. We focused on the WebSocket protocol as a key web technology. The WebSocket protocol creates a bi-directional connection between a server and a web browser. We highlighted its advantage by comparing it with the traditional method, HTTP, according to the behaviours of fetching resources from a web server. Furthermore, we chose WAMP as a subprotocol to complement the WebSocket to enable efficient and scalable interactions with Internet of Things services.

Moving onto the part of Internet of Things, we integrated a sensor (light sensor) and an actuator (light switch/dimmer) into a web application. The major feature of the sensor (sending data) and that of the actuator (commanding and configuration) perfectly match the massage patterns, Pub/Sub and RPC, which WAMP provides. Consequently, the smart lighting and controlling Internet of Things service emerged.

Furthermore, we compared the smart lighting and controlling application with social network IoT sensing application, an application interacting with IoT services with traditional method (HTTP), from the perspective of architectures, implementation, scalability and security.

Moreover, we compared WAMP with other registered WebSocket subprotocols (MBWS, SOAP and STOMP) in terms of the related features; and with other potential protocols (CoAP and MQTT), in terms of the related practical deployments. We further evaluated WAMP and MQTT with com-
communication overheads and RTT.

There are, surely, many potential room for the future research on the blending of web technology with Internet of Things. The upsurge in bleeding-edge web technology might be supportive to the future Internet of Things services research, e.g., for the efficiency, maintainability and scalability.

At the current stage, the implementation is done at software level: hardware part are simulated. Therefore, the development process can be trimmed to fit in the scope of the thesis. However, we, carefully, preserved the essential features of each hardware component and simulated at the software level. This means, there should not be many different behaviours for further deployment on hardware components. One example, which could link all the features back to hardware, is Raspberry Pi\(^1\). Raspberry Pi supports Python as its official programming language by default, but any language which will compile for ARMv6 can be used. This implies that NodeJS can be ported on Raspberry Pi. Moreover, the embedded web sever and its upper layers, which we implemented, are not restricted by NodeJS, meaning the architecture can be built with other programming languages, such as Python. A light, a switch and a sensor can, subsequently, be connected to Raspberry Pi. We can, furthermore, use a phone’s sensors as a data source, and transmit the data from the phone to Raspberry Pi.

\(^1\)http://www.raspberrypi.org/
Bibliography


[10] DEERING, S. E. Internet protocol, version 6 (ipv6) specification.


[35] Shelby, Z., Hartke, K., and Bormann, C. Constrained application protocol (coap).


[41] Sterling, B. Shaping things.


Appendix A

Related Protocols and Formats

In this appendix, we will introduce the protocols and formats that we mentioned in this thesis.

A.1 JSON

JavaScript Object Notation (JSON) is a text format for the serialisation of structured data. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard [8]. JSON has numbers, strings, booleans and null as its primitive types. Additionally, it has objects and arrays as structured types. JSON becomes popular to exchange data between in web services or applications. With JSON, WAMP message payload is serialisable and provides data types according to those of JSON.

A.2 URI

Uniform Resource Identifiers (URIs) provide a simple and extensible means for identifying a resource. URIs have a global scope and are interpreted consistently regardless of context [27]. WAMP binds to URIs by default, then it assumes global assignments and resolution to be unique. URIs are used as IDs for both topics in Pub/Sub and procedures in RPC.
A.3 XML

Extensible Markup Language (XML) is a markup language that was designed to transport and store data. Furthermore, XML was designed to be self-describing and can be used straightforward over the Internet. XML has been widely used, for example, the SOAP protocol that we mentioned in the thesis.
Appendix B

Implementation Details

In this appendix, we will show the screenshots of the aforementioned web-based applications. We will also briefly introduce the libraries we used in the implementation.

B.1 Web-based IoT Applications Screenshots

B.1.1 Smart Lighting and Control Application

In this implementation, we used WAMP.IO\(^1\), a WAMP server library, to attach to a WebSocket server running on NodeJS. As for the front-end, we used AutobahnJS\(^2\), a WAMP client library, to connect to the WAMP server.

Figure B.1 shows the visualisation of the lights changing according to the sun brightness, for the purpose of keeping the light brightness in a constant level.

Figure B.2 shows the visualisation of remote controlling lights status.

B.1.2 Social Sensing Application

In this implementation, we built the customised web interface by using BackboneJS\(^3\), which simplifies the process of handling data and connecting to the

---

\(^1\)https://github.com/nicokaiser/wamp.io
\(^2\)http://autobahn.ws/js
\(^3\)http://backbonejs.org/
Figure B.1: Visualisation of Lights Changing According to the Sun
database. We also used D3\(^4\) library to visualise the data.

Figure B.3 shows the social sensing application based on geo-location.
The density of the nodes is visualised in heat map. This is, the colour scale
is the reflection of the number of nodes at a place.

Figure B.4 shows the social sensing application based on nodes interconnection.

Figure B.5 shows the visualisation of data in database-centric approach.

\(^4\)http://d3js.org/
Figure B.2: Visualisation of Remote Controlling Lights
Figure B.3: Visualisation of Social Sensing based on Geo-location
Figure B.4: Visualisation of Social Sensing based on Nodes Interconnection

Figure B.5: Visualisation of Data in Database-centric Approach