AALTO UNIVERSITY
School of Science
Department of Media Technology

Kalle Juhani Säilä

UbiQloud: A Platform-as-a-Service for the Web of Things

Master's Thesis
Helsinki, June 25, 2012

Supervisor:	Professor Petri Vuorimaa, D.Sc. (Tech.), Aalto University
Instructor:	Markku Laine, M.Sc. (Tech.), Aalto University

AALTO UNIVERSITY
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Kalle Juhani Säilä		
Title:			
UbiQloud: A Platform-as-a-Service for the Web of Things			
Date:	June 25, 2012	**Pages:**	xiii + 76
Professorship:	Media Technology	**Code:**	T-111
Supervisor:	Professor Petri Vuorimaa, D.Sc. (Tech.)		
Instructor:	Markku Laine, M.Sc. (Tech.)		

Over the years, the World Wide Web (Web) has evolved from a simple system for sharing static documents to a social and dynamic application platform. In addition, the range of devices and input methods used to interact with Web applications has increased. This evolution has opened up new possibilities for application development as data from other applications and physical devices is now available for mashups through open APIs. At the same time, however, the complexity of application development has increased and the technologies at the foundation of the Web fail to meet the requirements for modern Web applications.

The main objective of this Thesis was to study how the development of modern Web applications can be facilitated leveraging third-party services and modern technologies. Furthermore, the study focused on designing and implementing a modern cloud-based platform, *UbiQloud*, offering a wide range of essential services needed to develop social and location-aware Web of Things applications with real-time communication capabilities. The platform was validated by developing two sample applications on top of it as well as by conducting a series of performance and stress tests.

The results show that it is possible to implement a scalable, high performance cloud-based platform that offers a wide range of services essential for modern Web application development and can be used with real applications in real world settings.

Keywords:	UbiQloud, server push, Web of Things, positioning, social media, cloud service, XMPP, WWW, Web
Language:	English

AALTO YLIOPISTO
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Kalle Juhani Säilä

Työn nimi:
UbiQloud: Palvelualusta Esineiden Web -sovelluksille

Päiväys:	25. kesäkuuta 2012	**Sivumäärä:**	xiii + 76
Professuuri:	Mediatekniikka	**Koodi:**	T-111

Työn valvoja: Professori Petri Vuorimaa, TkT
Työn ohjaaja: DI Markku Laine

Vuosien saatossa World Wide Web (Web) on kehittynyt pöytäkoneilla käytettävästä yksinkertaisesta dokumenttien jakojärjestelmästä sosiaaliseksi ja dynaamiseksi ohjelmistoalustaksi, jota käytetään monenlaisilla päätelaitteilla eri yhteyksissä. Webin kehitys on luonut sovelluskehittäjille mahdollisuuden uudenlaisten sovellusten kehittämiselle, joissa hyötykäytetään tietoa muista sovelluksista ja fyysisistä laitteista avointen rajapintojen kautta. Samanaikaisesti sovelluskehitys on muodostunut entistä haastavammaksi, koska Webin perustana toimivat teknologiat eivät enää pysty vastaamaan nykyaikaisten sovellusten vaatimuksiin.

Tämän diplomityön tarkoituksena oli selvittää, kuinka nykyaikaisten Websovellusten kehittämistä voitaisiin helpottaa kolmannen osapuolen palveluiden ja uusien teknologioiden avulla. Lisäksi diplomityöhön sisältyi nykyaikaisen pilvipohjaisen alustan, UbiQloudin, suunnittelu ja toteutus. Alustan tarkoituksena on tarjota sovelluskehittäjille yhdestä paikasta suuri määrä palveluita, joita tarvitaan reaaliaikaisten, sosiaalisten ja paikkatietoisten Web of Things -sovellusten kehittämiseksi. Alusta validoitiin kahden esimerkkisovelluksen avulla sekä suorittamalla joukko testejä alustan suorituskyvyn mittaamiseksi.

Tuloksien pohjalta voidaan sanoa, että on mahdollista toteuttaa skaalautuva ja suorituskykyinen pilvipohjainen alusta, joka tarjoaa suuren määrän nykyaikaisille Web-sovelluksille tärkeitä palveluita. Lisäksi esimerkkisovellukset todistavat, että alusta soveltuu käytettäksi oikeiden sovellusten kanssa oikeassa ympäristössä.

Avainsanat: UbiQloud, push-viestit, esineiden Web, paikantaminen, sosiaalinen media, pilvipalvelu, XMPP, WWW, Web
Kieli: englanti

Acknowledgments

I would like to thank the following persons:

M.Sc. Markku Laine at the Aalto University for his invaluable guidance and support as the instructor of this Thesis. In addition to the formal instructor duties, I would like to thank Markku for the countless laughs and inspiring work sessions that made the work feel like something else entirely.

Professor Petri Vuorimaa at the Aalto University for giving me the opportunity to work with inspiring topics close to my heart as well as patiently guiding and supervising my Thesis.

MA in New Media Petri Saarikko, my closest co-worker and a friend, at the Aalto University for making every work day interesting and different. I would also like to thank Petri for all the insight and knowledge of graphics and service design as well as for both the serious and hilarious conversations on and off work.

Friends and family for their invaluable help and support through out the years. Especially I would like to thank my mum, **Heidi Nyman**, and aunt, **Christel Nyman**, for encouraging me to pursuit my dreams. In addition, I would like to thank my late father, **Pertti Säilä**, who would have been so proud of me.

Inka-Maria Karhunsuo, my beloved wife and best friend, who makes me whole and has pushed me forward during the years. I love you!

Nooa Säilä, my little baby boy, whose smile continuously makes my world a better place. I grow as a man as I watch him grow to a little human being.

Helsinki, June 25th, 2012

Kalle Säilä
kallesaila@me.com

Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BOSH	Bidirectional-streams Over Synchronous HTTP
CoAP	Constrained Application Protocol
CSS	Cascading Style Sheets
CSV	Comma Separated Values
EAN	International Article Number, formerly European Article Number
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUPSS	Gateway-Based Ubiquitous Platform for Smart Space
HMAC	Hash-based Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IM	Instant Messaging
IoT	Internet of Things
IP	Internet Protocol
ISBN	International Standard Book Number
JID	Jabber ID
JSON	JavaScript Object Notation
LAN	Local Area Network
MB2	Magic Broker 2
MIDE	Multidisciplinary Institute of Digitalisation and Energy
NFC	Near Field Communication
OS	Operating System
PaaS	Platform-as-a-Service

RDBMS	Relational Database Management System
REST	Representational State Transfer
RFID	Radio Frequency Identification
RPC	Remote Procedure Call
RSS	Really Simple Syndication
SaaS	Software-as-a-Service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UHF	Ultra-High Frequency
UI	User Interface
UPC	Universal Product Code
URI	Unified Resource Identifier
UTC	Coordinated Universal Time
VoIP	Voice over IP
W3C	World Wide Web Consortium
Web	World Wide Web
WLAN	Wireless Local Area Network
WoT	Web of Things
WSN	Wireless Sensor Network
WWW	World Wide Web
XEP	XMPP Extension Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol, formerly Jabber

Contents

Abbreviations and Acronyms **v**

1 Introduction **1**

1.1 Organization of the Thesis 3

2 Background **4**

2.1 Server Push 4

2.1.1 Publish/Subscribe 5

2.1.2 XMPP 6

2.1.3 WebSocket 8

2.2 Internet of Things / Web of Things 9

2.2.1 Smart Object 10

2.2.2 Smart Space 10

2.2.3 Representational State Transfer 11

2.3 Positioning 12

2.4 Social Media 14

2.4.1 Social Login 15

2.5 Cloud Computing 16

3 State-of-the-Art **19**

3.1 Related Research Activities 19

3.2 Existing Services 20

3.2.1 Server Push Services 20

3.2.2 Web of Things Services 22
3.2.3 Positioning Services 23
3.2.4 Social Media Gateways 24

4 Research Aims **26**
4.1 Research Objectives and Scope 26
4.1.1 4D-Space Project 27
4.2 Research Questions 27
4.3 Research Methods 28

5 Platform Requirements **29**
5.1 Terminals and Devices 29
5.2 Users 30
5.3 Objects 32
5.4 Verification of the Requirements 32
5.5 Requirements 32
5.5.1 R1: RESTful API 34
5.5.2 R2: Server Push 34
5.5.3 R3: User Profiles 34
5.5.4 R4: Positioning 35
5.5.5 R5: Object Recognition 35
5.5.6 R6: Sensor Integration 35
5.5.7 R7: Information Aggregation 36
5.5.8 R8: Social Sharing 36

6 Implementation **37**
6.1 UbiQloud Overview 37
6.2 Server Components 39
6.3 UbiQloud Application 41
6.3.1 Recognize Module 42
6.3.2 Locate Module 43

6.3.3 Connect Module 44
6.3.4 Share Module 45
6.4 Communication with UbiQloud 45
6.4.1 RESTful API 46
6.4.2 Server Push Interfaces 48
6.4.3 Smart Gateway 50
6.4.4 Social Gateway 51

7 Sample Applications 53
7.1 Retail Context: FeedThroat 53
7.1.1 Integration to UbiQloud 55
7.1.2 User Tests 56
7.2 Positioning Context: InView 57
7.2.1 Integration to UbiQloud 58
7.3 Conclusions 58

8 Testing and Evaluation 60
8.1 Setup 60
8.2 Tools 61
8.3 Testing the Client Interface 63
8.4 Testing the Sensor Interface 64
8.5 Evaluation 65

9 Conclusions 67
9.1 Research Objectives Revisited 67
9.2 Main Contributions 67
9.3 Results 68
9.4 Future Work 69

List of Tables

5.1 Requirements for the platform. 33

6.1 UbiQloud modules. 42

List of Figures

1.1 Service categories of the designed platform. 3

2.1 Comparison between a push-based and pull-based interaction model. 5

2.2 The publish/subscribe interaction model. 6

2.3 The XMPP-based client/server interaction model. 7

2.4 The WebSocket-based client/server interaction model. 8

2.5 A *Smart Gateway* as mediator between an RFID reader communicating with TCP protocol and a Web application. 10

2.6 Terminal-based and network-based positioning systems. 13

2.7 *Social Login* steps with Facebook: authentication and application authorization. 16

2.8 Cloud computing models. 17

3.1 Facebook chat accessed through the official Facebook iPhone application and in the iChat desktop application. 21

3.2 Example of extending an existing system with an RFID middleware. 22

3.3 Position fingerprints on top of an indoor map. 24

3.4 Janrain Engage *Social Login* widget. 25

5.1 Different display types. 30

6.1 UbiQloud communication architecture. 38

6.2 Components of the UbiQloud platform. 40

6.3 Relations between the UbiQloud modules and requirements. . 43

6.4 Example of using the UbiQloud APIs for pushing the location information of a user. 44
6.5 Secure communication with the UbiQloud RESTful API. . . . 47
6.6 Six steps for *Social Login* through UbiQloud. 52

7.1 FeedThroat UI views in iOS application. 54
7.2 FeedThroat retailer view in the Web application. 55
7.3 InView UI views in Android and iOS devices. 57

8.1 Benchmark results for a single request response time against the client interface of the single and scaled UbiQloud with 50 concurrent users and 10000 requests. 63
8.2 Benchmark results for a single request response time against the sensor interface of the single and scaled UbiQloud with 50 concurrent users and 10000 requests. 64
8.3 Requests per second served during the tests. 66

Listings

2.1 Minimal message stanza. 7
2.2 Message stanza with all attributes. 7
2.3 Initializing a WebSocket object with JavaScript. 9
2.4 Example of a REST URI. 12
2.5 Example of the JSON representation of a resource. 12
2.6 Example of the XML representation of a resource. 12
2.7 Example of using the W3C Geolocation API with JavaScript. 14
6.1 Example of the JSON representation of a resource 46
6.2 Example of the XML representation of a resource 46
6.3 Example of an XMPP publish/subscribe message. 50
6.4 Example of embedding the UbiQloud *Social Login* widget. . . 52
8.1 Example of an Apache Bench script. 62
8.2 Example of an Apache Bench console output. 62

Chapter 1

Introduction

Over the years, the *World Wide Web (Web)* has evolved from a simple system for sharing static documents to a social and dynamic application platform [1]. According to Mikkonen and Taivalsaari [2], Web applications have numerous benefits in comparison to binary end-user software, such as instant, world-wide deployment; end-user independent upgrades; and platform independent access to data.

As the Web has evolved, different terms have been introduced to describe the current and future state of the Web. Murugesan [3] has divided the Web evolution into four distinct generations, as follows: *Information-centered* (legacy, Web 1.0), *People-centered* (current, Web 2.0), *Machine-centered* (upcoming, Web 3.0), and *Agent-centered* (future, Web 4.0). At the beginning (Web 1.0), the Web was a platform for sharing static documents. The current *People-centered* Web (Web 2.0) emphasizes *Social Media* and has somewhat migrated the content providers and the content consumers to a single group of Web users. The next generation (Web 3.0) emphasizes machine to machine communication and makes the Web more context-aware and collaborative platform through data and device integration (i.e., the *Web of Things, WoT*). The *Agent-centered* generation sees the Web evolving to an ubiquitous environment seamlessly integrating both human and machine intelligence.

In addition to the evolution of the Web, the ways of interacting with Web applications have also changed drastically. Traditionally, the Web was accessed from a personal computer via a Web browser and the applications were controlled through standard input devices, such as a keyboard or a mouse. Nowadays, the Web can be accessed with numerous ways with an increasing range of different devices (e.g., smartphones and tablets) and input methods, such as touch. These changes have introduced new requirements as well as problems for Web applications and the Web itself [4]. Users and devices are connected to the Web all the time and everywhere. In addition, data needs to be (1) up-to-date (updates pushed in real-time), (2) context-aware, and (3) personalized.

First of all, the problem is that in the Web, the communication has been primary relying on pull-based protocols (e.g., *HyperText Transfer Protocol, HTTP* [5]), where client always needs to request (pull) the data from a server (i.e., the server cannot push updates not specifically requested by the client). The pull-based approach is not sufficient enough for the near real-time requirements of the modern Web applications [6].

Second of all, Web application data needs to be accessed and updated from different devices and with different input methods [4], and thus, it is not sufficient enough to provide a single Web *User Interface (UI)* for interacting with the application. The application needs to provide an *Application Programming Interface (API)* for accessing and updating the data in different formats with different access rights (i.e., user specific data and public data). Furthermore, as more and more physical devices are connected to the Web, the data should be accessible in formats understandable to both human and machine consumers (e.g., *HyperText Markup Language (HTML)* [7] for humans and *Extensible Markup Language (XML)* [8] or *JavaScript Object Notation (JSON)* [9] for machines).

Third of all, as Web applications are becoming more social and more connected with other applications and physical devices, the complexity of the application development increases. Developing a Web application without any API dependencies is already a complex task as is [10], but developing and maintaining a mashup application increases the complexity even more. Firstly, the application developer needs an implementation for each of the APIs used and secondly, the developer needs to keep track of changes in third-party APIs to maintain a fully functional application.

This Thesis focuses on addressing the aforementioned issues by presenting a powerful and scalable cloud-based platform, UbiQloud, for implementing real-time, context-aware, social and platform independent Web 3.0 applications. The platform allows developers to integrate physical devices to the system; use existing social media services for user management, collaboration, and sharing through unified APIs; and access as well as publish data in real-time via modern push-based protocols. The services provided by the platform can be divided into four distinct categories (i.e., server push, WoT, positioning, and social media) as illustrated in Figure 1.1.

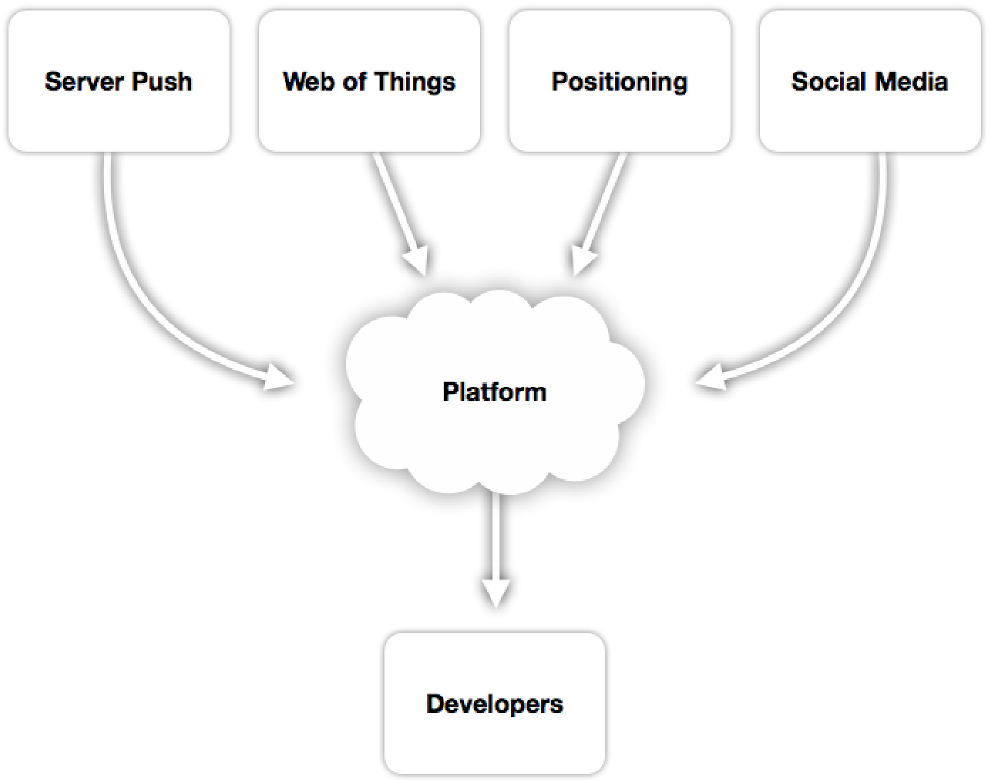


Figure 1.1: Service categories of the designed platform.

1.1 Organization of the Thesis

The rest of the Thesis is organized as follows. Chapter 2 covers the technical and conceptual background information relevant to the topic at hand. In Chapter 3, a state-of-the-art review of related research and existing implementations is presented. Next, the research aims, questions, scope, and settings are introduced in Chapter 4. Then, the requirements for the platform are presented in Chapter 5. Chapter 6 covers the proof-of-concept implementation of the platform, and Chapter 7 gives an overview of the sample applications developed on top of the platform. In Chapter 8, the evaluation tests are presented, and the results are analyzed and discussed. Finally, Chapter 9 concludes the Thesis.

Chapter 2

Background

In this chapter, an introduction to the concepts and technologies relevant to this Thesis is given. The first section describes the basics of server push and gives an overview of modern server push technologies, including XMPP and WebSocket. Then, the concept of the *Web of Things (WoT)* is presented in detail, followed by the description of technologies and problems in positioning. Next, the social media and *Social Login* are covered, and lastly the basics of cloud computing models are introduced.

2.1 Server Push

HyperText Transfer Protocol (HTTP) [5], which is the basis of communication on the Web, is based on the request/response paradigm. This means that all HTTP-based communication is based on request/response pairs always initiated by the client. After each response, the connection is closed and a new request must be made in order the receive new data. The server needs to wait that the client requests new data, before it can be sent to the client. As the Web has evolved from a simple document sharing system to a dynamic application platform, the pull-based technique is no longer sufficient to fulfill the requirements of modern Web applications [6].

To overcome the real-time limitations of HTTP, servers need to be able to push data to the clients. Server push is not a protocol like HTTP but a paradigm for techniques allowing robust communication between clients and servers. Regardless of the implementation, the common factor in all server push systems is the ability to push changes through permanent channel from the server to the client as soon as the changes occur on the server [6]. In other words, server push allows clients to utilize real-time data channels from a server to a client in parallel with the traditional information pull. Figure 2.1 demonstrates the difference between a push-based and a pull-based system. Even though, the figure shows two different clients, they could also be the same client utilizing both techniques. In addition, the permanent connection between client and server can be either one-way or two-way channel (i.e., data can be sent only from server to client or both ways).

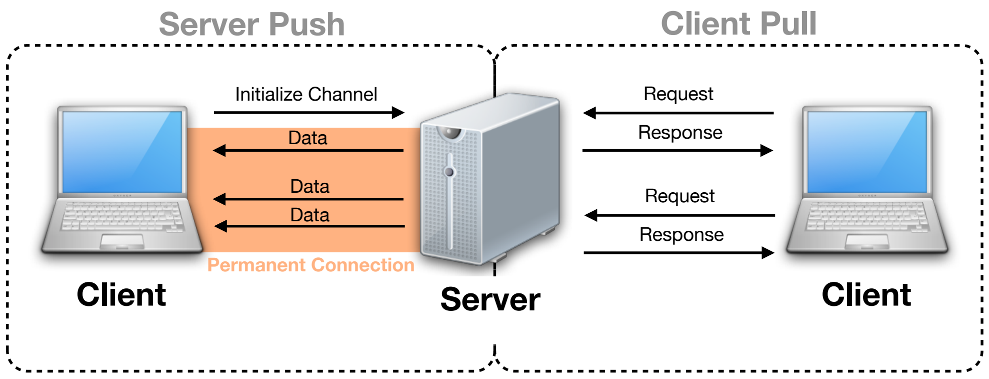


Figure 2.1: Comparison between a push-based and pull-based interaction model.

2.1.1 Publish/Subscribe

Publish/subscribe is an architecture in which the information is collected from a number of sources (publishers) and delivered to a number of interested consumers (subscribers) in an anonymous and asynchronous manner. In a publish/subscribe system, a publisher(s) publishes data in a node(s) and the data is then pushed to all subscribers of that node, if any. A publisher does not need to know the identities of the subscribers or wait for subscribers requests. The data is simply multicasted from a publisher to a number of subscribers in real-time. [11] Figure 2.2 illustrates the interaction model between publishers and subscribers in a publish/subscribe system. In the figure, there are two publishers pushing weather information to four nodes. One of the nodes is used by both of the publishers and the other three is used by a one publisher only. There are also three clients subscribed to a number of nodes. One of the nodes (*Helsinki Weather*) has no subscribers, but new data is still pushed to it because the publisher is not aware of the subscribers.

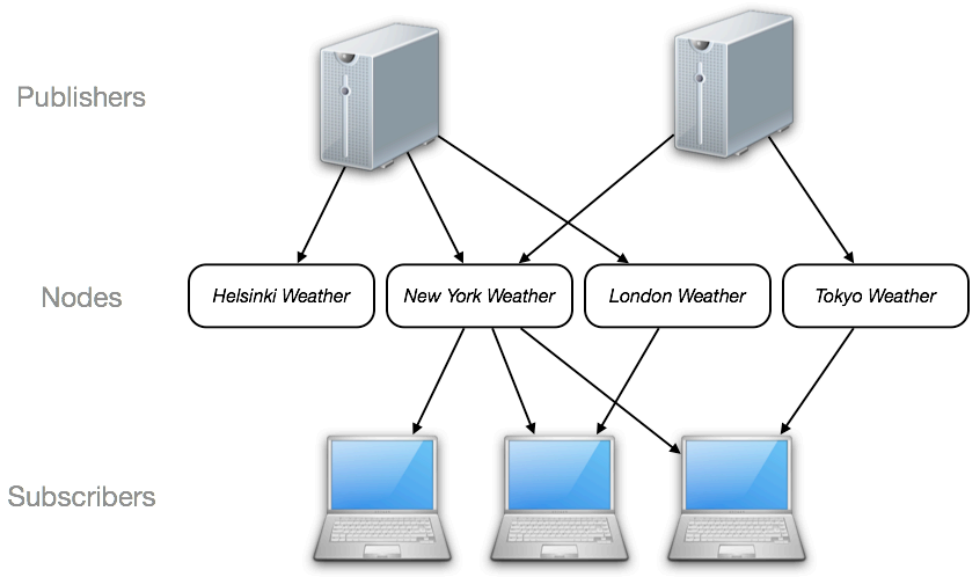


Figure 2.2: The publish/subscribe interaction model.

2.1.2 XMPP

Extensible Messaging and Presence Protocol (XMPP) [12] is an XML-based, application-level protocol for exchanging structured data between any network entities in near real-time. XMPP was originally developed under the name *Jabber* in the Jabber open-source community for instant messaging (IM) and presence applications, such as chats with authenticated users [13]. Later, the core of the Jabber protocol was revised and formalized by the *Internet Engineering Task Force (IETF)*[1], and published under its current name XMPP in their *Request for Comments (RFC)* series as RFC 6120 [14] and RFC 6121 [15]. In addition, the *XMPP Standards Foundation (XSF)*[2] has developed and published over 300 *XMPP Extension Protocols (XEPs)*[3] to support a wide variety of application scenarios, such as *XEP-0206: XMPP Over BOSH*, which is an HTTP binding for XMPP communications [16] and *XEP-0060: Publish-Subscribe* [17] for publish/subscribe systems.

Similar to the Web, also XMPP is based on a decentralized client/(server architecture. When an XMPP client wants to start a session with an XMPP server, it opens an XML stream over a long-lived connection (e.g., Trans-

[1]Internet Engineering Task Force, http://www.ietf.org/
[2]XMPP Standards Foundation, http://xmpp.org/about-xmpp/xsf/
[3]XMPP Extension Protocols, http://xmpp.org/xmpp-protocols/xmpp-extensions/

mission Control Protocol, TCP [18]) to the server. Next, the server opens another XML stream to the client, resulting in two XML streams over a single TCP socket, one in each direction. Figure 2.3 illustrates the XMPP interaction model between the client and the server.

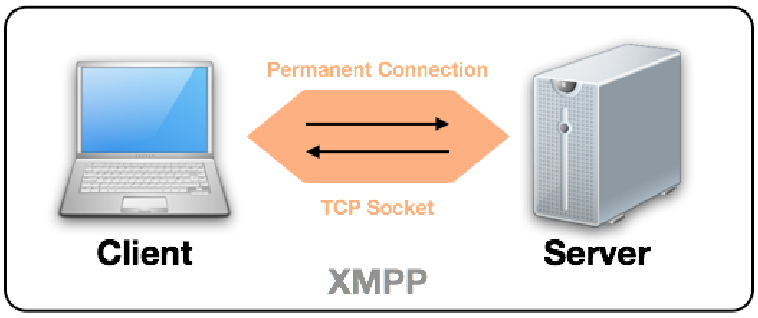


Figure 2.3: The XMPP-based client/server interaction model.

After the connections have been established, each entity can asynchronously exchange an unbound number of special XML snippets over the streams. These special XML snippets, called *XML stanzas*, define the basic units of communication in XMPP and are as follows: `message`, `presence`, and `iq` (Info/Query). The *Message Stanza*, which is the basis of client to client message transfer in XMPP, consists of a root element called *<message>* and a child element called *<body>*, the latter being a wrapper for the actual message payload. In addition, the root element contains 1-5 attributes that are: *to* (mandatory), *from*, *id*, *type*, and *xml:lang*. Listings 1 and 2 show examples of an XMPP message with a minimum and maximum amount of attributes.

```
<message to="{username}@{domain}">
  <body>{payload}</body>
</message>
```

Listing 2.1: Minimal message stanza.

```
<message to="{username}@{domain}/{resource}"
  from="{username}@{domain}/{resource}"
  id="{messageId}"
  type="{chat|error|groupchat|headline|normal}"
  xml:lang="{xmlLang}">
  <body>{payload}</body>
</message>
```

Listing 2.2: Message stanza with all attributes.

Even though XMPP provides a rich set of features and is a widely used protocol for real-time communication, it is hardly used on the traditional Web environment. There is two main reasons for this: (1) Web browsers do not provide native XMPP support and (2) regular XMPP communication is usually blocked by firewalls and proxies. Nevertheless, XMPP can be used in the Web with push techniques (e.g., BOSH [19]) supported by Web browsers.

2.1.3 WebSocket

WebSocket [20] is a new protocol for real-time communication on the Web. The basic idea of the protocol is to provide a full-duplex, bi-directional communication channel over a single TCP socket (cf. Figure 2.4). The WebSocket protocol can be used on a standard Web environment and by default it uses the same ports as the HTTP communication. Although WebSocket uses the same underlying Web infrastructure, it has very little to do with the HTTP protocol. The only similarity is that the connection is established with an HTTP Upgrade request. After the connection is established, there is a permanent communication channel between the client and server through out the session (i.e., no need to establish a new connection per request basis as in the HTTP). WebSocket is designed to be as raw as possible to minimize the network overhead and to allow more complex protocols, such as XMPP [21], to be used on top of it.

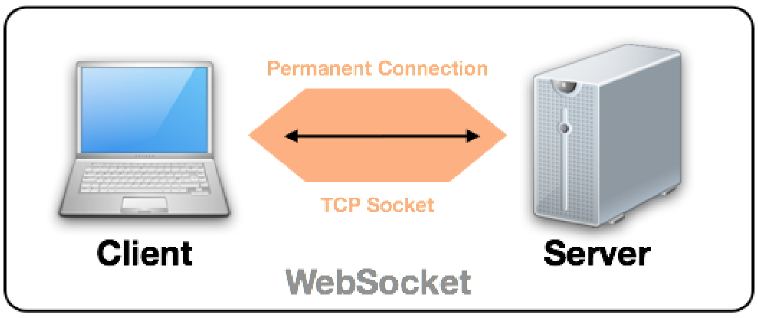


Figure 2.4: The WebSocket-based client/server interaction model.

In addition to the WebSocket protocol, there is a WebSocket API [22] designed in conjunction with the protocol by the *World Wide Web Consortium (W3C)*. The WebSocket API provides a JavaScript interface through which a client can interact with the browser's implementation on WebSocket.

Listing 2.3 shows an example of initializing a WebSocket object and attaching event listeners to it. On the first line, the WebSocket object is initialized with two parameters: (1) the URL of the connection endpoint, and (2) the protocol (e.g., XMPP) used on top of the WebSocket connection. The URL format is otherwise similar to HTTP, but it uses `ws(s)` instead of `http(s)` as the URL schema. The protocol parameter is optional, but can be used for example in the server side to only accept connections using certain protocols. The rest of the example shows how to listen when the connection is opened, an error has occurred, or a message is received. Although the interface is designed for JavaScript, it can be implemented in other languages as well.

```
var socket= new WebSocket("ws://example.com", "xmpp");
socket.onopen = function () {
  alert("WebSocket Open!");
};
socket.onerror = function () {
  alert("WebSocket Error!");
};
socket.onmessage = function (data) {
  alert("WebSocket Message: " + data);
};
```

Listing 2.3: Initializing a WebSocket object with JavaScript.

2.2 Internet of Things / Web of Things

According to Krannenburg [23], the *Internet of Things (IoT)* is an information architecture that extracts data from a network of devices and objects. On the other hand, IoT semantically is

> *"a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols." [24]*

In short, IoT is a goal in which things — that is, physical objects or devices — constitute a network capable of communicating and interacting with each other.

The *Web of Things (WoT)* can be viewed as a step towards reaching the Internet of Things vision and it basically means using the Web and Web technologies as means of connecting smart objects with the existing Web environment [25]. Using Web standards to interconnect objects raises the problem that in order to establish a connection, the object needs to understand these standards. In the future, it is possible that most smart objects have an embedded web server, but for constrained objects we need a mediator, referred as a *Smart Gateway* [25], that is capable of communicating with the object through proprietary/non-Web protocols and with the outside world through Web-based protocols. Figure 2.5 demonstrates a situation in which an *Radio Frequency Identification (RFID)* reader capable of communication via TCP is exposed to a Web application through a *Smart Gateway*.

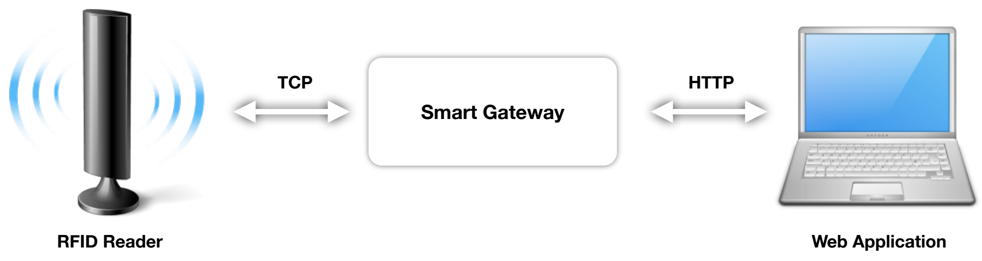


Figure 2.5: A *Smart Gateway* as mediator between an RFID reader communicating with TCP protocol and a Web application.

2.2.1 Smart Object

Kortuem et al. [26] define a *Smart Object* as autonomous object that possesses sensing, processing, and networking capabilities. On the other hand, a *Smart Object* can be considered as an object with a unique addressing schema capable of interacting and cooperating with each other [27]. Based on these definitions, a *Smart Object* can be something as simple as a plain object equipped with a RFID tag or something 'smarter', such as a *Wireless Sensor Network (WSN)*, an embedded device, or a smartphone. All in all, the key thing is that the object can be uniquely identified and is capable of autonomous interaction with other objects through a medium, such as the Web.

2.2.2 Smart Space

According to Kawashima et al. [28], a *Smart Space* is a physical environment equipped with sensing devices and actuation devices for contextual information collecting and context-aware responses. In general, a *Smart Space* is a space capable of autonomous, context-aware actions based on condition events generated by interconnected *Smart Objects*. For example, a smart home could autonomously adjust lightning and heating based on weather conditions (e.g., summer day or winter night), or a car could adjust seats and mirrors based on the driver identified by the car key.

2.2.3 Representational State Transfer

In order to connect physical objects as part of the Web, there needs to be a way to represent and access these objects on the Web environment. One viable method [29] is to utilize a concept called *Representational State Transfer (REST)* [30]. REST is a client/server architecture, in which a server contains resources with unique identifiers. When a client requests a resource from the server, the server responds with a representation of that resource. When the client moves from one representation to another, it also moves to a different state. The REST architectural style is based on six principles as follows:

Uniform Interface
Implementations are decoupled from the services they provide,

Client/Server
There is a clear separation (i.e., uniform interface) between clients and servers,

Stateless
No client context is stored on the server,

Cache
Responses must be labeled as cacheable or non-cacheable,

Layered System
A client cannot tell whether it is connected to the end server or to an intermediary, and

Code-On-Demand (Optional)
Servers are able to extend client functionality by transferring executable code (e.g., JavaScript).

HTTP-based Web services that follow the principles of REST (potentially excluding the *Code-On-Demand* constraint) are referred as RESTful Web services. In RESTful Web services, all resources are identified with a *Unified Resource Identifier (URI)* and accessed with common HTTP methods *GET*, *POST*, *PUT*, and *DELETE*.

- *GET* is for retrieving a resource,
- *POST* is for creating a resource without existing identifier,
- *PUT* is for creating or modifying a resource with known identifier, and
- *DELETE* is for removing a resource.

A representation of a resource can vary based on the request (e.g., HTML for a Web browser and JSON for a server). For example, consider a *Smart Space* environment where users can control lamps through a RESTful interface. Each lamp is identified with an unique URI as follows:

```
    http://example.com/lamp/{identifier}
```

Listing 2.4: Example of a REST URI.

The {identfier} is a unique property that distinguishes one lamp from another. So executing the following request:

HTTP GET: `http://example.com/lamp/3`

could result the following response in JSON:

```
{
"lamp" :
  {
  "id" : 3,
  "on" : true
  }
}
```

Listing 2.5: Example of the JSON representation of a resource.

or in XML:

```
<lamp>
  <id>3</id>
  <on>true</on>
</lamp>
```

Listing 2.6: Example of the XML representation of a resource.

2.3 Positioning

Positioning objects can be achieved with a wide variety of technologies, including Global Positioning System (GPS), Global System for Mobile Communications (GSM), Wireless Local Area Network (WLAN), and Bluetooth.

The accuracy of each technology is highly dependable of the environment it is used and there is no single technology capable of accurate positioning both indoors and outdoors. For example, GPS is capable of positioning objects with a 10 meter accuracy most of the time outdoors, but is almost unusable indoors because it requires a line of sight to the satellites. [31]

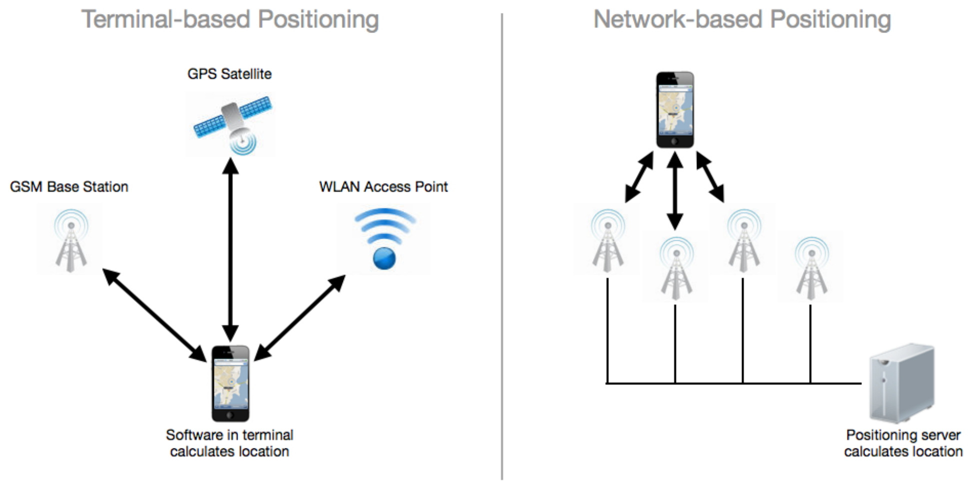

Terminal-based Positioning
GPS Satellite
GSM Base Station
WLAN Access Point
Software in terminal calculates location
Network-based Positioning
Positioning server calculates location

Figure 2.6: Terminal-based and network-based positioning systems.

Positioning systems can be divided into two categories [31], *Terminal-based* systems and *Network-based* systems, as illustrated in Figure 2.6. In the *Terminal-based* systems, the location is calculated on the terminal (e.g., smartphone) itself by using radio measurements and predefined antenna locations (e.g., GPS satellite or WLAN base stations). The *Network-based* positioning systems, on the other hand, use a dedicated positioning server attached to a network of antennas to calculate the terminal location. The latter is usually more accurate since the positioning server possesses more fine grained data of antennas. However, the solution does not scale well, because it relies on a fixed network of antennas.

While outdoor positioning is achieved relatively easy with GPS, indoor positioning is much harder in general. The biggest issue in indoor positioning is usually the physical facility because these facilities often contain places hard or impossible to detect using positioning technologies (e.g., no line of sight, reflecting surfaces, and thick walls). [32] These obstacles can be circumvented by installing enough hardware, but it is often too expensive and the solution is tied to a particular location. At the moment, the most accurate indoor positioning systems (e.g., Skyhook[4]) use a hybrid approach by combining multiple positioning technologies.

To ease the development of location-aware applications, W3C has proposed a standard [33] for a Geolocation API. The idea is to provide a standardized interface for developers to access location data. W3C does not guarantee the accuracy of the location data gained through the API, since the device implementing the API is responsible for providing the data, and the technologies for gaining location data vary as stated above. The interface itself is quite simple. It extends the *Navigator* object with a `geolocation` attribute, which in turns implements the *Geolocation* interface that has methods for gaining a current position, watching position changes, and canceling a watch. Listing 2.7 illustrates the use of the Geolocation API with JavaScript. In the example, two location related functions are executed at application startup. First, the current location is determined, and second, a listener that continuously watches the position is initialized.

```
window.onload = function() {
  var geoprovider = navigator.geolocation;
  geoprovider.getCurrentPosition(function(position) {
    var lat = position.coords.latitude;
    var lon = position.coords.longitude;
    alert("Started from location: " + lat + " " + lon);
  });

  geoprovider.watchPosition(function(position) {
    var lat = position.coords.latitude;
    var lon = position.coords.longitude;
    alert("Current location: " + lat + " " + lon);
  });
};
```

Listing 2.7: Example of using the W3C Geolocation API with JavaScript.

2.4 Social Media

Mayfield [34] describes social media as a group of online media sharing similar characteristics, such as participation, conversation, and community. Based on the characteristics, he has divided the social media into six distinct groups as follows: social networks, blogs, wikis, podcasts, forums, and content communities. As described in Chapter 1, the current phase (Web 2.0) in the Web evolution is *People-centric* and social media is one massive example of

[4]Skyhook, http://www.skyhookwireless.com/

that (e.g., Facebook had 526 million active daily users on average in March 2012[5]). In general, social media has provided the ability for the content consumers to become content providers on the Web and allowed users to build themselves a digital identity that can be used in the digital world.

2.4.1 Social Login

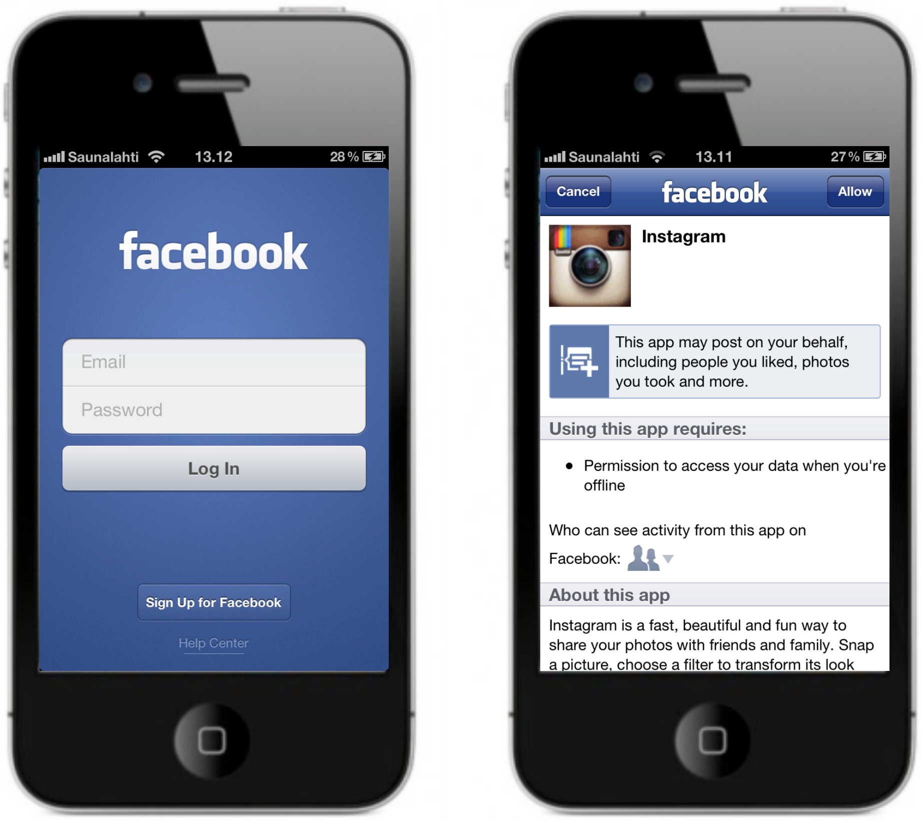
Identifying users is a vital part of social media. As social media has became such a popular phenomena, the number of social media applications and social networks has increased drastically [35]. Using different credentials in every social service is a major problem for both the users and developers of social Web applications. From a user's point of view, it is tedious to manage multiple different user accounts that often leads to security risks because same passwords are used in many services. From a developer's point of view, creating a new social network has became harder, since users are not willing to create yet another user profile. [36]

The problem of managing multiple user accounts has inspired a number of solutions to reduce the need of creating a new account for every social media service. The first solution was to use an ID management system (e.g., OpenID[6]), which allowed a user to authenticate to a Web service through a third-party authentication provider [36]. OpenID-like systems do solve the problem of managing credentials for multiple services, but these systems are only focused on authentication and lack other vital aspects of social media, such as creating and maintaining social graphs (e.g., friend connections). To allow a centralized *Social Login* — that is, authentication and authorization (e.g., OAuth[7]) — many of the popular social media services have started to provide services that allow developers to use their service both as an authentication provider and as a profile management system [37]. In other words, through *Social Login* services developers are able to utilize the existing profile and social graph of a user, which eliminates the need for a developer to provide a custom profile management system, and the need for a user to create a new profile and to build a new social network. Figure 2.7 illustrates the workflow of the *Social Login* with Facebook. First, the user needs to login to Facebook, and second, the user needs to give permissions to a third-party application to use the user's profile data.

[5]Facebook Key Facts, http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

[6]OpenID, http://openid.net/

[7]OAuth, http://oauth.net/



This app may post on your behalf, including people you liked, photos you took and more.

- Permission to access your data when you're offline

Who can see activity from this app on Facebook:

Instagram is a fast, beautiful and fun way to share your photos with friends and family. Snap a picture, choose a filter to transform its look

Figure 2.7: *Social Login* steps with Facebook: authentication and application authorization.

2.5 Cloud Computing

Cloud computing is a broad term used to describe different things in different contexts. Böhm et al. [38] discuss about the problems of giving a precise definition for cloud computing. In their research, they have conducted a list of common characteristics for cloud services, and based on that they have defined cloud computing as an

> *"IT deployment model, based on virtualization, where resources, in terms of infrastructure, applications and data are deployed via the internet as a distributed service by one or several service providers. These services are scalable on demand and can be priced on a pay-per-use basis."*

In comparison, Mirashe and Kalyankar [39] define cloud computing as an inter-connected cluster of computers and servers hosting applications and files accessible through the Web . Cloud computing can be viewed from the hardware point of view (the infrastructure that enables the cloud computing) or from the software point of view (the services provided as cloud services). In general, cloud computing means that applications and data used to be stored locally on a single computer are now stored in a scalable remote location, which can be accessed through the Web.

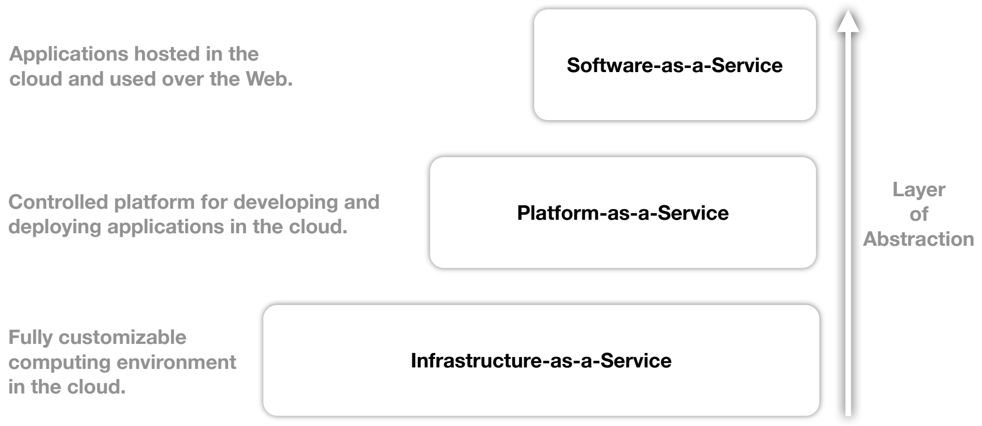

Applications hosted in the cloud and used over the Web.
Software-as-a-Service
Controlled platform for developing and deploying applications in the cloud.
Platform-as-a-Service
Fully customizable computing environment in the cloud.
Infrastructure-as-a-Service
Layer of Abstraction

Figure 2.8: Cloud computing models.

As cloud computing as a term has maturated, cloud-based services have been divided into different categories. The most common division include three categories as follows:

1. Infrastructure-as-a-Service (IaaS),
2. Platform-as-a-Service (PaaS), and
3. Software-as-a-Service (SaaS). [40]

The categories can be viewed as the layers of abstraction in cloud computing [38] as illustrated in Figure 2.8. IaaS refers to a service providing a whole computing infrastructure including computational resources, data storage, and communication. Virtual Machines (e.g., Amazon Elastic Compute Cloud, Amazon EC2[8]), are good examples of IaaS. Where IaaS offered a

[8]Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/

fully controllable infrastructure, PaaS offers a scalable but restricted platform to develop and possibly deploy applications. The platform usually provides APIs, authentication services, communication services, libraries and UI components for application development and often a runtime environment for application deployment. Google App Engine[9] is a widely used example of PaaS. The topmost category, SaaS, refers to a service providing desktop-like applications through the Web. A good example of SaaS is Microsoft Office 365[10] that provides a Web-based version of its widely used Office software suite.

[9]Google App Engine, https://developers.google.com/appengine/

[10]Microsoft Office 365, http://office365.microsoft.com

Chapter 3

State-of-the-Art

In this chapter, the concepts and technologies discussed in Chapter 2 are covered in more detail by introducing current research activities and existing services related to each topic. First, research concentrating on platforms for WoT and mobile applications are covered. Then, existing services offering features from one or many of the topics covered so far (i.e., server push, WoT, positioning, and *Social Login*) are introduced.

3.1 Related Research Activities

The Web of Things can be considered as a relatively new concept in research, although, for example, Ljungstrand et al. proposed a way to link physical objects to the Web over a decade ago [41]. However, the older research mostly concentrated on creating a virtual representation of real-world objects rather than actually integrate smart objects as part of the Web.

Recently there has been more research on creating solutions capable of controlling smart objects through Web-based protocols. For example, there has been a proposal for a RESTful architecture for the Web of Things [29] and a publish/subscribe-based solution for RESTful messaging for devices [42]. Both of these solutions provide a way to control smart objects, represented as resources, through a RESTful interface with common HTTP methods.

Blackstock et al. [43] proposed a *MAGIC Broker (MB2)* architecture for the Internet of Things. The MB2 is a publish/subscribe architecture based on the OSGi framework. Blackstock et al. also discuss how the MB2 architecture can be interconnected to social web services. They used available social Web APIs and created, for example, Facebook applications for real-world objects.

In other words, the MB2 acts as a gateway between social Web applications and real-world objects.

Kawashima et al. have implemented a *Gateway-Based Ubiquitous Platform for Smart Space (GUPSS)*, which is a general platform for *Smart Spaces* consisting of *Smart Space* gateways, *Smart Space* applications and a *Smart Space* server [28]. GUPSS focuses more on providing a centralized server to manage *Smart Spaces* rather than exposing smart objects to the Web.

Springer et al. [44] presented Mobilis, which is a middleware platform for collaborative mobile applications. The platform provides a set of collaborative services that are *Context Management* service, *Geolocation* service, *Multimedia Sharing* service, *Group Chat* service, *Collaborative Drawing* service, *Group Management* service, and *Multimedia Tagging* service. The communication architecture of the platform is based on the publish/subscribe paradigm and realized using the XMPP protocol and related XEPs. Mobilis can be used [45, 46] for developing collaborative location-based Android[1] mobile applications that integrate existing social media networks. For extending the support of Mobilis platform beyond Android-based applications, Jansen [47] has introduced a Web gateway for the platform that supports BOSH for real-time communication between Mobilis and Web-based applications.

3.2 Existing Services

This section provides an overview of existing commercial platforms providing services for server push, WoT, positioning, and social media integration. The separation of the platforms is made based on the core services provided, although some platforms would fit under multiple topics.

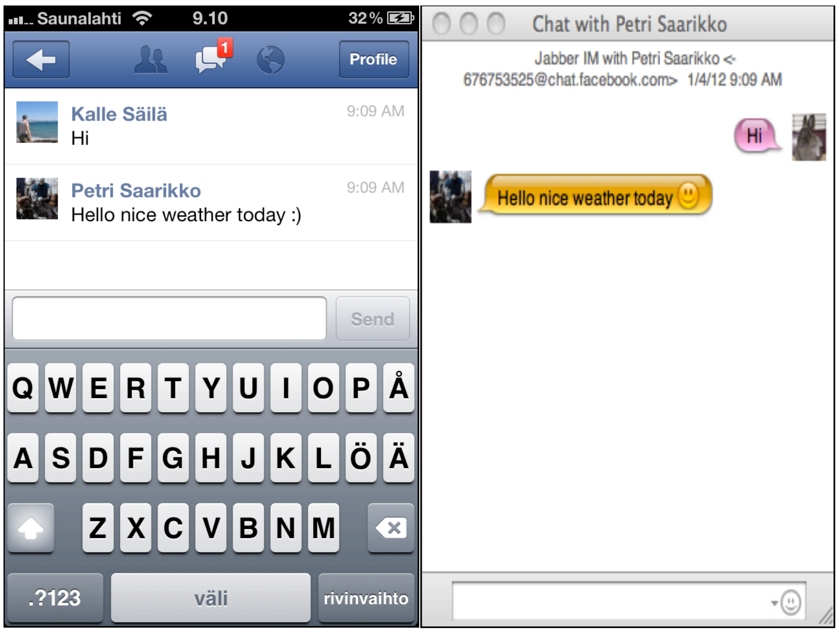
3.2.1 Server Push Services

As mentioned in Section 2.1, the future of the Web is likely to be more and more push-based rather than pull-based. Especially the maturing HTML5 [7] and associated specifications, along with WebSocket implementations in Web browsers, have generated new kinds of services relying on real-time data. These new kinds of services vary from asynchronously updated user interfaces and simple chats to cloud-based services offering server push to third-party applications.

[1]Android, http://www.android.com/

Pusher [48] is one of the WebSocket-based services offering APIs to add real-time functionality to third-party applications. Pusher allows developers to create authenticated publish/subscribe channels for real-time event notifications and a REST-based publishing mechanism. According to Pusher's website, there are already several well-known services using its push service, including Groupon[2] and Slideshare[3].

Many Web applications have also added real-time communication features as part of their applications. For example, Facebook [49] has also added real-time functionality, including a chat, as part of their service. In addition to using the chat through an official Facebook application (e.g., Web application or iPhone application), there is also an API for third-party applications willing to access the chat. The chat API[4] enables integration of the Facebook chat to other instant messaging services through an XMPP gateway. Figure 3.1 shows the Facebook chat in the native Facebook iPhone application and in the iChat desktop application.



Kalle Säilä 9:09 AM
Hi

Petri Saarikko 9:09 AM
Hello nice weather today :)

Chat with Petri Saarikko

Jabber IM with Petri Saarikko <676753525@chat.facebook.com> 1/4/12 9:09 AM

Hi

Hello nice weather today

Figure 3.1: Facebook chat accessed through the official Facebook iPhone application and in the iChat desktop application.

[2]Groupon, http://www.groupon.com
[3]Slideshare, http://www.slideshare.net
[4]Facebook Chat API, https://developers.facebook.com/docs/chat/

3.2.2 Web of Things Services

Many kinds of services can be consider WoT related. The main thing in common with all the services provided under the term is that there are objects connected to the Web. One of the most simple forms of WoT-based services is an RFID platform capable of identifying RFID tags. For example, LogicAlloy's ALE Server [50] is an RFID middleware for integrating RFID hardware with existing systems. It has support for the leading RFID readers and tag standards. In addition, it provides a SOAP [51] API for an easy integration with pre-existing systems. Figure 3.2 demonstrates the basics of an RFID middleware capable of communicating with RFID readers, processing RFID tag data, and mediating that data through APIs.

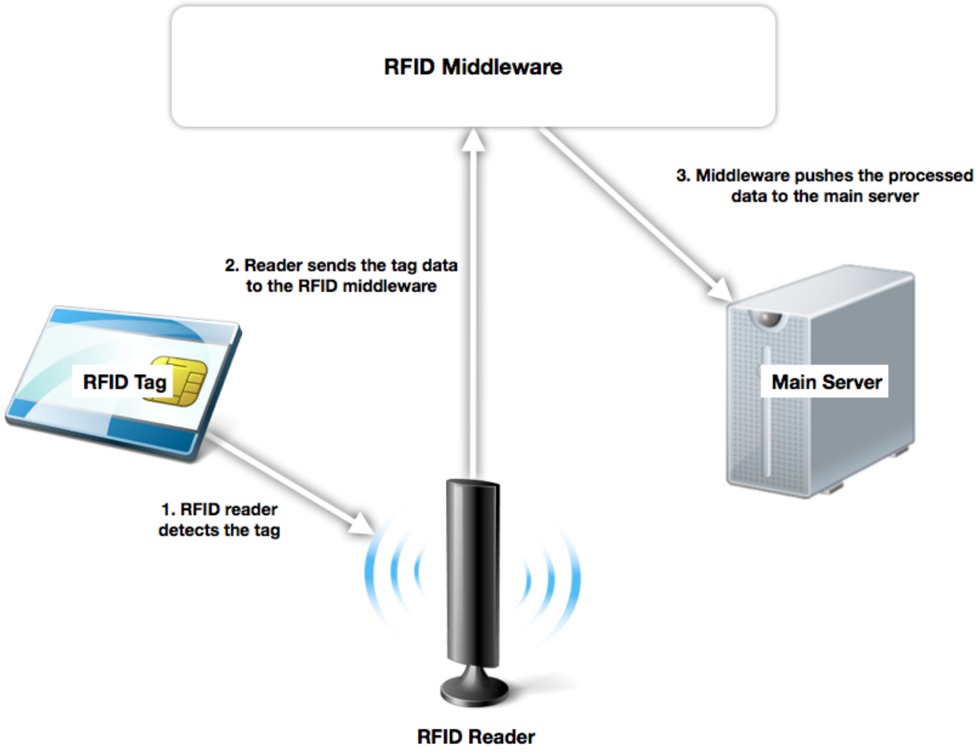

RFID Middleware
3. Middleware pushes the processed data to the main server
2. Reader sends the tag data to the RFID middleware
RFID Tag
Main Server
1. RFID reader detects the tag
RFID Reader

Figure 3.2: Example of extending an existing system with an RFID middleware.

Cosm [52] is a Web-based platform, to which users can connect different kinds of sensors and the sensor data is then pushed into dedicated channels in real-time. The main goal of Cosm is to facilitate the development of IoT applications by acting as a data broker between smart objects and applications. It provides push-based and pull-based APIs for accessing the data in different formats (e.g., XML and JSON) as well as ready-made applications, gadgets, and tools leveraging its APIs.

3.2.3 Positioning Services

There is a wide variety of platforms on the Web (e.g., Google Maps[5] and Bing Maps[6]) that offer services for developing location-based applications. Common features of these services include map tiles, geocoding (converting textual data, such as street address, to coordinates), and reverse geocoding (converting coordinates to textual data, such as street address). These platforms rely on internal map data and external input data (e.g., an address typed in by a user or a latitude/longitude pair captured by a smartphone through GPS), which makes these services a viable solution for outdoor, location-based applications where it is enough to position within a ten meter radius.

As stated before, it is much harder to provide indoor positioning services, because usually there is no publicly available maps of indoor locations and the current positioning solutions are not accurate enough indoors. However, to overcome the problems of indoor positioning systems, new kinds of platforms using hybrid positioning techniques and user generated data have emerged on the Web. For example, Qubulus [53] is a platform where developers can import indoor maps, fingerprint (cf. Figure 3.3) a physical location with a dedicated recording tool, and enable radio mapping based indoor positioning in that location. Google has also started[7] to add indoor maps and indoor location services to their positioning platform. These kind of approaches make it possible to develop accurate, location-based applications for indoor environments. However, the downside is that the developer needs map every location separately.

[5]Google Maps Developer Site, https://developers.google.com/maps/

[6]Bing Maps Developer Site, http://www.microsoft.com/maps/developers/web.aspx

[7]Google's announcement of indoor maps, http://googleblog.blogspot.com/2011/11/new-frontier-for-google-maps-mapping.html

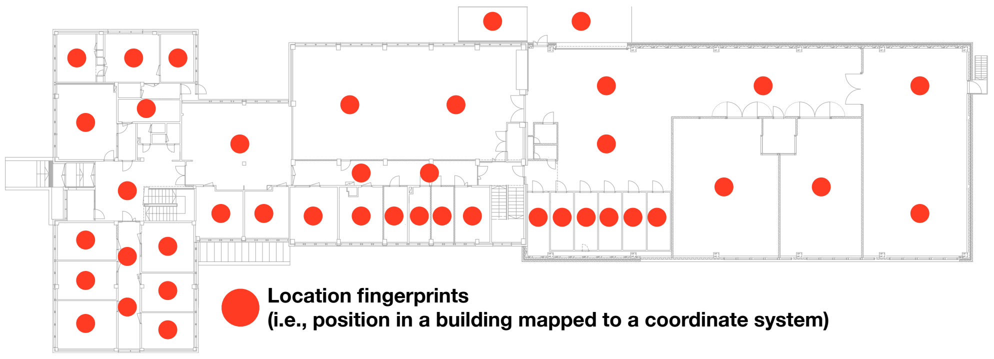

Location fingerprints
(i.e., position in a building mapped to a coordinate system)

Figure 3.3: Position fingerprints on top of an indoor map.

3.2.4 Social Media Gateways

The most popular social media services (e.g., Facebook[8] and Twitter[9]) offer a *Social Login* (cf. Section 2.4.1) to third-party applications. From a developer's point of view, the problem is that the structure of the APIs and the ways to interact with those APIs differ per social media service basis, meaning that if a developer wants to support a variety of social media services, each of the APIs need a separate, and potentially really different, implementation. To overcome the problem of the different implementations of the *Social Login* systems, some service providers have started to provide *Social Media Gateways* that offer an unified mechanism to integrate multiple social media services to an application.

Gigya [54] and Janrain [55] are both services that act as a gateway between an application and a range of social media services. Both Gigya and Janrain provide user management, JavaScript-based widgets (common social media activities, such as login and sharing, through a UI) that can be embedded to a Web application, and an unified API-level access to social media services. Figure 3.4 shows a Social Login widget[10] provided by Janrain.

[8]Facebook, http://www.facebook.com/
[9]Twitter, https://twitter.com/
[10]Janrain Engage *Social Login* widget, http://janrain.com/wp-content/uploads/2012/03/Janrain-Engage-01.png

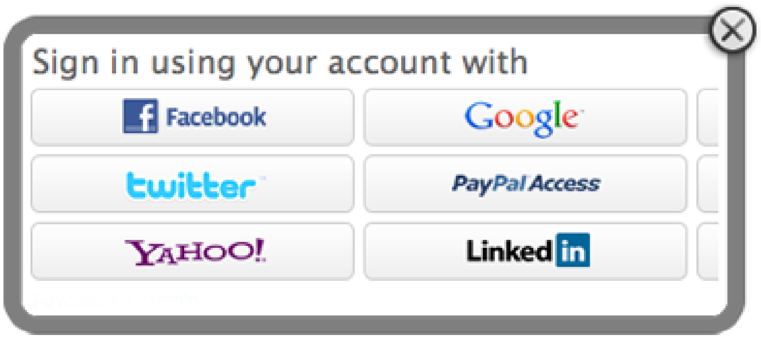


Figure 3.4: Janrain Engage *Social Login* widget.

For any application developer that wishes to integrate one or multiple social media services into their application, *Social Media Gateways* provide a fast and relatively easy way to reach a massive amount of users and viable profile data associated to those users.

Chapter 4

Research Aims

In this Chapter, the research settings of this Thesis are covered. First, the research objectives and scope are introduced. Then, the research problems are presented and the research questions are formulated. Last, the methods used for the research are demonstrated.

4.1 Research Objectives and Scope

The main research objective of this Thesis is two-fold. Firstly, the objective is to research the current solutions for controlling and creating *Smart Spaces* as well as the possibilities of integrating existing devices and objects as part of *Smart Space* controlling platforms. Secondly, the objective is to implement a proof-of-concept cloud-based platform for developing social *Smart Space* applications. In addition, the usefulness and the performance of the platform is tested by developing two sample applications on top of it as well as by measuring the performance of the platform with a series of automated test runs.

Two kinds of premises have been used as a basis for *Smart Spaces* in this research, although the platform is designed for applications used in any kind of *Smart Space*. The primary *Smart Space* context of this research is a shopping center (Iso Omena[1] shopping center for user tests) and the secondary *Smart Space* is a living lab at Aalto Design Factory[2]. The requirements analysis for the platform is based on the shopping center as a *Smart Space* and both contexts are used for the evaluation of the platform. The shopping center

[1]Iso Omena, http://www.isoomena.fi/

[2]Aalto Design Factory, http://aaltodesignfactory.fi/

is considered as an environment complex enough to cover a wide variety of *Smart Space* application scenarios, but the requirements may not be fully transferable to all *Smart Space* contexts.

4.1.1 4D-Space Project

This Thesis has been made as part of the research conducted in the *4D-Space* [56] project under the *Multidisciplinary Institute of Digitalisation and Energy (MIDE)* research program at the Aalto University. The aim of the *4D-Space* project is to research and implement novel retail services in collaboration with both customers and retailers. Given the research goals of the project, the cloud-based platform developed within this Thesis focuses on providing facilities for retail-centric services, although it is designed to scale outside the retail context.

4.2 Research Questions

As demonstrated in Chapter 3, there is a wide variety of platforms providing services for integrating real-time communication, physical objects, positioning, and social media as part of a Web application. However, most of these platforms offer features related to only one topic (e.g., platform independent server push or social media integration) meaning that in order to utilize the functionality of several platforms, a developer needs an implementation for each platform separately and is dependable on several third-party services. To study and overcome this problem, the main research question of the Thesis has been formulated as follows:

Q1: How to integrate server push, *Smart Object* integration, positioning, and *Social Login* services under a single cloud-based platform?

In addition to the main research question, this Thesis aims to give an answer to other related questions as well. These secondary research questions are:

Q2: How to transform indoor environments to *Smart Spaces* by utilizing the existing plain and smart objects?

Q3: How to seamlessly integrate digital services as part of the everyday physical activities?

The main research question of the Thesis aims to solve a more general problem from the service and developer's point of view, whereas the secondary research questions focus more on the broad concepts (e.g., WoT and ubiquitous environments) of the Web 3.0 generation. In other words, the secondary research questions focus on bridging the gap between the current and the next generation of the Web by discovering ways to interconnect constrained objects with Web technologies and finding out ways for an environment to autonomously adjust as well as perform actions based on the changes and events in that environment. By giving tools for the developers to create social applications that integrate digital and physical world with Web technologies, potentially any environment could be transformed into a *Smart Space*.

4.3 Research Methods

The research strategy for this Thesis consists of five steps. First, a state-of-the-art review of research, services, technologies, and best practices regarding to the concepts of the current and the next generation of the Web (i.e., Web 2.0 and Web 3.0) is conducted. Second, a requirements analysis for the platform is made by analyzing the results from the state-of-the-art review as well as the results [57, 58] gained from the previous research in the 4D-Space project. Third, a proof-of-concept prototype platform is designed and implemented. Fourth, the usefulness of the platform is tested by creating two sample applications, which are tested with real-world users. Last, the performance of the platform is evaluated by running a series of automated test runs in the laboratory environment.

Chapter 5

Platform Requirements

In order to find all the requirements for the platform, the *Smart Space* context (the shopping center environment, cf. Chapter 4) was examined from three aspects as follows:

- Terminals and devices,
- Users, and
- Objects.

Each aspect is covered in detail in the following sections. Then, the requirements derived from analyzing the aspects are compared against the results obtained from empirical studies. Finally, the list of requirements for the platform is conducted from the analysis.

5.1 Terminals and Devices

Nowadays, information should be accessible through a wide variety of devices. Users possess private terminals, such as smartphones, tablets, and laptops; and both indoor and outdoor environments are full of info kiosks, large information displays, and shared computers. For the requirement analysis, the displays are divided into three categories:

1. Public displays,
2. Semi-public displays, and
3. Private displays.

In order to support all the different display types (cf. above and Figure 5.1), the platform should be accessible from all kinds of devices and should provide information based on the context of use as well as based on the privacy level of both the user (e.g., anonymous user or logged in user) and the display (e.g., smartphone or public information display).

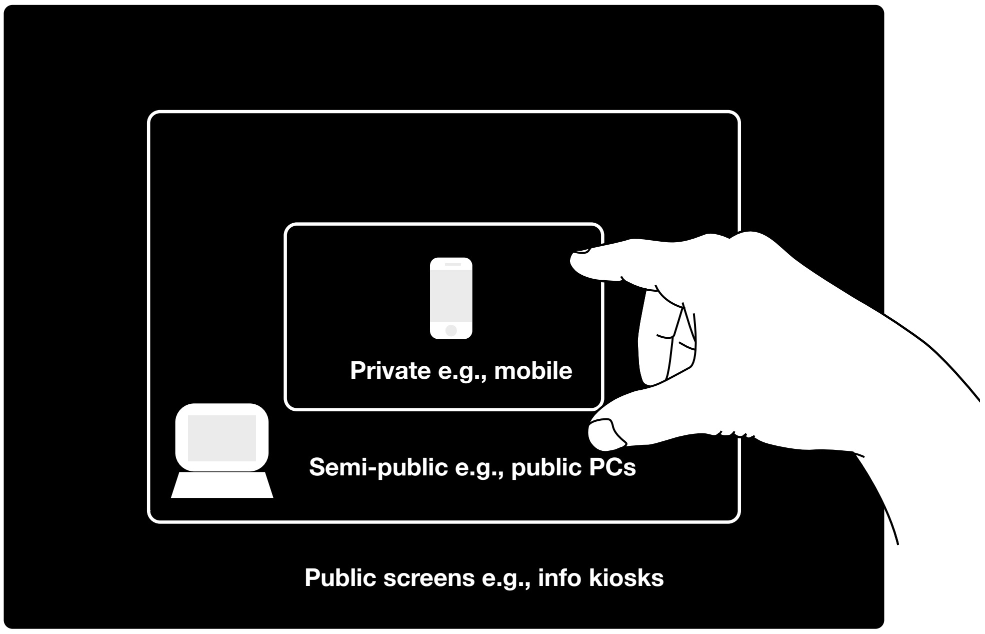

Private e.g., mobile
Semi-public e.g., public PCs
Public screens e.g., info kiosks

Figure 5.1: Different display types.

5.2 Users

Google's division of *Mobile Users* [59] was used as a basis for analyzing the users. Although, Google's analysis focuses on mobile phone users and not users in general, it fits well on the *Smart Space* context with a variety of different terminals and dynamic user groups. According to Wellman, Google[1] divides *Mobile Users* into three distinct categories that are:

[1]Google, http://www.google.com/about/company/

1. Repetitive Now,
2. Urgent Now, and
3. Bored Now.

The *Repetitive Now* user type wants repetitive information from well-defined sources regularly. For example, a user might check the latest news from dedicated mobile applications from time to time. To satisfy the needs of *Repetitive Now* users, the platform needs to provide access to information with pull-based methods (i.e., the data must be stored and be accessible later).

The *Urgent Now* user type wants relevant, personalized, and context-aware information in real-time. For example, a customer in a hurry inside a shopping center needs to know where is the nearest restaurant serving the food she likes, and where is the friend she is supposed to eat with. In order to fully serve the *Urgent Now* user type, the platform needs to be able to push information in real-time, know the preferences as well as social connections of a user (i.e., the social media profiles), and be able to locate the user both indoors and outdoors. Traditional positioning methods do not work or are not accurate enough indoors, so in order to provide indoor location information, the service also needs to support the integration of different sensors to track individuals and their movements.

The *Bored Now* user type does not need any specific information. This type of a mobile user has some spare time and wants to be entertained. For example, a user waiting for a bus has a couple of minutes and wants to play a game with her mobile phone. In the context of this Thesis, the *Bored Now* user type is considered the least important, because of the information-centric nature of *Smart Spaces*.

In addition to the users/visitors of any space, there are also other stakeholders, such as administrators of the space and employees (e.g., cleaners and janitors). For these type of users, it is important that the space can autonomously perform actions and signal conditions (e.g., open locks to restricted areas when a valid identity card is shown or calculate and show the number of free parking slots). To meet the needs of these users, the platform must support the integration of various types of sensors in addition to the sensors related to indoor positioning.

5.3 Objects

Different environments, especially shopping centers, are full of different objects with pre-existing identifiers, such as *Universal Product Codes (UPC)* on products and *International Standard Book Numbers (ISBN)* on books. To provide a digital representation of these objects, the platform must recognize objects based on different codes.

5.4 Verification of the Requirements

To verify the requirements with real shopping center users, three workshops in the Iso Omena shopping center were organized to collect ideas and opinions about the future of shopping centers [57, 58]. The workshops resulted to approximately 450 ideas. The majority of the ideas reflected the requirements presented above. For example, the participants of the workshops wanted to gain more information on products (e.g., carbon footprint) with their mobile phones. They also felt that advertisements should be personal as well as location-aware and the information should be aggregated to a single shopping center UI. In addition, the customers hoped that there would be a bi-directional channels between customers, retailers, and administrators to share ideas and information. Furthermore, the participants also wished that they could share information to other services, which is a common practice in modern Web applications. Based on the above, the platform should also support aggregating information from various information sources and sharing information to social media.

5.5 Requirements

Based on the workshops, state-of-the-art review, and requirements analysis, a list of requirements (cf. Table 5.1) for the platform was conducted. Each requirement is covered more thoroughly in the following sub-sections.

ID	Name	Description
R1	RESTful API	The platform must provide access to data in pull-based methods as well as provide representations and manipulation of physical objects through a uniform interface.
R2	Server push	The platform must be able to push data to clients in real-time regardless of the client device.
R3	User profiles	The platform must be able to identify users based on their existing profiles and other identifiers as well as to provide information on users preferences and social connections.
R4	Positioning	The platform must be able to provide and process location information both outdoors and indoors.
R5	Object recognition	The platform must be able to identify objects based on different identifiers, such as barcodes or *Near Field Communication (NFC)* enabled tags.
R6	Sensor integration	It must be possible to integrate different sensors to the platform in order to facilitate *Smart Spaces* and indoor positioning.
R7	Information aggregation	The platform must be able to aggregate information from connected applications, information services, and social media.
R8	Social sharing	The platform must provide means for sharing information to social media.

Table 5.1: Requirements for the platform.

5.5.1 R1: RESTful API

As stated earlier, some users (i.e., the *Repetitive Now* user type) wish to access specific type of information on-demand, at a time most suitable to them. These users know what they want and when they want it. To serve these users, the platform should provide access to historical data through an API. In addition, the physical objects need to be represented and accessed in the digital context as well. To support these requirements, the platform should provide an API-level access for the digital counterparts of physical objects.

5.5.2 R2: Server Push

In comparison to the *Repetitive Now* type users, the *Urgent Now* type users want context-aware information to be pushed to them at all times. These users want their preferred information to be pushed to them even if they are not actively concentrated on the topic of information at that particular moment. In order to satisfy these users, the platform needs to be able to push information in real-time, regardless of the context, end device, or location. Furthermore, in the *Smart Space* context, some actions are autonomously performed based on events triggered by *Smart Objects*. In order to function seamlessly, the events have to be transmitted in real-time.

5.5.3 R3: User Profiles

To fulfill the requirements R1 and R2, the information provided should vary per user basis. In order to provide personalized information, the platform must provide user information (i.e., user profiles and social connections). In addition, it is crucial for *Smart Space* environments to be able to separate a user from another in order to adjust the space and perform actions based on a particular user. To accomplish the above requirements, the platform must provide the ability to integrate existing social media as well as custom profiles to users (e.g., an NFC identity for performing actions within a *Smart Space*).

5.5.4 R4: Positioning

For *Smart Spaces* and location-based applications, it is vital to be able to locate objects and people accurately. Positioning in a shopping center environment is especially important because it is full of both indoor and outdoor areas providing different kinds of services, some more relevant than others, to a particular person. From a customer's point of view, it is important that both the customer and the services she is interested in are accurately positioned. From an administrator's point of view, it is important to be able to track movements inside a space in order to spot which areas are more crowded than others and to utilize that data in further planning. To satisfy these requirements, the platform must provide accurate location information of individual users both indoors and outdoors as well as the flow of people inside the space.

5.5.5 R5: Object Recognition

In a *Smart Space* environment, objects must contain some kind of an identifier so that these objects can be mapped to a correct digital representation. The importance of identifying an object varies based on the characteristics of it. Even with a plain object not capable of performing any autonomous actions, it is valuable to have a digital representation. The representation can be used to store information that cannot be embedded to the physical representation, such as the carbon footprint of a product or the amount of loans for a particular library book. In order to identify also plain objects, the platform must be able to recognize objects based on different codes.

5.5.6 R6: Sensor Integration

As discussed earlier, positioning in indoor environments is not an easy task since the most used positioning techniques (e.g., GPS) do not work indoors or do not provide accurate enough information. In order to provide usable location information in *Smart Spaces*, a room level accuracy should be considered to be the minimum requirement. To be able to provide such an accurate indoor location information, different kinds of sensing devices are needed in different spaces.

In addition to indoor positioning, sensors can provide valuable information for *Smart Space* administrators. For example, anonymous people flow tracking provides information about the most popular or crowded areas of the space, sensors can keep track of the available rooms or parking slots in the area, or an electricity consumption as well as the state of individual electronic appliances can be easily monitored and controlled. To satisfy all kinds of sensor-based activities, the platform must support the integration of different kinds of sensors.

5.5.7 R7: Information Aggregation

In a *Smart Space*, there is usually many information sources providing different information valuable for the users in that space. For example, in a shopping center a particular visitor might want to know where her friend is (personal information), a shop could advertise clothing for women (information targeted to a group), and a shopping center might want to inform all customers that the center is going to be closed in fifteen minutes (common information). The more information sources there are, the harder it is for the users to gain access to all the important information. To be able to deliver all the relevant information, the platform must be able to aggregate information from various information sources.

5.5.8 R8: Social Sharing

The Web 2.0 era and the emergence of social media have made it possible for users to become content providers on the Web. The users have become accustomed to be able to share information from almost any kind of Web application to social networks. To be able to facilitate these actions, the platform must provide means for sharing information to social networks, such as Facebook.

Chapter 6

Implementation

In this Chapter, a proof-of-concept implementation, the UbiQloud platform, is described. First, a brief overview of the platform is presented. Next, the server-side architecture of the platform is introduced. Then, the internal architecture of the UbiQloud application is described in detail, and last, the communication methods and protocols used to interact with the platform are covered.

6.1 UbiQloud Overview

UbiQloud is a cloud-based PaaS facilitating rapid development of social, location- and context-aware, real-time applications for smart indoor spaces. The platform provides a Web-based UI for developers, in which they can register applications and gain necessary resources to interact with UbiQloud. Figure 6.1 represents the overall communication architecture of UbiQloud. The platform can be divided into three kinds of interfaces that are (1) client APIs, (2) a *Smart Gateway*, and (3) a *Social Gateway*. Clients (i.e., the applications developed on top of UbiQloud) can communicate with the pull-based APIs using HTTP protocol and with the push-based APIs using XMPP (mobile applications) or XMPP over WebSocket (Web applications). The *Smart Gateway* is responsible for communicating with integrated sensors. The UbiQloud/sensor communication is established over HTTP or TCP Socket. The *Social Gateway* consist of a set of APIs and a widget that can be used to integrate multi-service *Social Login* to client applications.

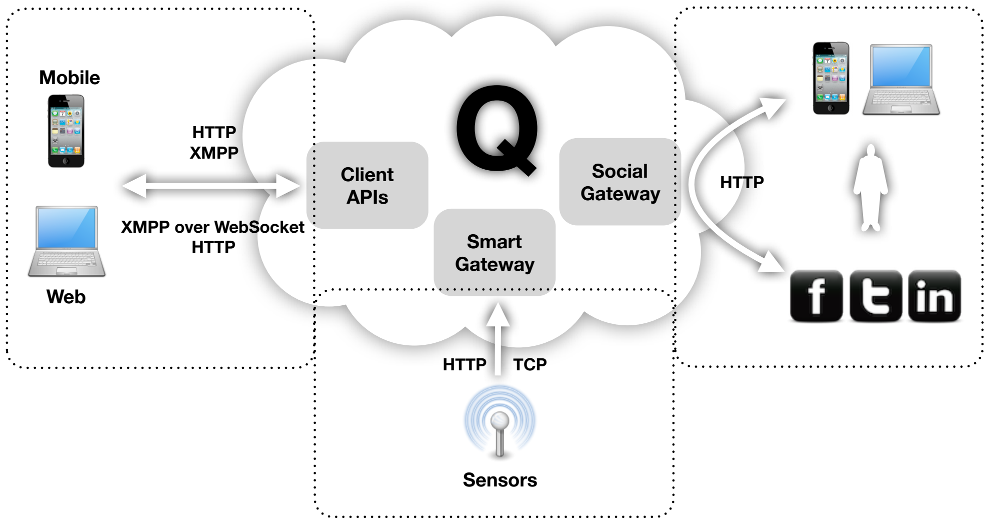


Figure 6.1: UbiQloud communication architecture.

From a service point of view, UbiQloud provides mechanisms for developers to:

- Integrate sensors,
- Manage users,
- Access user data on social media services,
- Use and extend a shared database of objects, and
- Store and access data with both pull- and push-based methods.

UbiQloud does not restrict the type of sensor to be integrated. However, for parsing and manipulating the sensor data on the cloud before pushing it forward, a dedicated driver should be implemented. At the moment, UbiQloud only contains drivers for two types of sensors that are (1) a custom Wireless Sensor Network (WSN), RealSense, for monitoring people flow and (2) Feig LRU Ultra-High Frequency (UHF) reader. The data coming from sensors lacking a driver is pushed as is. Each sensor has its own dedicated channel where to the data is pushed in real-time. Sensors can communicate with UbiQloud either through an HTTP interface or a raw TCP socket.

Each UbiQloud user contains an ID and XMPP credentials, which are shared with all UbiQloud applications. In addition, a user object can possess any number of application-specific profiles (e.g., a Facebook or NFC profile). The profiles are application specific in order to protect users' privacy (i.e., a user must authorize an application to use a particular profile). For the authorization of social media profiles, UbiQloud provides a JavaScript widget that can be embedded to an application.

Similar to the users, UbiQloud also hosts a database for objects. Each object has public properties, such as an identifier (e.g., UPC or ISBN), a set of tags for textual representation of the object, and a set of images. In addition to the public properties, each object has a set of application-specific activities (e.g., likes, comments, and ratings) to be used in an application-specific context. Furthermore, each activity is related to one application and one user.

Both users and objects in UbiQloud have also application-specific location feeds. An application can send coordinates (a latitude/longitude pair) to UbiQloud, which in turn converts the coordinates to a textual representation.

Data in UbiQloud is accessible with both pull- and push-based methods. In most cases, the data is processed in similar way. First, UbiQloud receives a request (e.g., a sensor sends new data or a user comments an object through an application UI) that affects on the data. Second, the data is stored in a database for later access through pull-based APIs, and third, the new data is pushed to connected clients through a dedicated channel. In other words, UbiQloud supports a common scenario, in which the history data is fetched (pull) from the server at application startup, and as long as the application is running, the data is kept up-to-date in real-time (push).

The platform aims to fulfill developers' needs in a variety of integration levels. UbiQloud can be used as the sole back-end solution for an application or a single module can be integrated to a custom solution. The main target group for UbiQloud is freelance developers or small businesses in need for a platform to rely on modern, Web 3.0 applications.

6.2 Server Components

UbiQloud relies only on free, widely used, and easily configurable third-party components. The idea is that setting up UbiQloud instances would be as easy as possible regardless of the environment (e.g., Windows, OS X, or Linux). Figure 6.2 represents the server-side architecture of the platform. The architecture consists of three third-party components that are:

1. Play! Framework (version 1.2.4),
2. Openfire (version 3.7.1) XMPP server, and
3. MySQL (version 5.1.50) *Relational Database Management System (RDBMS)*.

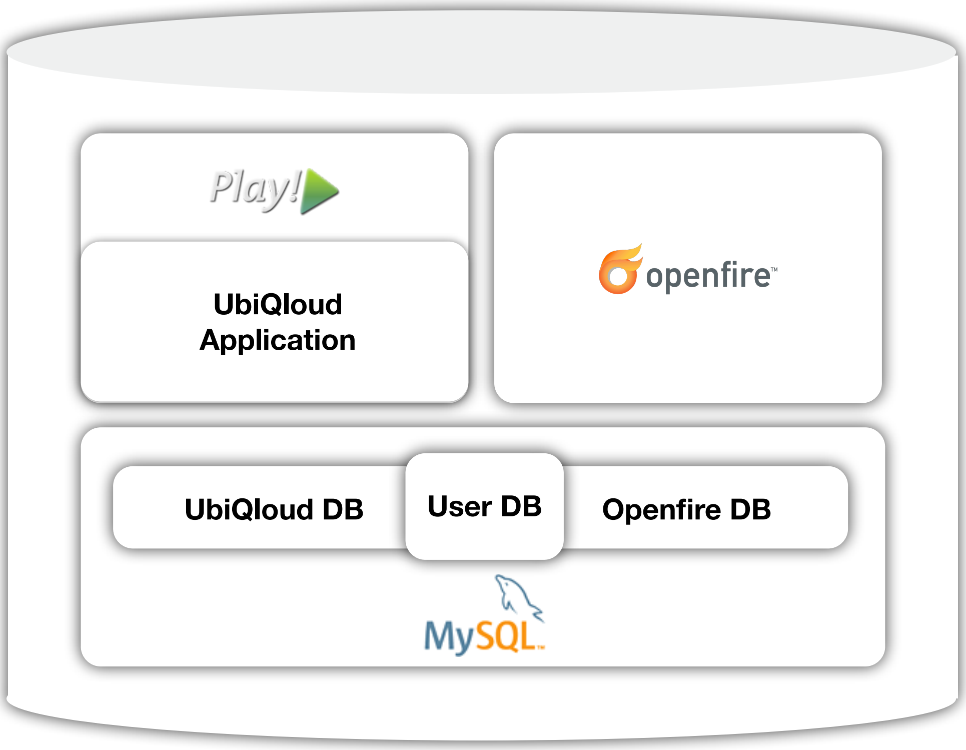


Figure 6.2: Components of the UbiQloud platform.

Play! Framework [60] is a modern Java Web framework targeted to RESTful architectures. The framework provides built-in support for modern Web application requirements, such as controllers for real-time communication (i.e., Comet and WebSocket), and clients for communicating with secure third-party Web services (e.g., an API using OAuth). Play! is based on a stateless architecture, which makes it easy to scale Play!-based applications (i.e., multiple instances of the same application can be run simultaneously).

Openfire [61] is a Java-based XMPP server for real-time communication. It supports a wide variety of XEPs (e.g., publish/subscribe) and can be integrated with existing user management systems. Furthermore, Openfire can be easily extended with plugins and custom classes implementing its interfaces. For example, UbiQloud relies on a plugin for providing the XMPP over Websocket gateway and a custom class for authenticating users per application basis.

MySQL [62] is the world's most popular open source RDBMS. It runs on a server and provides access to a number of databases. MySQL can be used on a variety of different environments including Windows, OS X, and the most popular Linux variants. UbiQloud does not rely on any MySQL-specific features, meaning that it could be easily used with other RDBMS as well (e.g., PostgreSQL[1]). In addition to the popularity of MySQL, it was chosen as the RDBMS for UbiQloud, because both Play! and Openfire have a built-in support for it.

6.3 UbiQloud Application

Based on the requirements introduced in Chapter 5, the UbiQloud application architecture is divided into four modules and seven sub-modules. The primary goal for the module-based approach is to ease the development of the platform by dividing the code base into smaller pieces that could be developed separately. In reality, the modules are not fully undependable of each other and the functionality is not restricted to a certain topic (e.g., requirement R6 is linked to the *Indoors* module although it covers all sensor integration). Table 6.1 represents the different modules and Figure 6.3 represents the relations between the modules and the requirements. The modules are covered in detail in the following sub-sections. Requirement R1 can be considered module independent because almost everything in UbiQloud can be accessed through the RESTful API, and therefore the requirement R1 covers all modules.

[1]PostgreSQL, http://www.postgresql.org/

ID	Name
M1	Recognize
m1.1	User Profiles
m1.2	Objects
M2	Locate
m2.1	Indoors
m2.2	Outdoors
M3	Connect
m3.1	Push
m3.2	Aggregate
M4	Share
m4.1	Social Media

Table 6.1: UbiQloud modules.

6.3.1 Recognize Module

In order to fulfill the requirements R3 and R5, the *Recognize (M1)* module was implemented. The module consists of two sub-modules that are *User Profiles (m1.1)* and *Objects (m1.2)*. Every UbiQloud user (i.e., any user of any application developed on top of UbiQloud) is represented as an object with various profiles and a location feed. All the users have an unique XMPP profile, which is automatically created during the first login and is completely transparent to the user. In addition, users have a set of application-specific profiles ranging from different social media profiles to digital identifiers, such as NFC tags and Bluetooth devices.

Objects in UbiQloud behave similar to users. They are also identified and they have a location feed. However, objects do not have profiles like users but a single unique identifier. In addition, all objects have application-specific activity feeds consisting of different activities made by users. A single activity can be, for example, a `like`, a `comment`, or a `check-in`. UbiQloud allows using any of the provided activities with any kind of object.

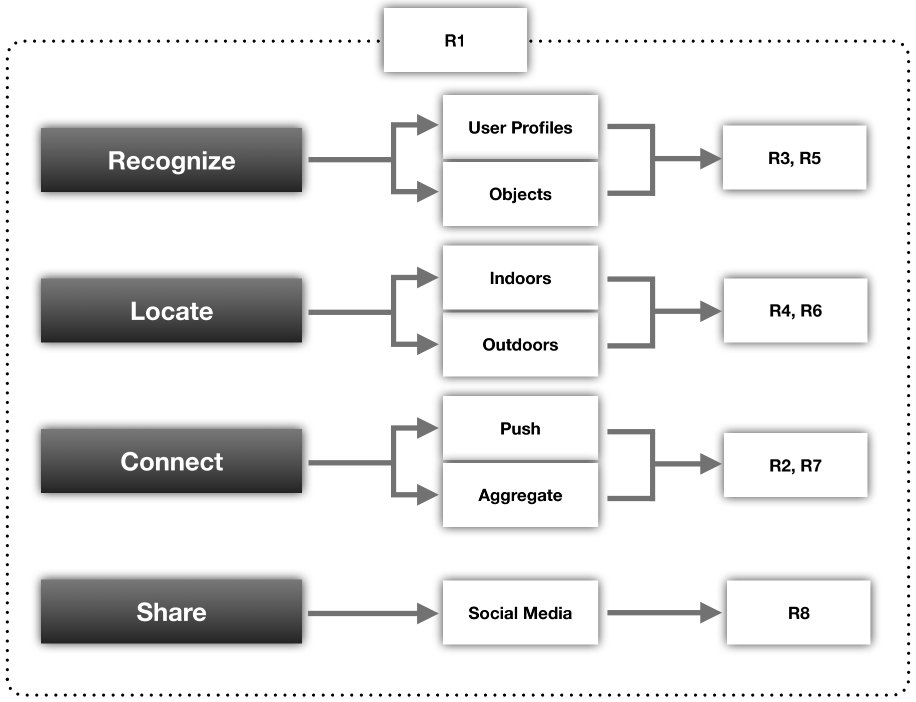


Figure 6.3: Relations between the UbiQloud modules and requirements.

6.3.2 Locate Module

The *Locate (M2)* module represents the requirements R4 and R6 and consists of the *Indoors (m2.1)* and *Outdoors (m2.2)* sub-modules. As the sub-module topics suggest, the aim of the modules is to provide positioning related functionality. Although positioning is the core context, the *Indoors* sub-module is responsible for all sensor integration with the UbiQloud platform (i.e., the sensor interfaces and drivers). The *Indoors* sub-module can be used with any sensor, but in order to gain parsed and well-formed data, a dedicated driver for each sensor should be implemented. At the moment, the drivers implemented (RealSense and Feig LRU) are meant for indoor positioning.

The *Outdoors* sub-module is responsible for providing meaningful outdoor location data for other modules and applications. The main task of the sub-module is to receive coordinate points and transform those coordinates to textual representation by utilizing the *MapQuest Nominatim Search API Web Service*[2].

[2]MapQuest Nominatim Search API Web Service, http://developer.mapquest.com/web/products/open/nominatim

6.3.3 Connect Module

For the requirements R2 and R7 the *Connect (M3)* module was implemented. The module consists of two communication related sub-modules, that are *Push (m3.1)* and *Aggregate (m3.2)*. The *Push* sub-module is responsible for all the real-time communication functionality in UbiQloud. When a data manipulation request is received by UbiQloud, the *Push* sub-module performs three steps as follows:

1. Intersects the request,
2. Converts the data to XML if possible/needed (the data is already parsed, manipulated, converted, and stored by other modules), and
3. Pushes the data to a dedicated channel (e.g., sensor feed or application feed).

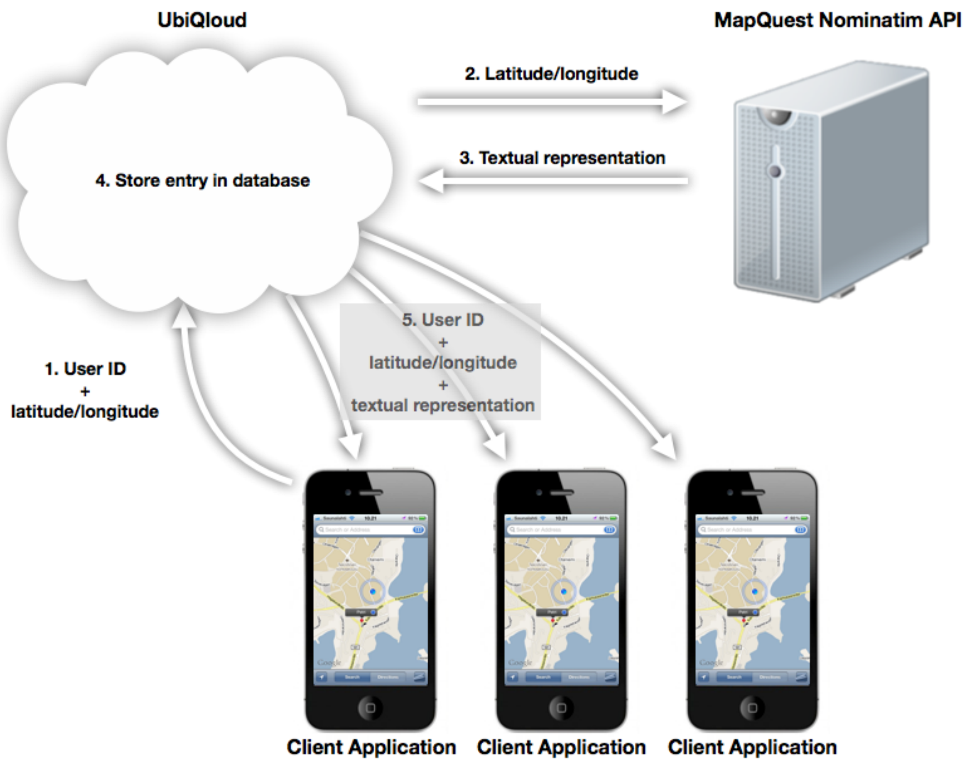

UbiQloud
MapQuest Nominatim API
2. Latitude/longitude
3. Textual representation
4. Store entry in database
1. User ID + latitude/longitude
5. User ID + latitude/longitude + textual representation
Client Application
Client Application
Client Application

Figure 6.4: Example of using the UbiQloud APIs for pushing the location information of a user.

The *Aggregate (m3.2)* sub-module is responsible for aggregating data from different sources. For example, an application might request user's connections and if the user has authorized the application for her Facebook and Twitter profile, the *Aggregate* sub-module combines the data from both services. Another example would be data aggregation from different modules as illustrated in Figure 6.4. First, a client application sends a user's location (i.e., user ID and coordinates) to UbiQloud. Second, the coordinates are converted to textual representation by using the *MapQuest Nominatim Search API.* Third, the data (i.e., user ID, coordinates, and textual representation) is aggregated to a single location point object and stored into database. Fourth, the data is pushed to the application channel.

6.3.4 Share Module

The *Share (M4)* module aims to fulfill the Social Sharing (R8) requirement, by creating a unified API for the third-party publishing APIs of the supported social media services. The sub-module *Social Media (m4.1)* is responsible for communicating with the social media services and thus abstracting the differences of the third-party APIs by providing a single API method for sharing data to different services.

6.4 Communication with UbiQloud

UbiQloud was designed to support different protocols when establishing communication channels with various applications and devices (cf. Figure 6.1). Applications implemented on top of UbiQloud have access to real-time data through XMPP (native mobile or desktop application) or XMPP over WebSocket (Web applications). All the data is also stored in the platform and can be accessed through the RESTful API in either JSON or XML format. Devices connected to UbiQloud can use either an HTTP interface or a pure TCP Socket interface. The design of the platform also supports adding device-specific drivers for handling proprietary protocols. For allowing a user to login with her existing social media credentials, UbiQloud offers a *Social Gateway* in form of a JavaScript widget. The JavaScript widget acts as a bridge between social media services and applications developed on top of UbiQloud.

6.4.1 RESTful API

For accessing the data stored in UbiQloud retrospectively, the platform provides an HTTP-based RESTful API. The REST-based architecture is well suited for UbiQloud because UbiQloud contains resources, which are linked to physical objects (i.e., smart objects). The API is divided into four parts representing users, objects, sensors, and applications. The resources are accessed with HTTP methods (*GET*, *POST*, *PUT*, and *DELETE*) and identified with the following URL patterns:

URL
`http://api.ubiqloud.io/{user|object|sensor|app}/...`

For example the following request:

HTTP GET
`http://api.ubiqloud.io/object/1/activity/1`

could result the following response in JSON:

```
{
"activity" :
  {
  "id" : 1,
  "type" : "like",
  "published" : "2011-02-07 11:18:32.0",
  "userid" : "foobar"
  }
}
```

Listing 6.1: Example of the JSON representation of a resource

or in XML:

```
<activity>
  <id>1</id>
  <type>like</type>
  <published>2011-02-07 11:18:32.0</published>
  <userid>foobar</userid>
</activity>
```

Listing 6.2: Example of the XML representation of a resource

To make sure that the data is accessible only for authorized parties, several methods have been implemented to secure the API and to identify the client application making a request (cf. Figure 6.5). The API security procedures are inspired by the *Signature version 2* [63] of the *Amazon Web Services (AWS)*.

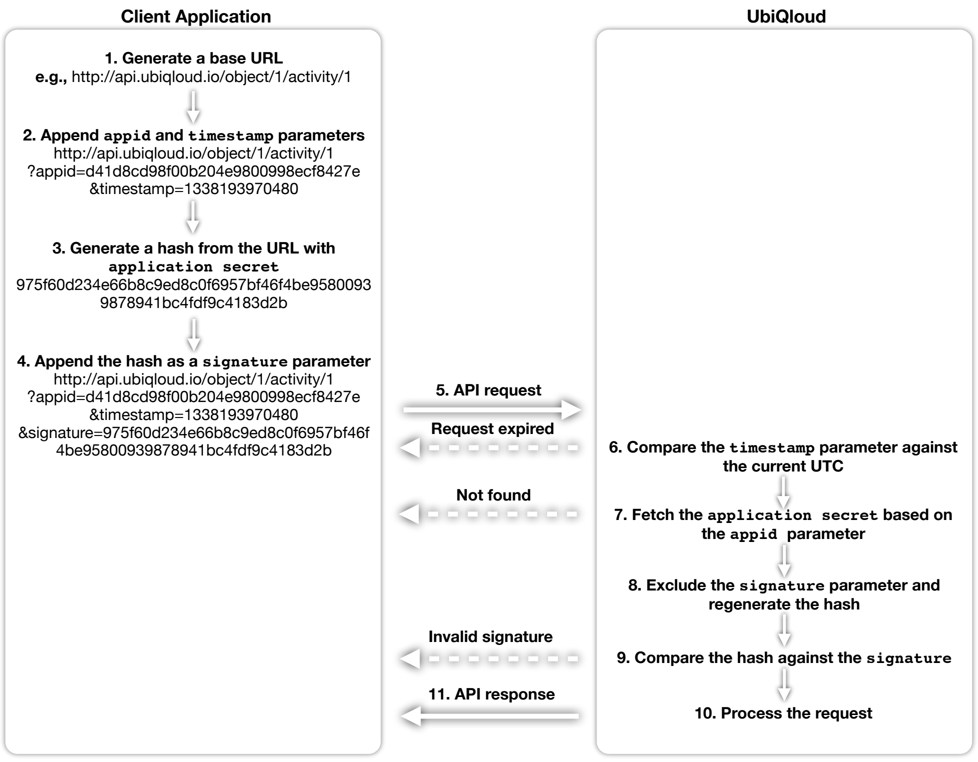

Client Application
UbiQloud
1. Generate a base URL
e.g., http://api.ubiqloud.io/object/1/activity/1
2. Append appid and timestamp parameters
http://api.ubiqloud.io/object/1/activity/1
?appid=d41d8cd98f00b204e9800998ecf8427e
&timestamp=1338193970480
3. Generate a hash from the URL with application secret
975f60d234e66b8c9ed8c0f6957bf46f4be9580093
9878941bc4fdf9c4183d2b
4. Append the hash as a signature parameter
http://api.ubiqloud.io/object/1/activity/1
?appid=d41d8cd98f00b204e9800998ecf8427e
&timestamp=1338193970480
&signature=975f60d234e66b8c9ed8c0f6957bf46f
4be95800939878941bc4fdf9c4183d2b
5. API request
Request expired
6. Compare the timestamp parameter against the current UTC
Not found
7. Fetch the application secret based on the appid parameter
8. Exclude the signature parameter and regenerate the hash
Invalid signature
9. Compare the hash against the signature
11. API response
10. Process the request

Figure 6.5: Secure communication with the UbiQloud RESTful API.

Once a developer registers an application to UbiQloud, unique ID and secret — that is public and private keys — are generated for the application. For every request the application makes against the API, three additional parameters should be included into the URL. The first parameter is `appid` with the value of the ID assigned to the application. The second parameter is `timestamp` with the value of the current *Coordinated Universal Time (UTC)* time in milliseconds. The third parameter is `signature` with a base64 encoded *Hash-based Message Authentication Code (HMAC)* generated from the request string and the application secret with the SHA-256 cryptographic hash function. For example, the request presented above could look as follows:

HTTP GET

```
http://api.ubiqloud.io/object/1/activity/1
?appid=d41d8cd98f00b204e9800998ecf8427e
&timestamp=1338193970480
&signature=975f60d234e66b8c9ed8c0f6957bf46f4be95800939878941bc4
fdf9c4183d2b
```

When UbiQloud receives the request, it first compares the `timestamp` parameter against the current UTC time and if the value of the `timestamp` is more than five minutes in the past the request is rejected. The timestamp comparison is made to prevent replay attacks (a hostile party copies a request and tries to use the same request in the future to make authenticated API calls). If the `timestamp` is valid, the `appid` parameter is used to fetch the correct application secret from the database. Then, the `signature` parameter is omitted from the request string and the same hashing operation made in the application side is made in UbiQloud as well. If the resulted hash value equals to the value of the `signature` parameter, UbiQloud can be sure that the request has come from the application and the request is delegated to the controller processing the actual operation requested.

6.4.2 Server Push Interfaces

Server push in UbiQloud is based on the publish/subscribe architecture and uses XMPP and related XEPs (namely, XEP-0060: Publish-Subscribe) as the underlying protocols for real-time communication. In UbiQloud, the publisher is always the platform itself, meaning that all the push events are emitted from operations in the RESTful API. There is two reasons for not allowing a client application to use real-time channels directly. First of all, the data needs to be stored and possibly manipulated for later use in a correct format (i.e., XML or JSON). Second of all, by restricting the way to publish data, an application cannot accidentally or deliberately pollute wrong channels.

The publish/subscribe model fits well for the service-oriented nature of UbiQloud. The platform consists of related, yet loosely-coupled, services that need to provide data both to each other and to a dynamic group of data consumers (i.e., applications developed on top of the platform and *Smart Objects* integrated to it). In other words, the platform includes services, users, objects, sensors, and applications that do not need to be aware of each other, but need to be able to provide data to each other. To facilitate this kind of a loosely-coupled system, each entity capable of creating or manipulating data has a unique channel to push changes. For example, a sensor communicating with UbiQloud could provide valuable data to several applications, but it does not need to know the identity of these applications (the amount and type of consumer applications may vary over time). Furthermore, some of the applications might need data from other sensors as well, but the sensors do not need to be aware of each other (i.e., a centralized data broker for all the sensors would not be an optimal solution). The solution is to provide a unique channel (node) for every sensor to push data and let interested parties to subscribe to that channel.

XMPP was chosen as the real-time communication protocol for several reasons. Firstly, It is a mature, well supported, and standardized protocol. Numerous client libraries exists for several platforms and languages, meaning that XMPP would not become a bottleneck for developing UbiQloud applications. Secondly, XMPP is extensible and there is already a vast range of extensions covering features needed in UbiQloud (e.g., publish/subscribe). Furthermore, because of the extensibility, it is possible to implement custom, platform-specific extensions if needed. Thirdly, XMPP has a built-in support for user management, authentication, and presence, which is important for the UbiQloud platform (i.e., regardless of the social media profiles, users need to be able to authenticate to the platform in a common way). Lastly, XMPP is reliable and secure [12], which is important for UbiQloud, because (1) it consist of interconnected entities that rely on each other, and (2) it mediates sensitive and private data (e.g., user and application-specific data).

As stated in Chapter 2 (Section 2.1.2), although XMPP is a widely used protocol for real-time communication, it is hardly used in traditional Web applications, because none of the major Web browsers support it natively. Luckily, XMPP can also be used [16, 21] over protocols suitable for Web browser environments. To support also browser-based applications, UbiQloud provides a WebSocket gateway for XMPP communication.

Every user in UbiQloud has a shared XMPP profile (i.e., unlike social media profiles, the XMPP profile is not application specific). Even if the XMPP profile of a user is available to all UbiQloud applications, UbiQloud requires that an application making the connection (e.g., user authentication) on behalf of a user is a registered application. In general, when a user wishes to authenticate to an XMPP server, the identity (i.e., *Jabber ID, JID*) and password of the user is sent to server. In UbiQloud, the JID is sent as is, but instead of the password the application must send a string in format {appid}:{hash}, where hash is a base64 encoded HMAC constructed from the user's XMPP password and application's secret with the SHA-256 cryptographic hash function.

Once authenticated, the user is able to subscribe to any number of channels to start receiving real-time updates through the push interfaces. For example, if subscribed to the application's channel in which another user adds an activity to an object, the following XMPP message could be received:

```
<message from="pubsub.ubiqloud.io" to="examplejid@ubiqloud.io
   " id="foo">
  <event xmlns="http://jabber.org/protocol/pubsub#event">
    <items node="{unique node identifier}">
      <item id="ae890ac52d0df67ed7cfdf51b644e901">
        <root>
          <object id="1">
            <activity id="1" method="post">
              <type>like</type>
              <published>2011-02-07 11:18:32.0</published>
              <userid>foobar</userid>
            </activity>
          </object>
        </root>
      </item>
    </items>
  </event>
</message>
```

Listing 6.3: Example of an XMPP publish/subscribe message.

6.4.3 Smart Gateway

When a developer integrates a sensor to UbiQloud, the type and *Internet Protocol (IP)* address of the sensor must be specified. The type can be one of the known sensor types (i.e., RealSense or Feig LRU) or undefined. In the case of the known sensor types, the received data is manipulated to more usable form (i.e., XML) before storing and pushing forward. In the case of an undefined type, the data is stored and pushed as is. The IP address is used to identify the sending server when new data is received in UbiQloud. If the IP address is known, the data can be treated correctly (i.e., where to store and push), and if the address is unknown the data is ignored. Each sensor has a unique channel, where the data is pushed when ready.

For communicating with UbiQloud, there are two kinds of interfaces available for sensors. One is a HTTP interface and the other is a TCP Socket interface. For using the HTTP interface, a sensor must send either a *GET* or *POST* request to a dedicated URL (`http://api.ubiqloud.io/sensor`) with the data as a value of the `data` parameter. The other option, the TCP Socket interface, can be used by initializing a socket connection to UbiQloud

(`ubiqloud.io`) through the port 9999. Once the connection is established, new data can be sent anytime.

6.4.4 Social Gateway

In general, to integrate a *Social Login* provided by any social media service, a developer needs to complete certain steps. First, an application needs to be created in the platform. During the application creation, a developer needs to provide some basic information (e.g., name of the application and URL used to communicate with the API) and gain application specific keys to securely communicate with the social media service. Second, a JavaScript widget needs to be embedded to the client application and initialized with application-specific options (e.g., public key). Third, a controller that communicates with the social media services during and after the authorization process needs to be implemented. From a developer's point of view, the problem is that social media services use different techniques for the authorization (e.g., different versions of OAuth or OpenID/OAuth hybrids) and differently implemented APIs. In other words, to support more than one *Social Login* providers, a developer needs a unique implementation for each service.

To ease the *Social Login* integration, UbiQloud provides a unified mechanism to integrate multiple *Social Login* providers. The mechanism works similarly as described above. First of all, a developer still needs to register applications to the social media services individually, because the users need to give an authorization per application basis. Once the applications are created, the developer adds the key pairs to UbiQloud which allows UbiQloud to communicate with the social media service on behalf of the client application. After the keys are added to UbiQloud, the developer can start using the *Social Login* service as illustrated in Figure 6.6.

First, the developer embeds the *Social Login* widget provided by UbiQloud to the client application as well as initializes it with an application specific URL and the target URL to post a one-time token (cf. Listing 6.4). The widget provides an UI to login to all the supported services (i.e., Facebook, Twitter, and LinkedIn). Once the user is logged in to a selected social media service and has authorized the application, UbiQloud receives the user information and user-specific tokens.

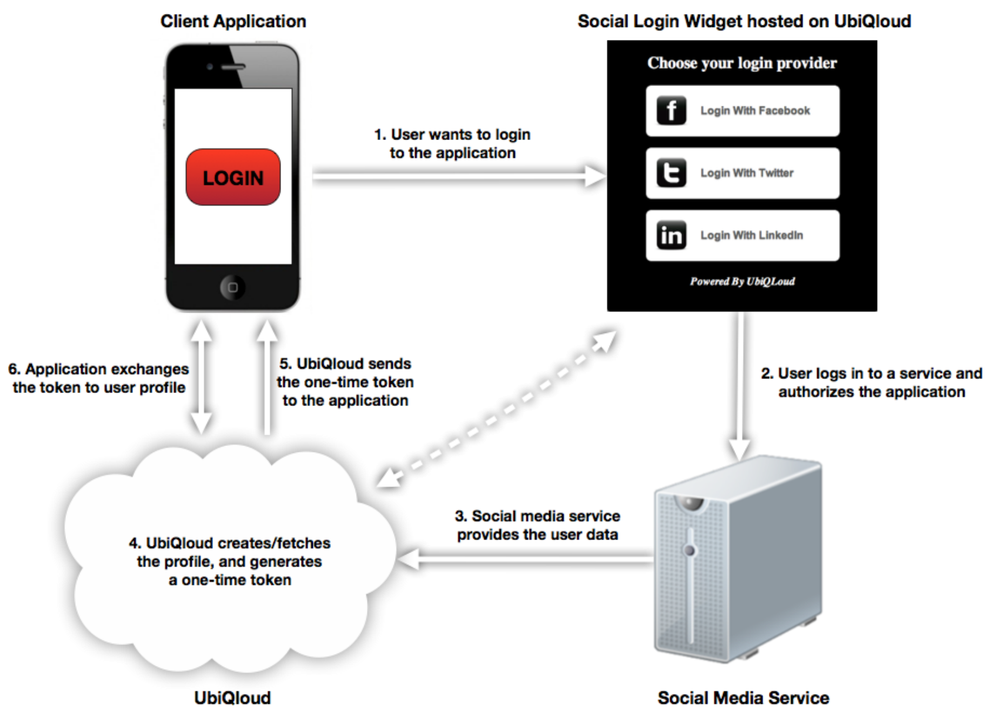

Client Application
Social Login Widget hosted on UbiQloud
LOGIN
Choose your login provider
Login With Facebook
Login With Twitter
Login With LinkedIn
Powered By UbiQLoud
1. User wants to login to the application
2. User logs in to a service and authorizes the application
3. Social media service provides the user data
4. UbiQloud creates/fetches the profile, and generates a one-time token
5. UbiQloud sends the one-time token to the application
6. Application exchanges the token to user profile
UbiQloud
Social Media Service

Figure 6.6: Six steps for *Social Login* through UbiQloud.

Then, a one-time token is generated in UbiQloud and sent to the client application via the widget. After receiving the token, the client application can exchange it to the user profile information. The authorization part needs to be completed only once per application and once authorized, the application can interact with the social media service on behalf of the user through UbiQloud's unified RESTful APIs.

```
<input id="UQ_login_button" type="button" value="{login
   button label}"/>
<script type="text/javascript" src="http://ubiqloud.io/public
   /javascripts/widgets.js"></script>
<script type="text/javascript">
  UQ.init("{appid}.ubiqloud.io", "{token URL}");
</script>
```

Listing 6.4: Example of embedding the UbiQloud *Social Login* widget.

Chapter 7

Sample Applications

This chapter demonstrates two sample applications, FeedThroat and InView, developed on top of the UbiQloud platform. The sample applications were developed for testing two aspects of the platform as follows:

1. Does the implemented platform meet the requirements set, and
2. Does the implemented platform work with real applications in real-world settings.

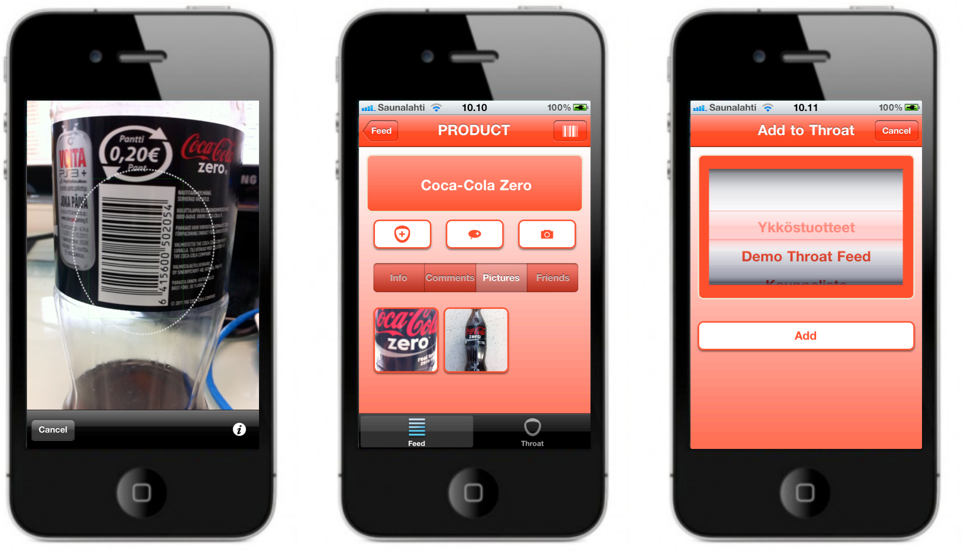
The first sample application, FeedThroat, was developed in conjunction with the first version of the UbiQloud core (i.e., FeedThroat was not initially a separate application, but part of the UbiQloud code base). The reason for the tight integration was to be able to easily test different technologies and tweak the APIs. The second sample application, InView, was developed as a separate application to actually test UbiQloud as a PaaS. A detailed overview of the sample applications is given in the following sections.

7.1 Retail Context: FeedThroat

FeedThroat is a social shopping assistant application developed for iOS[1] and the Web. The core idea of the application is to provide retailer independent product information with social sharing and collaboration functionality. The iOS version of the application is targeted for regular users (i.e.,

[1]iOS is the operating system used in iPhone, iPod Touch, and iPad

customers) and it contains all end-user specific functionalities. The Web version of the FeedThroat also contains most of the features (excluding camera-related functionality) for end-users. However, the main target group of the Web version is retailers, and therefore it contains retailer features not present in the iOS application.

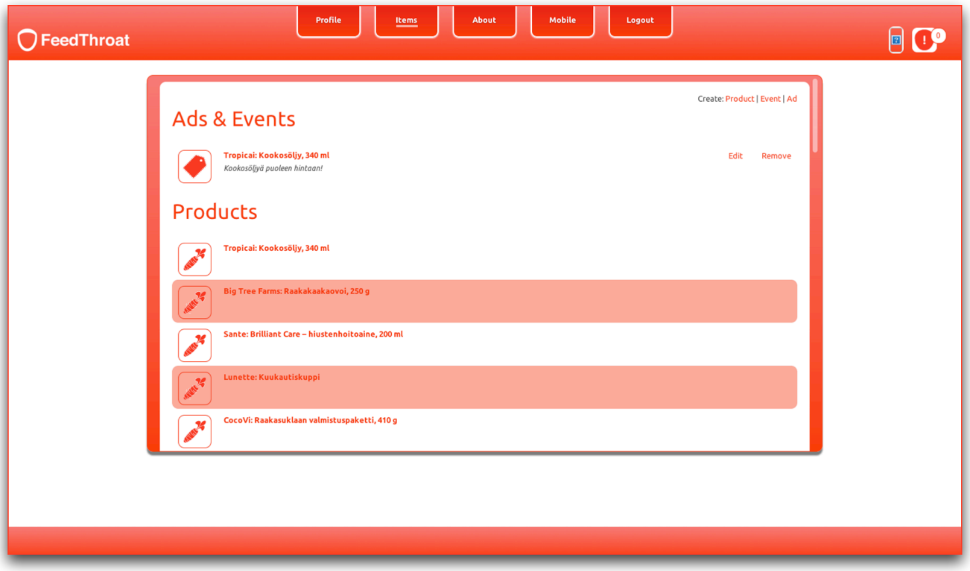


(a) Scanning a product (b) Viewing product information (c) Adding a product to a list

Figure 7.1: FeedThroat UI views in iOS application.

Figure 7.1 represents the main functionalities of the iOS application. First, the user can use the smartphone's built-in camera to scan the barcode of a product (a) to gain instant information of it (b). In addition to the traditional product information, there is a social layer on top of it including photos, comments, and a list of friends interested in the same product. Second, the application allows users to generate shared shopping lists with real-time updates and in-list product information (e.g., private comments and photos). Users are able to drop products to a list (c) by scanning a barcode, manually entering a product, or choosing from a dynamic product list of the newest products. Products in shared lists can also be checked as purchased (i.e., similar to tasks in todo lists that can be marked as done). The ability to check products is useful for example in shared shopping lists.

As mentioned, the Web version of FeedThroat shares most of the functionality with the iOS version. In addition, the Web application also provides a custom view for retailers, in which they can add products, events, and ads that are presented to customers in both applications. These items can also be dropped into a list the same way as products. Furthermore, retailers can automatically share the added items to Facebook, which means that while both FeedThroat and Facebook can be used as an advertisement channel, only one application is needed for generating the ads. In Figure 7.2, a retailer view containing the retailer's items is presented. In the view, a retailer can add new items as well as view, edit, and remove the already added items.



Ads & Events

Tropicai: Kookosöljy, 340 ml
Kookosöljyä puoleen hintaan!

Products

Tropicai: Kookosöljy, 340 ml

Big Tree Farms: Raakakaakaovoi, 250 g

Sante: Brilliant Care – hiustenhoitoaine, 200 ml

Lunette: Kuukautiskuppi

CocoVi: Raakasuklaan valmistuspaketti, 410 g

Figure 7.2: FeedThroat retailer view in the Web application.

7.1.1 Integration to UbiQloud

FeedThroat uses three of the four available modules of the UbiQloud platform. The three modules used are *Recognize* (M1), *Connect* (M3), and *Share* (M4). In addition, all the sub-modules of these modules are used. The UbiQloud interfaces used are *Client APIs* and *Social Gateway*.

FeedThroat uses the *User Profiles* (m1.1) sub-module to authenticate users and retrieve information about the authenticated user's Facebook friends. The friend information is used to provide the possibility for the user to share lists between friends and to see which products are popular or used among the friends. The other sub-module *Objects* (m1.2) of the *Recognize* module is used to identify products based on scanned barcodes. Users can also expand the UbiQloud *Object* database through FeedThroat UI by providing a textual representation for the unrecognized object.

All the data changes made in FeedThroat are pushed to other FeedThroat instances through the *Push* (m3.1) sub-module. The iOS version uses native XMPP for real-time communication and the Web version uses XMPP over WebSocket. FeedThroat also uses the UbiQloud RESTful API to fetch data during the application initialization process and during other operations, such as fetching friend information.

The *Share* (M4) module is used by the retailer users of FeedThroat to disseminate information to social media. For example, ads created in FeedThroat can be automatically shared to Facebook.

7.1.2 User Tests

For testing the application with real users, a small user test [57] was organized with 15 participants (divided to two groups) in the Iso Omena shopping center. The participants were given a task to buy a set of products to organize a party. For the task, the participants were divided to pairs and each pair gained a smartphone with a pre-installed FeedThroat application. In addition, the list of needed products was populated to FeedThroat before the test. The list contained products from various shops in order to prevent one group for purchasing everything from a single shop.

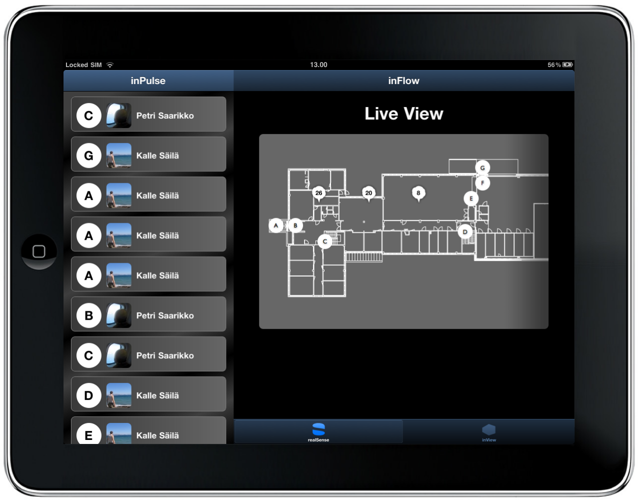
The idea was that the pairs did not decide beforehand which pair is going to buy what. Instead, the pairs used FeedThroat to scan and check products as they purchased them. As soon as a product was checked by one pair, all the other pairs gained a notification that the particular product was purchased.

The test showed that communication (both pull and push) with UbiQloud worked well even in a crowded shopping center over a *3G* cellular network. The latency of the communication was not measured, but at least the users did not complain about slowness in data fetching (e.g., scanned barcode was sent to UbiQloud and product information was received fast). Also, all push notifications were sent correctly without any dropped messages.

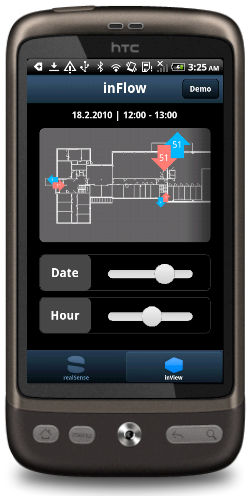
7.2 Positioning Context: InView

InView is a cross-platform mobile application for monitoring people flow indoors. The application was developed with Web technologies — that is, HTML, *Cascading Style Sheets (CSS)*, and JavaScript — and cross-compiled to multiple mobile *Operating Systems (OS)* with PhoneGap[2]. The main purpose of the InView application is to visualize both authenticated and anonymous movements in indoor spaces. The location data was obtained from two types of sensors integrated to UbiQloud. A set of UHF readers were used to identify individual users and their location based on user-specific RFID tags. For anonymous people flow (i.e., the amount and direction of people passing by), a custom ZigBee-based WSN was used.

Figure 7.3 illustrates the three main views of the application. The first view is a location feed sidebar (a) that is updated in real-time based on the movements of identified users. The second view, called *Live View* (b), visualizes the people flow by drawing a hit map on top of a floor map. The final view, called *History View* (c), can be used to retrospectively view the amount of people passed by each sensor on both directions at a given time point. The *History View* contains two sliders to adjust the date and the time within one hour interval.



(a) Sidebar and (b) Live View in iPad



(c) History View in HTC Desire

Figure 7.3: InView UI views in Android and iOS devices.

[2]PhoneGap, http://phonegap.com/

7.2.1 Integration to UbiQloud

InView utilizes the first three main modules (M1, M2, and M3) of UbiQloud. From the main modules, the most relevant sub-modules for the application are *Indoors* (m2.1), *User Profiles* (m1.1), *Push* (m3.1), and *Aggregate* (m3.2). In addition, all UbiQloud interfaces are used (i.e., *Client APIs*, *Smart Gateway*, and *Social Gateway*).

The indoor space, Aalto Design Factory monitored with InView, has two types of sensors integrated to UbiQloud. For people flow monitoring, there is a ZigBee-based WSN that monitors movements and sends each pulse to UbiQloud, which in turns pushes the data to a dedicated channel subscribed by InView. As the data arrives to InView, a small animation illustrating activity and the direction of the movement is presented in the *Live View* at the spot of the sensor.

For identifiable movements, there is a set of UHF readers installed in the space, which are integrated to UbiQloud. When a reader detects an RFID tag, the ID of the tag is sent to UbiQloud. UbiQloud then maps that ID to a specific user profile and aggregates that profile with social media profiles of that user into a single user object. The user object is then pushed to the sensor's channel. After receiving the information, InView shows the user in the sidebar and displays an animation in the *Live View* to show the location of the reader.

In addition to real-time updates, InView fetches the history data generated by the sensors through UbiQloud's RESTful API. The history data is used in the sidebar to show the recent movements of authenticated users and in the *History View* to visualize people flow on the selected date and time in the past.

7.3 Conclusions

Based on the sample applications developed on top of the UbiQloud platform, it can be said that the platform can be used with real applications in real-world settings and it fulfills the requirements set in Chapter 5. Firstly, the platform provides both pull-based and push-based communication channels (requirements R1 and R2), as showed in both sample applications. Secondly, UbiQloud is able to identify people based on different profiles (e.g., RFID profile in InView, requirement R3) and objects based on identifiers (e.g., product recognition in FeedThroat, requirement R5). Thirdly, the platform provides information on user's social connections (e.g., friends' interests in FeedThroat, requirement R3). Fourthly, UbiQloud provides ways to integrate sensors (requirement R6) that can be used in positioning (requirement R4) as shown in InView, and is able to aggregate information (e.g., combined user profile in InView, requirement R7). Lastly, the platform allows sharing information to social media services (e.g., ad sharing in FeedThroat, requirement R8).

Chapter 8

Testing and Evaluation

In this Chapter, the settings and tools used to test UbiQloud in a laboratory environment are presented. In addition, the results gained from the tests are presented and discussed.

8.1 Setup

The main goal of the experiments was to measure the performance and scalability of the UbiQloud platform in a controlled environment. The idea was to simulate a situation, in which the platform receives a fair amount of requests from multiple clients simultaneously [64]. In the tests, the amount of requests was set to 10000 and the amount of clients was set to 50. The values for the request count and concurrency were decided based on two criteria. First, it was important to generate enough traffic to monitor the performance of the platform under heavy load. Second, it was important that the testing tool also performed well in order to exclude any anomalies not related to UbiQloud (e.g., if the concurrency was set over 50, the client machine was not able to run the test reliably).

The experiments included test runs against two kinds of methods: requesting a resource from the *client interface* and adding new data through the *sensor interface*. The first API method tested was a method used to request a specified activity of an item (HTTP GET, http://api.ubiqloud.io/object/1/activity/1). The second method tested was a method used by sensors to publish data (HTTP GET/POST, http://api.ubiqloud.io/sensor). The methods were selected for the tests due to their internal differences. Methods used for requesting UbiQloud resources (e.g., the former test method) do not manipulate data, meaning that UbiQloud does not need to push changes to any channel. Methods that do manipulate data in UbiQloud (e.g., the latter test method) trigger the *Push* (m3.1) module resulting to additional tasks in the platform. In addition, the latter test method is part of the *Smart Gateway*, which has looser security procedures (i.e., UbiQloud does not check any timestamps or regenerate/compare hashes, which yields faster response times).

Furthermore, to test the scalability of the platform, both tests were conducted against two kinds of server setups: against a *single* instance of UbiQloud running on the server and against *five* UbiQloud instances running side by side on the server. The idea was to determine how well the platform scales and does the amount of instances correlate the performance benefits (i.e., is it worth to use a higher number of instances). The number of instances (five) was selected to clearly separate the result between the single and scaled UbiQloud, but in general the number of instances could be anything above one. In real-world settings the number of instances should be selected per use case basis. Both server setups included Apache[1] HTTP server as a reverse proxy (i.e., redirecting requests between the Play! framework and clients through the standard HTTP port 80). In addition, Apache handled the load balancing in the scaled UbiQloud setup. The tests ran against the single UbiQloud could have been conducted without the reverse proxy, but for better comparison it was used in both cases.

All the tests were carried out in a high speed *Local Area Network (LAN)* (100/100 Mbit/s) network. The server machine was a Mac mini with a 64-bit OS X 10.7.4 operating system, a 2.0 GHz quad-core Intel Core i7 CPU, and 8GB of RAM. The client machine was a MacBook Pro with a 64-bit OS X 10.7.4 operating system, a 2.53 GHz Intel Core 2 Duo CPU, and 4 GB of RAM.

8.2 Tools

Apache Bench (ab) [65] was used on the client machine to create the desired amount of traffic and concurrency. *Ab* is an open-source command line tool to benchmark HTTP-based servers. It is pre-installed on many operating systems and supports a wide variety of options, such as authentication, concurrency, and result output in different formats. Listing 8.1 shows an example of an *ab* script that performs 10000 request with a concurrency of

[1]Apache HTTP Server, http://httpd.apache.org/

50 as well as outputs the results to a *Comma Separated Values (CSV)* file called `results.csv`, whereas Listing 8.2 shows the console output of that test run.

```
ab -n 10000 -c 50 -e results.csv http://api.ubiqloud.io/
   sensor?data=4F97D177E50D00072B1
```

Listing 8.1: Example of an Apache Bench script.

```
Server Software:        Play!
Server Hostname:        api.ubiqloud.io
Server Port:            80

Document Path:          /sensor?data=4F97D177E50D00072B1
Document Length:        110 bytes

Concurrency Level:      50
Time taken for tests:   7.070 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      5380000 bytes
HTML transferred:       1100000 bytes
Requests per second:    1414.33 [#/sec] (mean)
Time per request:       35.352 [ms] (mean)
Time per request:       0.707 [ms] (mean, across all
   concurrent requests)
Transfer rate:          743.08 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.3      1       4
Processing:    10   34   7.1     34     122
Waiting:       10   34   7.1     34     122
Total:         11   35   7.1     35     123

Percentage of the requests served within a certain time (ms)
  50%     35
  66%     37
  75%     38
  80%     39
  90%     42
  95%     48
  98%     56
  99%     59
 100%    123 (longest request)
```

Listing 8.2: Example of an Apache Bench console output.

8.3 Testing the Client Interface

Figure 8.1 represents the benchmark results of the tests against the client interface (i.e., http://api.ubiqloud.io/object/1/activity/1). As stated, the same tests were performed against the single UbiQloud instance and the scaled UbiQloud (5 instances). Both tests consisted of 10 identical test runs in order to gain some variance for statistical analysis. The graph lines are constructed of median values from the 10 test runs.

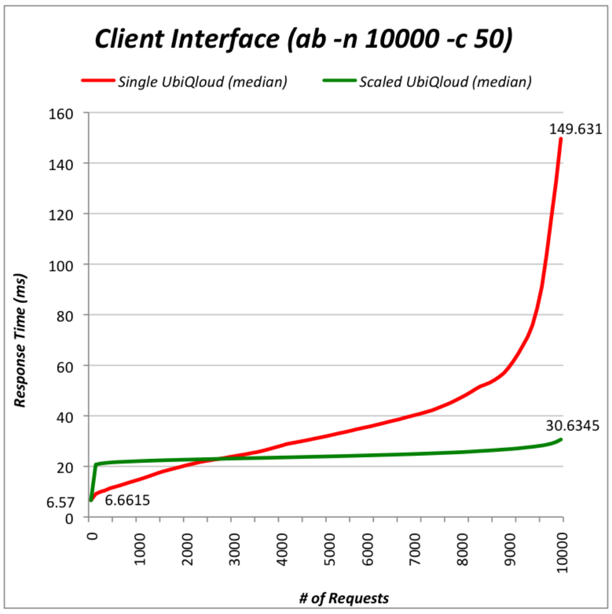


Figure 8.1: Benchmark results for a single request response time against the client interface of the single and scaled UbiQloud with 50 concurrent users and 10000 requests.

The graph clearly shows the superiority of the scaled UbiQloud in response time as the amount of requests increase (maximum response time of the scaled UbiQloud for a single request was around 30ms, whereas the corresponding value of the single instance was slightly below 150ms). However, the single instance of UbiQloud was capable of responding faster before the request count increased over 3000. The reason for this is the nature of the request (i.e., the method called in UbiQloud). The request did not cause any data manipulation on UbiQloud, meaning that UbiQloud did not need much time processing the request. With a smaller number of requests, the time taken in the load balancer to route requests increased the response time with the scaled UbiQloud. Furthermore, the response time per request with the single instance increases drastically when the request count reaches over 9000. The reason for this might be that the Play! framework reached its limit for simultaneous requests or the reverse proxy was not configured properly (neither of the two were optimized).

8.4 Testing the Sensor Interface

In these tests, the interface tested was the sensor interface of UbiQloud. In comparison to the client interface tests, the method tested in the sensor interface manipulated data in UbiQloud, which increased the processing time in the platform. The data sent to UbiQloud was formatted similar to the data sent from the RealSense sensors, meaning that with each request, UbiQloud needed to parse the data, convert the data to XML, store the data, and push the data to the sensor channel. Figure 8.2 represents the results of the sensor interface tests. The values are again median values from 10 test runs per test case.

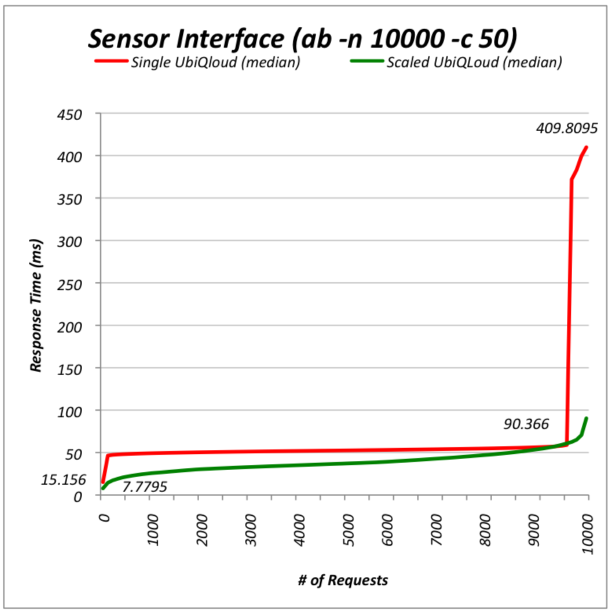


Figure 8.2: Benchmark results for a single request response time against the sensor interface of the single and scaled UbiQloud with 50 concurrent users and 10000 requests.

Overall, the graph clearly shows that methods manipulating data in UbiQloud need more time for processing (longest response time with the scaled UbiQloud was around 90ms and with the single instance around 410ms). In contrast to the client interface tests, the scaled instance performed better all the way in the sensor interface test because the time taken in the load balancer is relatively shorter. Furthermore, in these tests the increase in response times with the single UbiQloud towards the end is even more dramatic.

8.5 Evaluation

The performance tests clearly showed that UbiQloud is responsive even with one instance running on the server [64, 66]. In addition, using several UbiQloud instances in parallel showed that the platform scales well and is able to process a large number of requests from several clients. Figure 8.3 shows the mean count of requests served in a second in all the tests. Without scaling, UbiQloud is able to process nearly 1300 request per second when a client only requests a resource and over 750 requests per second with data manipulation requests. The corresponding values for the scaled UbiQloud are 2081 requests per second and 1277 requests per second. However, the tests were performed in a controlled laboratory environment with a high speed wired Internet connection, which is not often the case in real-world settings. In real-world settings, clients often utilize lower speed wireless networks, which in terms affects the response time. All in all, the tests showed promising results of the performance and scalability of the platform, but if the platform would be publicly available, additional testing with additional parameters and in different kinds of networks should be performed.

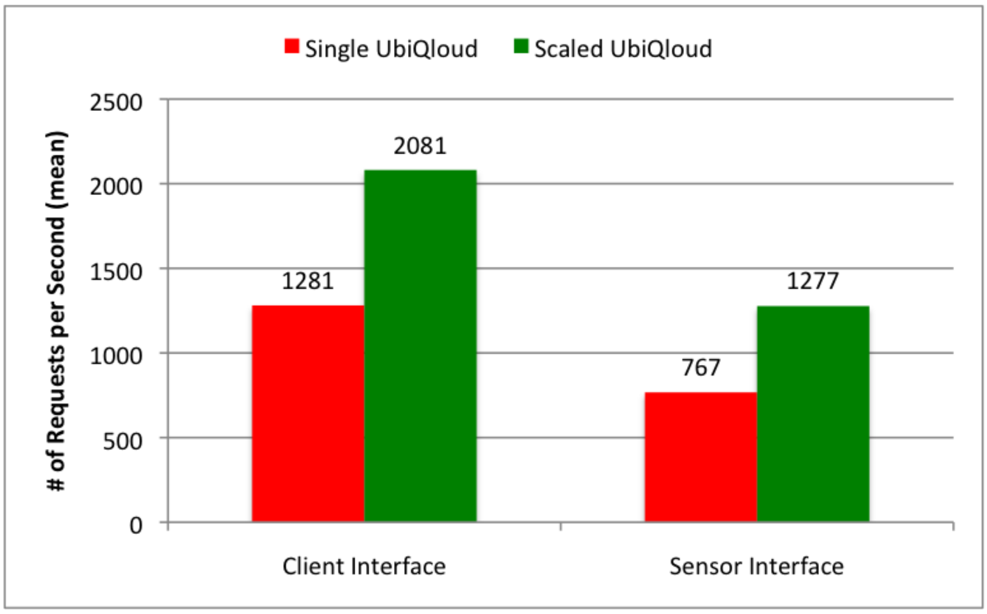


Figure 8.3: Requests per second served during the tests.

Chapter 9

Conclusions

In this Chapter, this Thesis is concluded by first revisiting the research objectives. Then, the main contributions of the author are presented. Next, the main conclusions are drawn based on the results from the literature review and empirical validation. Finally, suggestions for future work are discussed.

9.1 Research Objectives Revisited

Before proceeding with the results and conclusions, the research questions of this Thesis (cf. Chapter 4) are revisited. The research questions of this Thesis were:

Q1: How to integrate server push, *Smart Object* integration, positioning, and *Social Login* services under a single cloud-based platform?

Q2: How to transform indoor environments to *Smart Spaces* by utilizing the existing plain and smart objects?

Q3: How to seamlessly integrate digital services as part of the everyday physical activities?

9.2 Main Contributions

The main contributions of this Thesis are as follows:

- Requirements analysis for the platform, which is partly conducted by the Author
- Design of a scalable cloud-based platform offering services for developing real-time, location-aware WoT applications for *Smart Spaces*, which has been solely designed by the Author
- Implementation of a prototype, UbiQloud, whose sole implementer the Author has been
- Validation of the design and prototype with performance tests and two sample applications, FeedThroat and InView (both mainly developed by the Author).

9.3 Results

UbiQloud shows that it is possible to develop a cloud-based platform offering services related to server push, Web of Things, positioning, and social media from a single entry point. In addition, because of the module-based architecture of the platform, it is relatively easy to add new services and extend existing ones in the future.

Developing new applications on top of UbiQloud as well as utilizing UbiQloud services in existing applications should be relatively easy for developers familiar with third-party APIs due to the following reasons: UbiQloud uses standardized technologies, it is based on the widely used RESTful architecture, and it uses well-known security procedures for secure API communication. For a novice developer, the learning curve might be fairly steep, but not any steeper than using other third-party services.

The sample applications showed (cf. Chapter 7) that UbiQloud can be used with real applications in real-world settings and that the implementation meets the requirements set (Q1). In addition, the functionality and integration to UbiQloud showed that the platform provides a gateway for creating *Smart Spaces* (Q2) and that existing objects can be brought as part of the Web of Things (Q3). Moreover, the performance benchmarks showed that UbiQloud performs well and can be easily scaled if needed.

All in all, the results suggest that UbiQloud is a powerful and scalable platform for developing state-of-the-art Web applications for the Web 3.0 era. There were no indications that the platform would not be suitable for real-world application development. Furthermore, the wide range of services provided by UbiQloud would ease the development of Web applications and reduce the need to rely on multiple third-party APIs.

9.4 Future Work

There are several possibilities for future work regarding to the work presented in this Thesis. First of all, the platform could be extended in a variety of places. The amount of social media services supported by the platform could be extended to allow users from other services use their existing credentials; the range of sensor-specific drivers could be expanded to offer more usable data; the coverage of supported protocols for sensor integration could be broaden (e.g., XMPP and *Constrained Application Protocol, CoAP* [67]) to allow integration of sensors not capable of communicating with the existing interfaces.

In addition, the use of XMPP could be increased by supporting, for example, strict client-to-client communication instead of publish/subscribe to allow more targeted communication and interaction between users. Furthermore, there could be a platform-wide notification channel for broadcasting events to better support event-driven architectures.

Furthermore, the APIs could be protected using OAuth 2.0 [68] instead of a custom solution to simplify the communication with the APIs even more. Although, the OAuth 2.0 specification is not yet finalized, some of the popular APIs (e.g., Facebook Graph API) already use it for secure API communication.

Finally, it would be interesting to release UbiQloud as a public service in order to see how well the platform performs with multiple real-world applications developed outside the research community.

Bibliography

[1] A. Taivalsaari and T. Mikkonen. The Web as an Application Platform: The Saga Continues. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'11)*, pages 170–174. IEEE, 2011. doi: 10.1109/SEAA.2011.35.

[2] T. Mikkonen and A. Taivalsaari. Reports of the Web's Death Are Greatly Exaggerated. *Journal of Computer*, 44(5):30–36, 2011. doi: 10.1109/MC.2011.127.

[3] S. Murugesan. *Web X.0: A Road Map*, volume 1, pages 1–11. Information Science Reference, 2010.

[4] M. Meeker, S. Devitt, and L. Wu. Internet Trends. Technical report, Morgan Stanley, 2010. URL `http://www.morganstanley.com/institutional/techresearch/pdfs/Internet_Trends_041210.pdf`.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Updated by RFCs 2817, 5785, 6266, 6585.

[6] M. Pohja. Server Push for Web Applications via Instant Messaging. *Journal of Web Engineering*, 9(3):227–242, 2010.

[7] I. Hickson. HTML5. W3C working draft, W3C, March 2012. URL `http://www.w3.org/TR/2012/WD-html5-20120329/`.

[8] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, November 2008. URL `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[9] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. URL `http://www.ietf.org/rfc/rfc4627.txt`.

[10] A. Ginige and S. Murugesan. The essence of web engineering - managing the diversity and complexity of web application development. *Journal of Multimedia*, 8(2):22 –25, apr-jun 2001. ISSN 1070-986X. doi: 10.1109/MMUL.2001.917968.

[11] Y. Huang and H. Garcia-Molina. Publish/Subscribe in a Mobile Environment. *Journal of Wireless Networks*, 10:643–652, 2004. ISSN 1022-0038.

[12] P. Saint-Andre, K. Smith, and R. Tronçon. *XMPP: The Definitive Guide*. O'Reilly Media, Inc., Sebastobol, CA, the United States of America, 2009. ISBN 978-0596521264.

[13] P Saint-Andre. Streaming XML with Jabber/XMPP. *IEEE Internet Computing*, 9(5):82–89, 2005. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1510608`.

[14] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011. URL `http://www.ietf.org/rfc/rfc6120.txt`.

[15] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011. URL `http://www.ietf.org/rfc/rfc6121.txt`.

[16] I. Paterson and P. Saint-Andre. XEP-0206: XMPP over BOSH. Draft Standard, July 2010. URL `http://xmpp.org/extensions/xep-0206.html`.

[17] P. Millard, P. Saint-Andre, and R. Meijer. XEP-0060: Publish-Subscribe. Draft Standard, July 2010. URL `http://xmpp.org/extensions/xep-0060.html`.

[18] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. URL `http://www.ietf.org/rfc/rfc793.txt`. Updated by RFCs 1122, 3168, 6093, 6528.

[19] I. Paterson, D. Smith, P. Saint-Andre, and J. Moffitt. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). Draft Standard, July 2010. URL `http://xmpp.org/extensions/xep-0124.html`.

[20] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011. URL `http://www.ietf.org/rfc/rfc6455.txt`.

[21] J. Moffit and E. Cestari. An XMPP Sub-protocol for WebSocket. Internet Draft, June 2011. URL `http://tools.ietf.org/html/draft-moffitt-xmpp-over-websocket-00`.

[22] I. Hickson. The WebSocket API. W3C candidate recommendation, W3C, December 2011. http://www.w3.org/TR/websockets/.

[23] R. Krannenburg. *The Internet of Things. A critique of ambient technology and the all-seeing network of RFID.* Network Notebooks 02. Institute of Network Cultures, 2008.

[24] Internet of things in 2020, roadmap for the future. Technical report, INFSO D.4 Networked Enterprise & RFID INFSO G.2 Micro & Nanosystems in co-operation with the Working Group RFID of the ETP EPOSS, May 2008.

[25] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. *From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices*, chapter 5, pages 97–129. Springer, April 2011. ISBN 978-3-642-19156-5.

[26] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton. Smart objects as building blocks for the internet of things. *IEEE Internet Computing*, 14(1):44–51, 2010. URL `http://oro.open.ac.uk/31631/`.

[27] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, 2010. URL `http://linkinghub.elsevier.com/retrieve/pii/S1389128610001568`.

[28] T. Kawashima, J. Ma, R. Huang, and B. O. Apduhan. GUPSS: A Gateway-Based Ubiquitous Platform for Smart Space. *2009 International Conference on Computational Science and Engineering*, pages 213–220, 2009. doi: 10.1109/CSE.2009.265. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5284203`.

[29] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. *Journal of Evolution*, pages 1–8, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5678452`.

[30] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.9164\&amp;rep=rep1\&amp;type=pdf`.

[31] L. Reyero and G. Delisle. A Pervasive Indoor-Outdoor Positioning System. *Journal of Networks*, 3(8):70–83, 2008. URL `http://www.academypublisher.com/ojs/index.php/jnw/article/view/1056`.

[32] H. Liu, H. Darabi, P. Banerjee, and J. Liu. Survey of Wireless Indoor Positioning Techniques and Systems, 2007. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4343996`.

[33] A. Popescu. Geolocation API Specification. W3C proposed recommendation, W3C, May 2012. http://www.w3.org/TR/2012/PR-geolocation-API-20120510/.

[34] A. Mayfield. What is social media? *Journal of Networks*, 1.4: 36, 2008. URL `http://www.icrossing.co.uk/fileadmin/uploads/eBooks/What_is_Social_Media_iCrossing_ebook.pdf`.

[35] How Many Social Networking Websites Are There? URL `http://howmanyarethere.net/how-many-social-networking-websites-are-there/`.

[36] H. Oh and S. Jin. The Security Limitations of SSO in OpenID. *2008 10th International Conference on Advanced Communication Technology*, 3: 1608–1611, 2008. URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4494089`.

[37] M. N. Ko, G. P. Cheek, M. Shehab, and C. North. CONNECT SERVICES. *Journal of Computer*, 43(8):37–43, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5551044`.

[38] M. Böhm, S. Leimeister, C. Riedl, and H. Krcmar. Cloud Computing and Computing Evolution. In *Cloud Computing Technologies Business Models Opportunities and Challenges*, pages 1–28. CRC Press, 2010. URL `http://www.theseus.joint-research.org/assets/Wissenschaftliche-Publikationen/BoehmEtAl2009c.pdf`.

[39] S. P. Mirashe and N. V. Kalyankar. Cloud Computing. *Communications of the ACM*, 51(7):9, 2010. URL `http://arxiv.org/abs/1003.4074`.

[40] L. Savu. Cloud Computing Deployment models, delivery models, risks and research challanges. *Information Security*, 2011.

[41] P. Ljungstrand, J. Redström, and L. E. Holmquist. WebStickers: Using Physical Tokens to Access, Manage and Share Bookmarks to the Web. In *Proceedings of DARE 2000 Designing Augmented Reality Environments*, pages 23–31. ACM Press, 2000. doi: 10.1145/354666.354669. URL `http://portal.acm.org/citation.cfm?id=354669\&coll=ACM\&dl=ACM\&CFID=64850236\&CFTOKEN=86024593\#`.

[42] V. Trifa, D. Guinard, V. Davidovski, A. Kamilaris, and I. Delchev. Web Messaging for Open and Scalable Distributed Sensing Applications. *Web Engineering*, 6189:129–143, 2010. URL `http://www.springerlink.com/index/G21NG6L3T60147H7.pdf`.

[43] M. Blackstock, N. Kaviani, R. Lea, and A. Friday. MAGIC Broker 2: An open and extensible platform for the Internet of Things. *Computing*, (Nov. 29 2010-Dec. 1 2010):1–8, 2010. URL `http://dx.doi.org/10.1109/IOT.2010.5678443`.

[44] T. Springer, D. Schuster, I. Braun, J. Janeiro, M. Endler, and A. Loureiro. A Flexible Architecture For Mobile Collaboration Services. *Proceedings of the ACMIFIPUSENIX international middleware conference companion on Middleware 08*, page 118, 2008. doi: 10.1145/1462735.1462770. URL `http://portal.acm.org/citation.cfm?doid=1462735.1462770`.

[45] R. Lübke, D. Schuster, and A. Schill. MobilisGroups: Location-based group formation in Mobile Social Networks. *Computer Networks*, pages 502–507, 2011. doi: 10.1109/PERCOMW.2011.5766941. URL `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=5766941`.

[46] D. Schuster, T. Springer, and A. Schill. Service-based development of mobile real-time collaboration applications for Social Networks. *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops PERCOM Workshops*, pages 232–237, 2010. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470662`.

[47] N. Jansen. Design and Implementation of a Web Gateway for Mobile Collaboration Services. Master's thesis, TU Dresden, 2011.

[48] Pusher. URL `http://pusher.com/`. [last checked: June 7, 2012].

[49] Facebook. URL `http://www.facebook.com`. [last checked: May 7, 2012].

[50] LogicAlloy ALE Server. URL `http://www.logicalloy.com/`. [last checked: May 7, 2012].

[51] M. Gudgin, M. Hadley, N. Mendelsohn, Y. Lafon, J.-J. Moreau, A. Karmarkar, and H. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/soap12-part1/.

[52] Cosm. URL `https://cosm.com/`. [last checked: May 7, 2012].

[53] Qubulus. URL `http://www.qubulus.com/`. [last checked: June 7, 2012].

[54] Gigya. URL `http://www.gigya.com/`. [last checked: June 7, 2012].

[55] Janrain. URL `http://janrain.com/`. [last checked: June 7, 2012].

[56] 4D-Space. URL `http://mide.aalto.fi/en/4D-Space`. [last checked: June 7, 2012].

[57] P. Ojanen, P. Vuorimaa, P. Saarikko, and S. Uotinen. Sharing and Browsing Social Objects in Physical Space within Close-Knit Groups. In *1st International Workshop on Mobile Interaction in Retail Environments*, 2011.

[58] P. Ojanen, P. Vuorimaa, P. Saarikko, and S. Uotinen. *Ubeel: Generating Local Narratives for Public Displays from Tagged and Annotated Video Content*. 2011. URL `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-720/`.

[59] S. Wellman. Google Lays Out Its Mobile User Experience Strategy - mobility Blog, April 11 2007. URL `http://www.informationweek.com/blog/229216268`. [last checked: June 7, 2012].

[60] Play! Framework. URL `http://www.playframework.org/`. [last checked: June 7, 2012].

[61] Openfire. URL `http://www.igniterealtime.org/projects/openfire/`. [last checked: June 7, 2012].

[62] MySQL. URL `http://www.mysql.com/`. [last checked: June 7, 2012].

[63] *Signature Version 2 Signing Process.* Amazon Web Services. URL `http://docs.amazonwebservices.com/general/latest/gr/signature-version-2.html`. [last checked: June 7, 2012].

[64] Benchmarking APIs using PerfectAPI vs Express.js vs Restify.js. URL `http://blog.perfectapi.com/2012/benchmarking-apis-using-perfectapi-vs-express.js-vs-restify.js/`. [last checked: June 7, 2012].

[65] ab - Apache HTTP server benchmarking tool. URL `http://httpd.apache.org/docs/2.0/programs/ab.html`. [last checked: June 7, 2012].

[66] S. Ahuja and J.-E. Yang. Performance Evaluation of Java Web Services: A Developer's Perspective. *Communications and Network*, 2:200–206, 2010.

[67] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). Internet Draft, March 2012. URL `http://datatracker.ietf.org/doc/draft-ietf-core-coap/`.

[68] D. Recordon and D. Hardt. The OAuth 2.0 Authorization Framework. Internet Draft, May 2012. URL `http://tools.ietf.org/html/draft-ietf-oauth-v2-26`.