Veli-Matti Ojala

Addressing the interoperability challenge of combining heterogeneous data sources in a data-driven solution

Master's Thesis
in Information Technology
December 12, 2017

University of Jyväskylä

Department of Mathematical Information Technology

Kokkola University Consortium Chydenius

Author: Veli-Matti Ojala
Contact information: veli-matti.ojala@outlook.com
Phonenumber: 0445335730
Title: Addressing the interoperability challenge of combining heterogeneous data sources in a data-driven solution
Työn nimi: Heterogeenisten datalähteiden yhteentoimivuushaaste dataohjautuvissa järjestelmissä
Project: Master's Thesis in Information Technology
Page count: 78
Abstract: Data-driven solutions often combine data from various and heterogenic data sources. These data sources might use different network layer protocols, message layer protocols, data formats and semantical models. The combination of these creates an interoperability challenge since different protocols do not interoperate with each other. In the IoT-domain these challenges are often solved within systems, not among them. This creates a siloed structure for many IoT-systems. This Thesis observes the interoperability challenge on four layers and presents some possible solutions to solve these problems in a data-driven case example.
Suomenkielinen tiivistelmä: Dataohjautuvat ratkaisut yhdistävät usein erilaisten ja heterogeenisten tietolähteiden tietoja. Nämä tietolähteet voivat käyttää erilaisia verkkokerrosprotokollia, viestikerroksen protokollia, dataformaatteja ja semanttisia malleja. Näiden yhdistelmä luo yhteentoimivuuden haasteen, koska usein eri protokollat tai formaatit eivät toimi toistensa kanssa. IoT-ratkaisuissa nämä haasteet ratkaistaan usein järjestelmien sisällä, ei niiden välillä. Tämä luo siilomaisia rakenteita IoT-järjestelmien välille. Tämä opinnäytetyö esittelee yhteentoimivuusongelman neljällä kerroksella ja lisäksi ehdottaa joitain mahdollisia ratkaisuja näiden ongelmien ratkaisemiseksi datapohjaisen esimerkkitapauksen avulla.
Keywords: Interoperability, Network protocols, Message protocols, Data Format, Semantic models, Data-Driven
Avainsanat: Yhteentoimivuus, Verkkokerrosprotokollat, viestikerroksen protokollat, dataformaatit, semanttiset mallit, dataohjautuvuus

Preface

This thesis was truly an educating experience. It deepened my knowledge about the phenomenon surrounding the IoT-solutions. I'm confident that lessons learned from making this thesis give me a good ground knowledge for building my professional career. It wasn't always easy, but in the end, it has paid off.

I would like to thank my instructor Henri Leisma at Ambientia Oy for giving me valuable information about the exciting domain of software development and business design cases. I'm extremely impressed by Henri's technical expertise, but also by his people and business skills. It was an enjoyable experience to have him as a colleague.

Finally, I thank my supervisers Ismo Hakala and Joakim Klemets from the University of Jyväskylä. They guided me all the way through this process and helped me to stay focused on what truly matters. Their support was also vital when more focused descriptions of phenomena were needed. Both of them have a vast domain and technical expertise, which made my job easier.

Glossary

3G	Third Generation wireless mobile telecommunications
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
BA	Business Analytics
BI	Business Intelligence
Bluetooth	Commercial Wireless technology
CBOR	Concise Binary Object Representation
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CR	Carriage Return
CSV	Comma Separated Values
DB	Database
DDD	Data-Driven-Decision making
FMI	Finnish Meteorolocigal Institute
GPRS	General Packet Radio Service
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IANA	Internet Assigned Numbers Authority
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IO	Input/Output
IoT	Internet of Things
ISO	International Organisation for Standardisation

JSON	JavaScript Object Notation
JSON-LD	JSON Linked Data
LF	Linefeed
LoRaWAN	MAC for WAN
M2M	Machine-to-communications
MAC	Media Access Protocol
MQTT	Message Queue Telemetry Transport
NoSQL	non SQL / Not only SQL database
OGC	Open Geospatial Consortium
OWL	Web Ontology Language
POC	Proof of Concept
Pub/Sub	Publish-Subscribe paradigm
QoS	Quality of Service
RAM	Random Access Memory
RDF	Resource Description Framework
REST	Representational state transfer
SDRAM	Synchronous Dynamic Random-Access Memory
SAS	Semantic Annotation Service
SGS	Semantic Gateway as Service
SemSOS	Semantic Sensor Observation Service
SoC	System on a Chip
SPARQL	SPARQL Protocol And RDF Query Language
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
SSN	Semantic Sensor Network
SWE	OGC Sensor Web Enablement
TCP	Transmission Control Protocol

UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
W3C	World Wide Web Consortium
WAN	Wide-Area Network
Wi-Fi	IEEE 802.11 Wireless Local Area network technology
WLAN	Wireless Local Area Network
WWW	World Wide Web
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
ZigBee	IEEE 802.15.4 Personal Area Network technology
Z-Wave	IEEE 802.15.4 Personal Area Network technology

Contents

1 Introduction

The amount of available data has long surpassed our ability to analyse all of it [13]. This is largely due to the fact, that computing is becoming more and more ubiquitous. The increasing growth of the Internet, mobile devices and IoT-systems offers great resources for gathering data [5] [14]. Atzori et. al. [5] define the Internet of Things (IoT) as a broad paradigm, where object or "things" implement the computational resources required to communicate over the Internet with each other. This changes world, since the resources available on the Internet, are pervasively available in various domains: homes, workplaces, factories and on many portable devices.

Examples of IoT-systems could be a heart rate watch or a smart thermostat for controlling indoor temperature. Both of them require access to the Internet to be able to communicate to end services that are located on the remote backend. To achieve this, both of them need a network connection, proper use of messaging protocols and the data in messages needs to be semantically appropriately formatted.

Alongside with this, the possibilities to enrich organisations own data with third party data has also increased dramatically [66]. At the time of writing one of the popular third party data collections, the Programmable Web [64], offered over seventeen thousand APIs to gather data from.

The combination of all of this is often referred as the Big Data [54]. Big Data is somewhat a vague expression, but is often used to describe the sheer Volume and the Velocity of the data [13]. But since the Big Data is coming in from so many sources, also the Variety of the data is an essential challenge to overcome [13]. This Variety can be seen as a challenge on many levels, starting from network layer protocols and ending all the way to the semantic and data collaboration layers. Desai et. al. [18] refer these challenges as the interoperability challenge.

Desai et. al. [18] observe the interoperability challenge of the IoT systems on three layers: the network layer, the messaging protocols and the data annotation layer. Each of these layers present an unique interoperability challenge. This Master's thesis focuses on the interoperability challenges among the messaging protocols and the different data annotation possibilities.

The first interoperability challenge, as Desai et. al. [18] state, is on the network layer. They claim, that this means the lack of interoperability among various network protocols like very low power radio protocols such as the ZigBee or the Bluetooth, but also traditional networking protocols like the Ethernet, Wi-Fi or the TCP/IP. As the TCP/IP tutorial by Network Working Group states [37], the purpose of TCP/IP protocol is to transport a data packet from source host to the destination source successfully. This principle can be generalised to all network layer protocols, and as Desai et. al. [18] state, the purpose of a network layer is to connect things.

But previously mentioned connections can be used in various ways. Different messaging protocols, such as CoAP, MQTT, XMPP and others, create the messaging protocol interoperability challenge, as stated by Desai et. al. [18]. They also claim, that each of these messaging protocols have unique architecture for the actual messaging, thus making some more suitable for some specific tasks. Diaz et. al. [19] add to this, that still one of the most wide spread messaging protocols is the HTTP [36]. This is an interoperability problem, since these messaging protocols do not interoperate without integration or translation [18].

After successfully connecting to a data-source on the network layer, and after using shared message protocol, the third interoperability challenge to solve is the data annotation [18]. Various schemas, formats and standards exist on how to present the data. XML, JSON, HTML and others, are some of the popular data formats, but these are just ways to annotate the data. Semantics, a shared understanding of what the data means, are an essentiality when creating added value from data [18]. There are various possibilities to use ontologies or standards, out of which Desai et. al. [18] mention the OGC Sensor Web Enablement (SWE) [60], Semantic Sensor Network (SSN) [86] and the Semantic Sensor Observation Service (SemSOS) [38]. As Desai et. al. [18] state, these technologies solve the semantic interoperability within that specific domain, but lack interoperability among other ontologies.

Despite these interoperability challenges, the business value of the Big Data has been broadly acknowledged [14] [65]. Data is becoming increasingly important. Data can even be seen as a design material, when creating new solutions [68]. This design method, where data is the actual driving force behind the system, allows us to enter the Data-Driven domain. The actual Data-Driven solutions vary from a few lines of program code, all the way to transforming the service offerings of organisations through service design [32]. Despite the various natures of the outcome, the unifying property is the driving factor of the data [54].

1.1 Research problem

The research question for this Master's thesis was derived from a real-world customer situation. A customer wanted to study what benefits a predictive software could enable for their business. The customer was a people transport company, which operated under the Finnish taxi regulations. The demand for transport varies a lot, and those companies that can predict future transport needs most accurately, are also often most profitable. It is unwise to have either too many or too few vehicles actively on duty. To make smarter decisions that are based on data-analysis, the transport company wanted to have a dashboard software, that would present the future transport demands of the Tampere downtown region in simple map view.

The transport company had recognized three significant events which affect the future demand. These were the occurrence of previous transports, the weather and time. For example, on sunny summer Saturday evenings there are high peaks of demand, especially in those areas, where many previous transports had been made.

To be able to make predictions about future demand variations, these pre-mentioned data sources needed to be accessed. The transport company had all information considering their previous transport actions stored locally in their relational database. The weather information was accessible through the Finnish Meteorological Institute using an HTTP-REST-API. A GPS-system for logging locations of transport vehicles shared messages using the MQTT-protocol. Finally, there was also need for accessing the map data for enabling software to present districts or grid-overlay to the map. These grids or regions were then used as a calculation unit when predicting the future demand.

Previous data sources are not an exhaustive list of all possible events that affect the demands, but they offer a sufficient amount of data so that a proof-of-concept (POC) could be built. This presented the research question of this Thesis. When combining data from multiple heterogeneous data sources, many different interoperability challenges must be solved. This also became apparent to the development team. Combining data from heterogeneous data sources, there are many interoperability challenges that need to be solved.

To make a more analytic presentation of these interoperability challenges, the research question of this Masters Thesis follows the interoperability model presented by Desai et. al. [18], where interoperability challenges are presented on four layers:

- Network layer

- Message protocol layer
- Data format layer
- Semantic layer

When developing the solution for the customer, the network layer challenges were not an issue. Despite this, these network interoperability challenges are explored in more detail in the following theoretical sections, since they are commonly present in many IoT-solutions.

2 The Interoperability challenge

Interoperability is a comprehensive paradigm, that can be seen on many levels and in different domains. One definition of interoperability was given by IEEE in [31], where they argue that interoperability is the ability for systems to exchange information so that the information is also useful after the exchange. Thus, interoperability does not mean that all systems and components involved in the information exchange need to be fully standardised. As history has shown, this is, and probably will be, a somewhat unlikely scenario. But despite the lack of shared standards, systems do need to collaborate. As Palfrey et. al. [62] state, interoperability, and sameness are two different things. As an example from the human world, people can interoperate even if they don't have a shared language. To interoperate, they just need an interpreter.

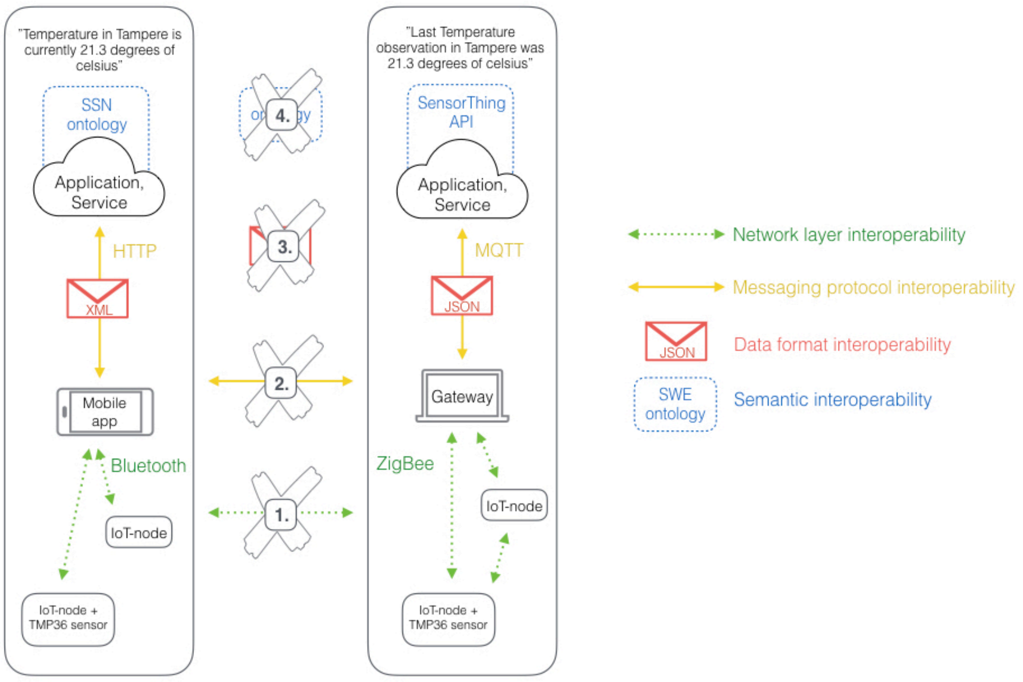

"Temperature in Tampere is currently 21.3 degrees of celsius"
SSN ontology
Application, Service
HTTP
XML
Mobile app
Bluetooth
IoT-node
IoT-node + TMP36 sensor
4.
3.
2.
1.
"Last Temperature observation in Tampere was 21.3 degrees of celsius"
SensorThing API
Application, Service
MQTT
JSON
Gateway
ZigBee
IoT-node
IoT-node + TMP36 sensor
Network layer interoperability
Messaging protocol interoperability
JSON
Data format interoperability
SWE ontology
Semantic interoperability

Figure 2.1: Example of two non-interoperable vertical IoT-silos by Desai et. al. [18] (Data annotation divided into data format and semantic interoperability).

Desai et. al. [18] explore the interoperability challenges of the IoT-domain, and present it on three layers: the network layer, the messaging protocol layer, and the data annotation layer. Each of these layers presents a unique interoperability challenge. The data annotation layer can be divided into two layers, the data format, and the semantics.

Currently, the lack of interoperability has lead to many co-existing IoT-systems which lack interoperability among one another. Desai et. al. [18] refer to them as vertical IoT-silos. By this Desai et. al. mean, that many interoperability challenges are solved within each system, but those solutions are not interoperable among other IoT-silos. Zanella et. al. [88] state, that each of these technologies has different strengths and weaknesses, and are thus suitable to different usages. They argue, that different services like the smart lighting or structural sensor observing, require different network layer technologies due to the energy and computational recourses of the end-node.

In Figure 2.1 there are two vertical IoT-silos, which both use different technology and data annotation stack. Both of them actually use the same kind of temperature sensor to provide temperature readings from the city of Tampere. The following example illustrates how this layered challenge is solved within each silo, but not between them.

The interoperability challenge between vertical IoT-silos needs to be solved either on the network layer or the messaging protocol layer. If neither of these layers interoperates, the message exchange is impossible, despite possible total data format or semantic interoperability. But interoperability can be an issue also on data format and semantic layers.

In following sections, we take a closer look at each of those layers. What common technologies there are on that layer, what strengths and weaknesses those technologies have, and how to enable interoperability among those technologies?

2.1 Network layer interoperability

The first interoperability challenge, as Desai et. al. [18] state, is in the network layer. They claim, that this means the lack of interoperability among various network protocols like the Wi-Fi [3], Bluetooth [73], ZigBee [4] or the LoRaWAN [2], but also Internet networking protocols like the UDP [63] or the TCP [37], just to name few. Each of these network protocols has unique strengths and weaknesses, and are designed to meet the connection requirements of very different scenarios. And they do not interoperate with each other.

To form an interoperating network, the first requirement for all participants is the access to the shared media. The shared media might be the same radio frequency (within range) or physical connectivity through wires. As a possible solution, Desai et. al. [18] propose the shared use of standards for data transmission. But by using the same standards in different ways, interoperability cannot be guaranteed.

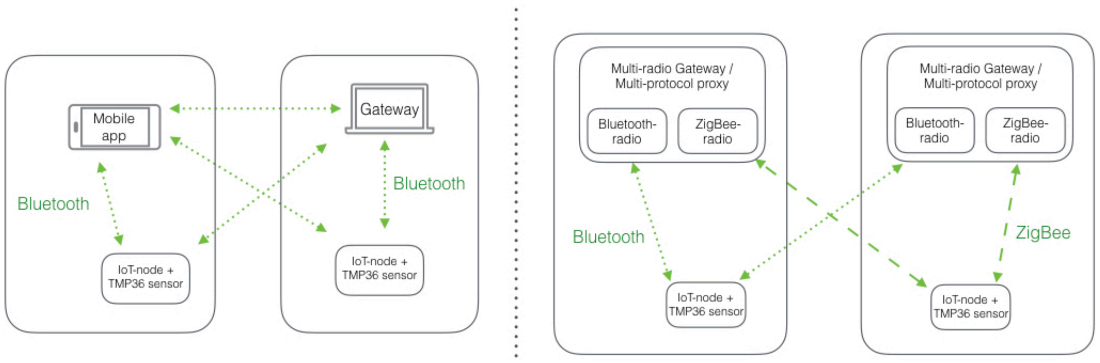


Figure 2.2: On the left: Interoperability is achieved by using the same radio technology. On the right: Interoperability is achieved by using proper Multi-radio Gateway.

Although the IEEE 802.15.4 is a standard, it can be implemented in various, possibly non-interoperable ways. Out of this Mainetti et. al. [52] give a detailed example, since they argue that the IEEE 802.15.4 only defines the MAC and physical layer of the network. They also state that ZigBee builds the network and application layers upon those presented in the IEEE 802.15.4. Thus the use of standard will not necessarily solve the radio network interoperability since there might be various non-interoperable implementations of it.

Thus, as Desai et. al. [18] state, the network layer interoperability is initially a hardware problem. All participants need interoperable processing units (radio module, modem, etc.) for data transmission processing. Naturally, the interoperability can also be achieved by using the pre-mentioned shared use of standards, which in many cases might be the use of commercial products.

Figure 2.3 illustrates how interoperability can be achieved on the Network layer. By enabling interoperability among the IoT-silos on the Network layer, rest of the development process can be done more efficiently, since the developers can choose the messaging layer, data format, and semantic technologies.

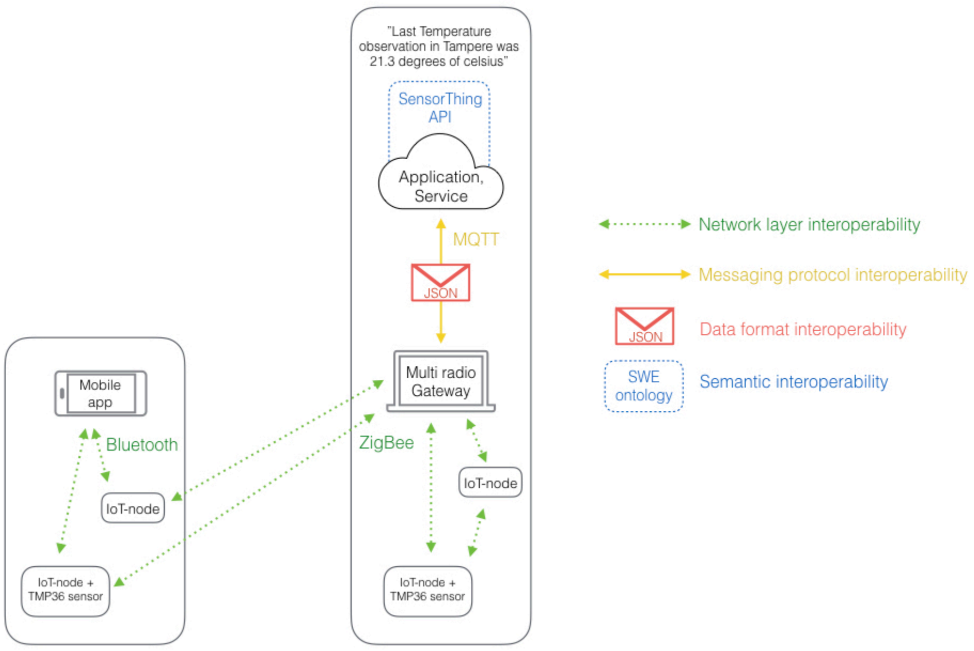

"Last Temperature observation in Tampere was 21.3 degrees of celsius"
SensorThing API
Application, Service
MQTT
JSON
Multi radio Gateway
ZigBee
IoT-node
IoT-node + TMP36 sensor
Mobile app
Bluetooth
IoT-node
IoT-node + TMP36 sensor
Network layer interoperability
Messaging protocol interoperability
JSON
Data format interoperability
SWE ontology
Semantic interoperability

Figure 2.3: An example of network layer interoperability among vertical IoT-silos using the multi-radio gateway by Desai et. al. [18].

2.1.1 Wireless network protocols

According to Tse et. al. [80] and also Schwartz [70], wireless networking is essentially about making compromises among three properties:

1. The higher the used radio frequency is, the faster the data transmission can be. (Modulation possibilities)
2. The higher the used radio frequency is, the shorter the range of communication will be. (Propagation, Fading)
3. The more efficient the use of radio channels is, more recourses are required. (Access mechanisms, Modulation)

Wi-Fi [3] and Bluetooth [73] are both commercial standard brands development based on the IEEE 802.11 standard family [34]. The Wi-Fi and the Bluetooth are both essentially Wireless Local Area Networks (WLAN). The Wi-Fi and the Bluetooth have many different products, out of which many (not all) operate on the 2.4 GHz frequency. Thus, they commonly offer connectivity among wireless devices within the local area (roughly tens of meters). Some Wi-Fi products are widespread since today practically all laptop owners also have Wi-Fi routers connecting laptops and other portable devices to the Internet/Ethernet. Wi-Fi is very suitable for fast data transmissions but requires a lot of energy and processing power. Bluetooth is on the other hand more suitable for constrained devices, but cannot match the speed of Wi-Fi.

The ZigBee [4] and the Z-Wave [74] are commercial standards based on the IEEE 802.15.4 [87] low-rate wireless personal area network standard. Both of them also have several products with different characteristics, but many of those operate on lower frequencies (most commonly under 1GHz, though the 2.4GHz is also used). This allows the ZigBee and the Z-Wave to have longer physical distances between devices, but at the same time, it restricts the speed of data transmissions. Both of the products claim to be very energy efficient, and at the same time, the throughput speed is only a fraction of what Wi-Fi can offer. Thus making them very suitable for the constrained nature of IoT-devices.

Whereas all pre-mentioned wireless technologies require a gateway or a router to form and operate the network, the LoRaWAN [2] takes another approach by introducing cellular-like connectivity to IoT-devices. By using low radio frequencies, the LoRaWAN claims to have broader communication range than cellular (GPRS, 3G) networks. The LoRaWAN is also very energy efficient. Thus it offers good IoT-connectivity. But as with other wireless technologies, the LoRaWAN is not compatible with different standards. Therefore there is a need for interoperability measures. The LoRaWAN [2] is designed to serve data to/from IoT-nodes by using TCP/IP-operating Network servers. Thus the first layer where different wireless technologies could achieve interoperability is the standard Internet protocol TCP/IP.

2.1.2 Internet Protocol (IP) and TCP, UDP

As the TCP/IP tutorial by Network Working Group states [37], the purpose of TCP/IP protocol is to transport a data packet from source host to the destination source successfully. This principle can be generalized to cover all of the network layer protocols, and as Desai et. al. [18] state, the purpose of the network layer is to connect things. It is common for third-party data providers to serve their data through publicly available APIs on the Internet [64]. Organisations may also have their own databases accessible through servers or various cloud-based systems. This means the use of the TCP/IP stack upon the link layer of the Internet. As the acronym TCP (Transmission Control Protocol) suggests, TCP offers a reliable transmission (using, e.g., the three-way handshake) between network entities, making it very common on the Internet.

But there are other possibilities to share data on the Internet. The UDP (User Datagram Protocol) is a network protocol, which according to its specifications [63], is designed to minimise the protocol activity when sending messages on the Internet. The UDP specifications also state [63], that message delivery cannot be guaranteed in UDP, since the protocol itself doesn't include any handshakes, to ensure connection establishment (though a checksum could be used to discover faulty received packages).

But UDP does have its benefits. Zhang et. al. [89] compare the network performance of voice transmission data among globally distributed entities. They claim [89], that UDP out-performs TCP stack on delay and jitter, even if the more throughput optimised TCP NODELAY was used. Zhang et. al. [89] also measured the packet loss rate. This means the portion of sent data packets, which never actually reach the receiver. Whereas each packet in TCP is securely confirmed to reach the receiver (acknowledgments), the UDP does not have any builtin checkups whether the data packets get lost or not. In their measurements, Zhang et. al. [89] discovered, that on the maximum data loss rate on UDP was 3%. On voice transmission this might be acceptable and depending on decoding techniques, the human perception might not even detect any changes in the sound quality. But if transmitting a file or an operating system online, the 3% missing from the source code would be disastrous.

2.2 Messaging protocol interoperability

Different messaging protocols, such as CoAP, MQTT, and others, create the messaging protocol interoperability challenge, as stated by Desai et. al. [18]. They also claim that each of these messaging protocols has unique architecture for the actual messaging, thus making some more suitable for some specific tasks. Diaz et. al. [19] add to this, that still one of the widest spread messaging protocols is the HTTP [36]. This is an interoperability problem since these messaging protocols do not interoperate without integration or translation [18].

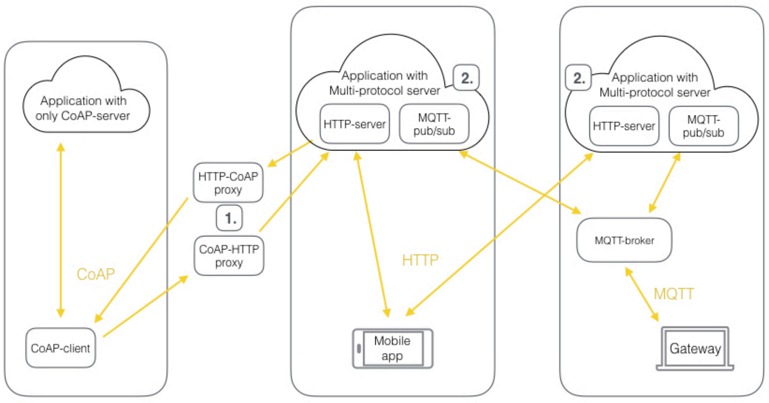


Figure 2.4: An example of (1. proxy-solution) and (2. multi-protocol server solution).

The message protocol layer can be useful for creating interoperability [18]. Out of this Desai et. al. [18] mention that the resource-constrained nature of the IoT-nodes does not limit the gateways. Thus they mention, that various heterogeneous IoT-gateways can interoperate using a multi-message protocol proxy. This data traffic would happen over the Internet. Therefore network protocol challenges like the TCP/IP or UDP/IP could also be handled.

2.2.1 HTTP

The HTTP (Hypertext Transfer Protocol) [36] is the basic building block of the World Wide Web as we know it. It is widely used in Internet browsers and servers. Fielding et. al. [22] state, that HTTP is a stateless protocol where a client sends a request to a server and waits for a response. This response can be a web-page or a file, but nothing happens without the request. This stateless design method creates many advantages since no party in the request process needs to remember or keep track of the others states.

There are also many workarounds for seemingly keeping up a state [36], like the sessions, which are accomplished by shared cookies, an included information of who made the request and what was its last request numbered. Thus both parties can act upon the information they receive with a cookie. This is the reason why the online-shop can remember what the customer has selected to the shopping cart.

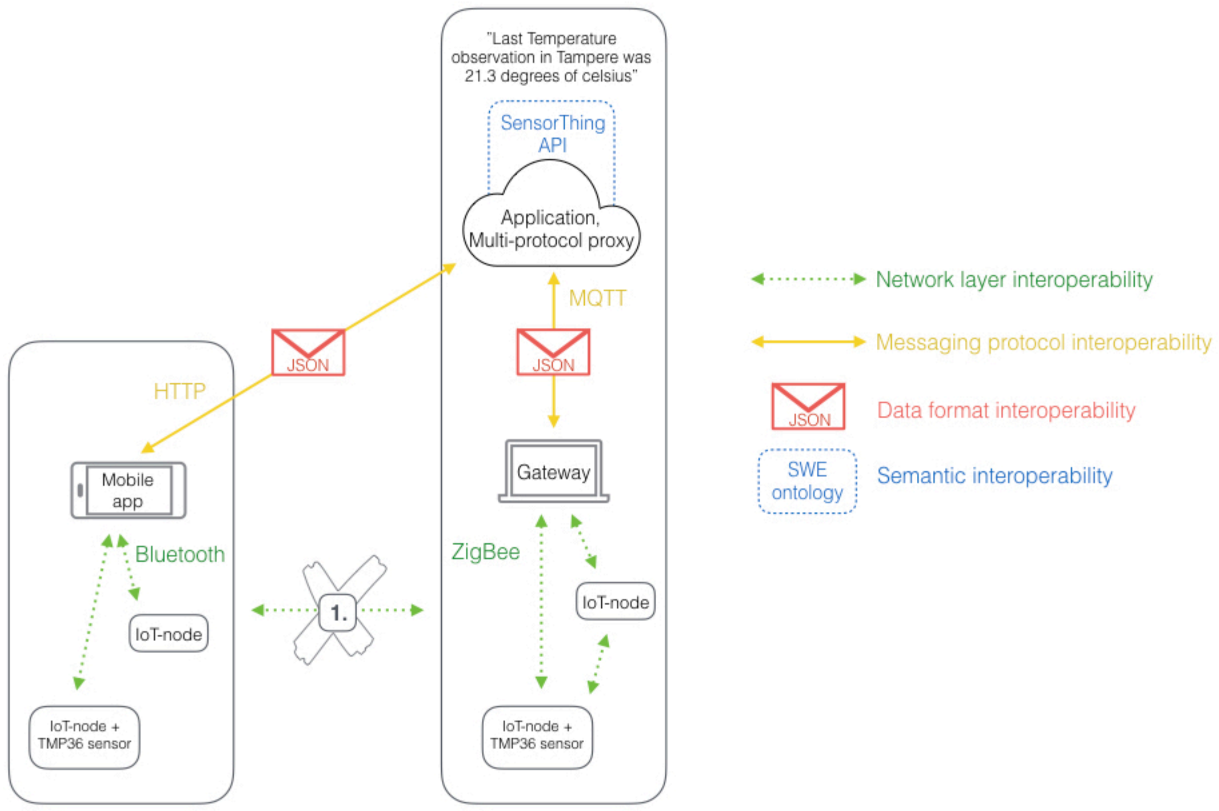

"Last Temperature observation in Tampere was 21.3 degrees of celsius"
SensorThing API
Application, Multi-protocol proxy
MQTT
JSON
HTTP
JSON
Gateway
Mobile app
Bluetooth
ZigBee
IoT-node
1.
IoT-node
IoT-node + TMP36 sensor
IoT-node + TMP36 sensor
Network layer interoperability
Messaging protocol interoperability
JSON
Data format interoperability
SWE ontology
Semantic interoperability

Figure 2.5: An example of message protocol layer interoperability among vertical IoT-silos using the multi-protocol server by Desai et. al. [18].

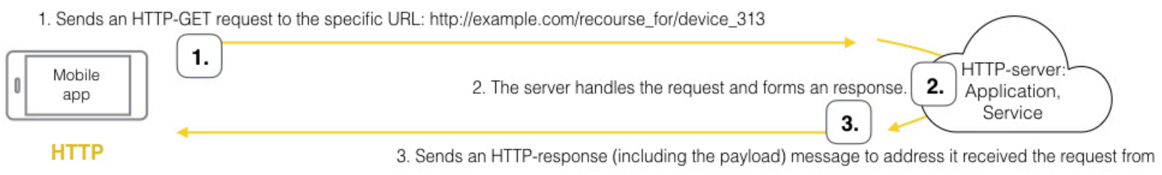

1. Sends an HTTP-GET request to the specific URL: http://example.com/recourse_for/device_313
1.
Mobile app
HTTP
2. The server handles the request and forms an response.
2.
HTTP-server: Application, Service
3.
3. Sends an HTTP-response (including the payload) message to address it received the request from

Figure 2.6: An example of HTTP request-response communication.

There are numerous request methods mentioned by Fielding et. al. [23]: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS and TRACE. Each request is pointed to a specific URI (Uniform Resource Identifiers) [22], which are in the Internet-domain the URL (Uniform Resource Location) addresses, a.k.a. the web addresses. Response to any request depends solely on the behavior of the receiving system.

Fielding et. al. [23] state, that both the request and the response always have the header fields, which represent the metadata of the HTTP-protocol transaction, like the timestamp of when the message was created, or what charsets are accepted. The header fields may also contain data about who made the request, or in which language the response is hoped to be received.

Every HTTP-protocol message has the header fields, but most messages also have what Fielding et. al. [23] refer as the payload or the body of the message. Initially, the payload was defined by Borenstein et. al. [9] into several media types like text, image or video. This payload agnosticism allows the HTTP to be used on sharing almost any possible datasets. A payload can be attached either to the response message, but can also be added to the request message itself (like POST).

As a simplified example, one enters a www-address (*http://cinetcampus.fi/studies/*) onto the address-field of an internet browser. The browser converts this to an HTTP-GET request with proper header fields, and by using the TCP/IP networking technologies sends the GET-request to the server. After when the server has processed the request, the client's browser receives a response message, that also has a payload - the text-file, that is annotated to be in HTML-format (this is the Web-page).

But when doing the same procedure to a different URL (*http://data.fmi.fi/...*), the response message has a payload of weather forecast to the Tampere region in an XML data format. This possibility to different usage has made the HTTP a common messaging protocol. But not all of the data traffic use it. This is especially true in the IoT-domain, where resource-constrained devices need to communicate over the Internet.

2.2.2 CoAP

The CoAP (Constrained Application Protocol) [39] is an HTTP-like standardized protocol, but with a much higher efficiency of data transfer. This is due to the smaller header sizes, and the fact that CoAP uses UDP on the transport layer. This cuts down many messages familiar in the TCP caused by the connection establishment, ending and the checking whether the data packet was received or not. Desai et. al. state [18], that the CoAP is primarily designed to be used in IoT-solutions and the sensor networks domains, due to its minimum need of resources.

The CoAP and HTTP have many similarities like they both use the requests and responses. But unlike in HTTP, the headers are in CoAP are in binary format. This optimises the payload usage of messages. In CoAP this is fundamentally important since it uses the UDP, every message needs to fit into single UDP datagram, whereas in TCP the fragmentation of oversized messages is a built-in feature. The use of UDP does create some issues since UDP doesn't have the same integrated acknowledgment system for confirming received messages like in the TCP. But Shelby et. al. [72] states, which the CoAP has a feature, that allows messages to be set confirmable, creating almost the same trustworthiness for message transport as in TCP.

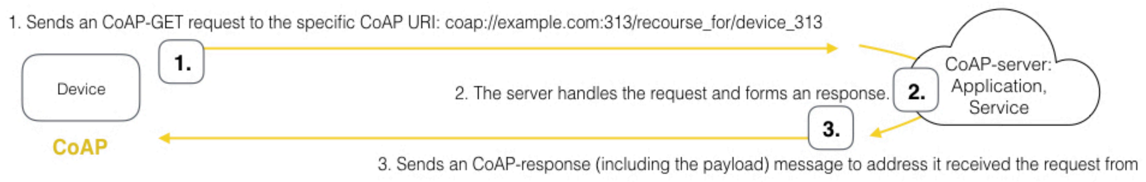

1. Sends an CoAP-GET request to the specific CoAP URI: coap://example.com:313/recourse_for/device_313
1.
Device
CoAP
2. The server handles the request and forms an response.
2.
CoAP-server: Application, Service
3.
3. Sends an CoAP-response (including the payload) message to address it received the request from

Figure 2.7: An example of CoAP-request and response. Note the similarities with the HTTP.

The CoAP offers a more limited selection of request methods, since according to Shelby et. al. [72] it only allows the use of GET, POST, PUT and DELETE methods. They also claim that the CoAP allows easy integration with previously mentioned HTTP. This is because they both share the REST-model (Representational state transfer). The documentation of the CoAP even claims [39], that using cross-protocol proxies, it is possible to send an HTTP-GET request and get a CoAP-originated response without even knowing about the transformation. Zanella et. al. [88] also offer this possibility as a part of an interoperability proposition to a heterogeneous Smart City architecture.

The strength of CoAP is in its ability to communicate entirely in binary format. As already stated, header fields are in binary form, but also the payload of the message can be in binary. CoAP is entirely payload agnostic. Thus it can be used with JSON, XML or with binary encoded CBOR [10] making messages very small and efficient.

As a significant difference with HTTP, the CoAP has a feature that allows it to subscribe for new content on a specific URI. Thangavel et. al. [78] state, that whenever new content is available on the subscribed URI, all subscribers are noted about this. Then each subscriber makes a GET-request and receives the content. This architecture is according to Thangavel et. al. [78] called observe/Publish-Subscribe.

2.2.3 MQTT

The MQTT (MQ Telemetry Transport) [76] also uses the TCP/IP-stack, but unlike the direct end-to-end use of HTTP, MQTT uses a topic-based publish-subscribe messaging pattern. Whereas CoAP had pub/sub-paradigm like features, the MQTT is designed to provide Publish-Subscribe message delivery [76]. Thangavel et. al. [78] state, that the MQTT is also intended to be suitable for devices with limited resources. Ahlgren et. al. [1] state, that MQTT is very useful in the IoT domain due to its small need of memory space and processing needs.

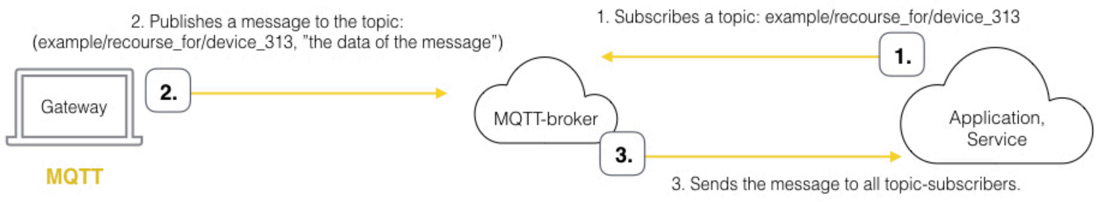

2. Publishes a message to the topic:
(example/recourse_for/device_313, "the data of the message")
1. Subscribes a topic: example/recourse_for/device_313
1.
2.
Gateway
MQTT-broker
Application, Service
3.
MQTT
3. Sends the message to all topic-subscribers.

Figure 2.8: An example of MQTT publish-subscribe communication over the MQTT-broker.

In the center of any MQTT message exchange is the message broker [76]. The broker acts as a server for messages exchange. Each message is always published on a topic. Thus MQTT can support one-to-many and many-to-one message exchange [76]. A client can subscribe topics from the broker, and if the broker receives any new messages to that specific topic, it transmits them to the topic subscriber.

Each topic has a specific name which is a UTF-8 encoded String [76]. Each topic can be separated into several levels by using the forward slash "/". This design principle allows the use of so-called wildcards. As an example from the MQTT documentation an individual tennis player could have a topic: *"sport/tennis/player1"*. Subscribers would probably receive messages relating to that specific player1. Since MQTT allows wildcards ("#"), subscribing a topic *"sport/tennis/#"*, might return messages relating to all players within that system. This naturally depends on the design of the topic structure but eases the one-to-many and many-to-one pub/sub usage.

Thangavel et. al. [78] state, that there are three QoS (Quality of Service) levels for the reliability of message delivery in MQTT. This allows MQTT to suit various designs, wherein some the simplicity of the system is the highest priority, and in some, the reliability of message exchange is the critical design principal. Thus selecting proper QoS-level can cut down unnecessary overhead of the system.

Security The MQTT can use the standard SSL encryption over the network, and additional encryptions may be also applied to the application layer. Security is also considered when clients register to topics since the broker can be set to expect a proper password for allowing registration.

2.3 Data format interoperability

After successfully connecting to a data source on the network layer, and after using shared message protocol, the third interoperability challenge to solve is the data annotation [18]. From the viewpoint of interoperability, the essential problem is to recognize and use the right structure. If this data is the organizations own, it is likely, that the data format is known. And like Bianchini et. al. [8] state, third party API-providers usually declare the used data format. Various schemas, formats, and standards exist on how to present the data. XML, JSON, HTML, and others, are some of the popular data formats, but these are just ways to annotate the data.

There are vast varieties on how data is represented, thus there also exist many data formats. This thesis focuses on the common data formats used on the Internet and in the Data-Driven domain. But as a curiosity, it is easy to see the need for different data formats used by the worlds most massive scientific experiment, the Large Hadron Collider, in CERN [12] and a single selfie in JPEG-format [15].

After successfully sharing a message (by using whatever network or message layer protocols), the next interoperability challenge is to share the understanding of the encoding, syntax, and after that the semantics of the message. As an analogy from the human domain, imagine that you receive a letter. At the first glimpse you see familiar letters, alphabet you recognize (Encoding interoperability), put in order

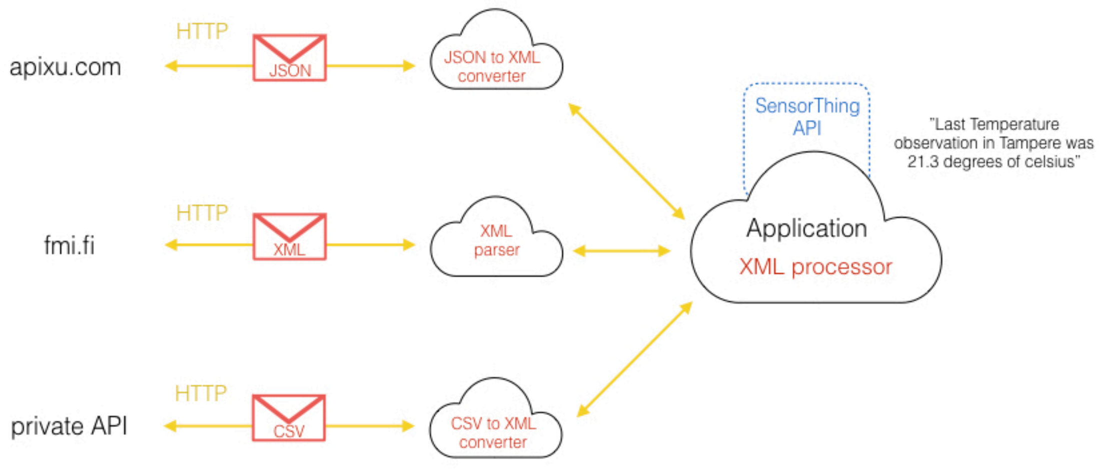


Figure 2.9: An example of an weather application that relies upon third party data. If data provider changes data format, the application developers need to fix the interoperability problem.

so that you can see words and sentences (Data format interoperability). After this, you can read the letter and understand its content (Semantic interoperability).

In the domain of IoT-system development, the data format interoperability is not really an issue. When developing new systems, it would just be bad design, if entities would communicate using non-interoperable data formats.

2.3.1 Encoding interoperability

Essentially all the data in the computer domain is just a composition of zeros and ones. When letters or numbers are presented, they need some encoding. The Unicode standard [79], is a set of different encoding forms, out of which most famous is the UTF-8. They are designed to global standards, which should replace the common ASCII [35] and other formats. In 2010 Google [33] did a study, where they discovered, that the UTF-8 Unicode encoding was used in over half of the web's content, and its portion was massively increasing. But others still exist like ISO-8859 family[44].

The used encoding is usually announced at some point of used protocol, like the response message in HTTP, which has the Content type-field, where the syntax and the encoding of the payload are announced. In JSON [46] a String is defined to consist of any number of Unicode encoded letters. In a XML-document [81], there is (optional, but common) declaration set, that defines the used XML version and the used character encoding.

The encoding interoperability is not a problem if the content is written in English since pretty much all standard encoding forms cover the English alphabet thoroughly. But some issues may arise, if the material has essential special characters or is written in, say, Finnish. This is because the ASCII doesn't have the å, ä or the ö letters, and replaces them with gibberish. This won't affect the data format interoperability but does possibly effect the semantic interoperability. A database might store Finnish word "lopputyö" (thesis), but with the letter ö miscoded. When querying for thesis workers from the database, this might be an issue. But as mentioned, if the declaration of the encoding matches the used one, there is hardly such an issue.

2.3.2 XML

The XML [81] is commonly used data format. The IANA Media Type [29] for XML is *application/xml*. According to the W3C's XML-specifications, every XML document is made out of Unicode characters. XML was designed to allow documents to include metadata about the content, so that it's both human and machine readable.

This is achieved by using markup, which has includes information about the content of an element, which has the actual data of the document. In addition to markup, which gives the document a specific structure, additional attributes can be included in the markups.

In the following imaginary and extremely simplified example, the first line is the declaration for the XML document. This declaration line also has an attribute of *encoding="UTF-8"*, which is optional, but often useful addition since it solves previously mentioned encoding interoperability problem. XML 1.1 is by no means UTF-8 restricted, but according to XML 1.1 specifications [81], UTF-8 is the only encoding that needs to be interpreted by all XML processors.

```
<?xml version="1.1" encoding="UTF-8"?>
<observation date="2017-09-17">
  <location>Tampere</location>
  <weather>sunny</weather>
  <weather>windy</weather>
  <temperature>21.3</temperature>
</observation>
```

The second line has an element called *observation* that also has an attribute of *date*, which has a simple date as a value. This is one of the most useful features of XML since the XML processor can look for only those observations (markup defines this), that have the desired date (the attribute for that element). Within the observation element, there are both the location and the weather elements. Thus, when receiving the observation element, one also obtains all those elements, that it contains. This versatile use for structure in the XML document is a highly useful feature.

2.3.3 JSON

JSON (JavaScript Object Notation) [46], is a light-weight data format very suitable for to data interchange. The IANA Media Type [29] for JSON is *application/json*. Like the XML the JSON is designed to be easily read and created by both humans and machines. According to JSON specifications [46], JSON is completely programming language agnostic, because it is essentially a text format. But it has many shared properties with popular programming languages since two of JSON's most common basic constructions are the object (a key-value pair) and the array (ordered collection or a list).

The following example has the same nesting structure as previously presented XML. Firstly the example is an JSON-object, that has the keys *observationDate* and *observation*. The *observation* is a JSON object, that is nesting an array of objects, out of which the later (*weather*) contains an array as a value.

```
{
"observationDate": "2017-09-17",
"observation":
   [
      {"location": "Tampere"},
      {"weather":
          [
             "sunny", "windy"
          ]
       },
       {"temperature": 21.3}
```

```
    ]
}
```

When comparing with the XML, both of the examples offer the same functionalities. Content can be searched by the date, and the same values are stored in both. In JSON an additional object of *observation* was needed to contain the data. But in XML the list of weather properties had two elements that had the same markup due to the lack of list or array functionality. This creates performance differences, but as Maeda [51] states, the used programming language and the selected libraries are the primary serialization performance effects. As Maeda [51] states, there is no single best solution for serialization, since the performance always depends on context.

2.3.4 CSV

CSV (Comma-Separated Values) [71] is also a popular data format, which is annotated as *text/csv* by the IANA Media Types [29]. Shafranovich [71] claims, that CSV existed, and was broadly used for data interchange among spreadsheet software, long before it was officially documented in 2005. A CSV-document essentially consists of lines which end with a combination of characters CR and LF in ASCII encoding. On every line, there exists a record that is a composition of values separated by a comma.

```
Carriage Return = CR = 0x0D = \r
Linefeed = LF = 0x0A = \n
Comma = 0x2C = ,
```

In the following example, there is one possibility to present the same instance as seen in the XML and the JSON Subsections. The first line is referred as the header line [71], which is optional but in many cases useful since it is the only official method for including metadata in the document. All lines, including the header line, should have the same amount of values separated by a comma.

```
date,location,weather,temperature\r\n
2017-09-17,Tampere,sunny,21.3\r\n
2017-09-17,Tampere,windy,21.3
```

But as Repici [67] claim, the CSV has many drawbacks. These are mainly due to the historical usage of CSV since it has originated from Windows Excel software, which originally operated on the Microsoft Windows operating system. Since there are differences among operating systems on how to present the linebreak, there might be issues converting CSV-documents from one system to another. Also the fact, that software using a CSV-document might make different assumptions about the used encoding, could create interoperability challenges. This is especially a challenge here in the Nordics, due to the need to use the UTF-8 encoding (see Subsection 2.3.1). Despite these problems, the CSV is a common format for data interchange [71].

2.4 Semantic interoperability

If we follow the previously presented analog of a letter, after being able to read it, we face the highest level of interoperability challenge, what does the text on the letter mean? We can approach this dilemma with an example from the human domain. Imagine that you want to know what are the circumstances in a newly built house, and write an email asking for them. Later you get a response saying that the conditions are 20,5. All the previously presented interoperability challenges are fully solved (since you got the email and you were able to read it), but you gained very little information, and by no means, your initial question was answered.

Murdock et. al. [57] claim, that the first enabler for solving the previous problem, is shared metadata. Essentially metadata is data about the data. Figure 3.2 opens this multilevel nature of metadata more deeply. Since the nested nature of metadata allows multiple levels of metadata to be added to the actual data, there is no limit on what can be described or not. Only limitations are practical, is it reasonable to stack meanings so far, that everything is eventually described as an object or a thing?

According to Ushold et. al. [55], semantic interoperability essentially means the exchange of information in a meaningful way. Murdock et. al. [57] adds upon that, when they state, that semantic interoperability is achieved if two or more systems share data, and more importantly, share the meaning of that specific data. Murdock et. al. [57] also state, that semantic interoperability among IoT-systems would provide much higher value. In a financial sense, this means higher profits.

Murdock et. al. [57] present the shared metadata as a three layered model:

- *No metadata*, Hardly reusable
- *Locally defined metadata*, Can create added value within that domain

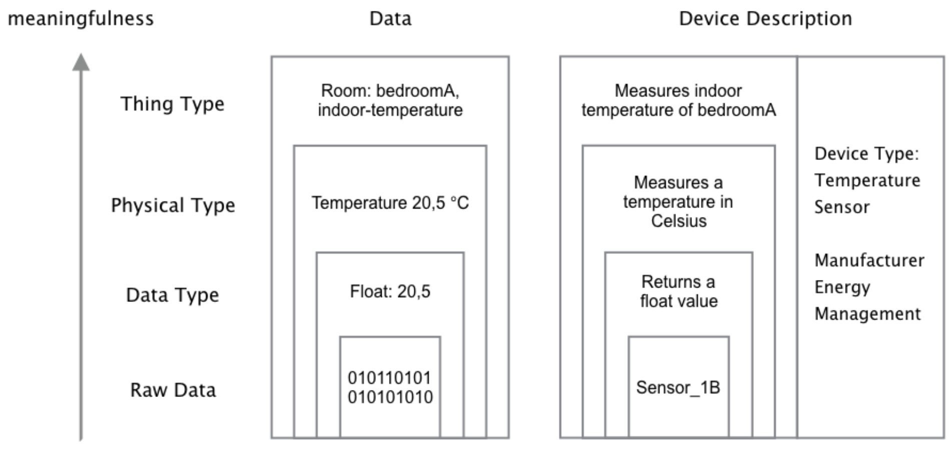


Figure 2.10: Meaningfulness of the data increases with more metadata, model by Murdock et. al. [57].

- *Metadata based on shared vocabularies*, Very reusable, could add great value

Building on the principle of the previous model, Murdock et. al. [57] state that taxonomies, ontologies, and different standard families are representations of extended shared vocabularies. As a side note, representing valuable data in a reusable way does not necessarily mean allowing the free use of it. The owners of data still decide who to share the data to. But if it is business wise to share that data (and maybe receive some money in exchange), sharing it in defined ontology or standard would enable more potential customers to use it.

There are various possibilities to use ontologies or standards, out of which Desai et. al. [18] mention the OGC Sensor Web Enablement (SWE) [60], Semantic Sensor Network (SSN) [86] and the Semantic Sensor Observation Service (SemSOS) [38]. As Desai et. al. [18] state, these technologies solve the semantic interoperability within that specific domain, a vertical silo as they state, but lack interoperability among other ontologies. But creating interoperability among well described (ontologically) datasets is far more easy, than harmonizing dataset without any metadata what so ever. Standardised metadata representations can also be used as a basis for Machine-to-Machine (M2M) communication.

Murdock et. al. [57] claim, that despite the existence of various semantical technologies, ontologies, and standards, the semantical interoperability challenge is yet to be solved. They also state that awareness of these techniques is the first step on the way to entirely semantically interoperable world.

2.4.1 Semantic Sensor Network - SSN

The Semantic Sensor Network ontology [86], or SSN in short, is an ontology developed by the W3C. The SSN is designed to be able to describe different sensors and the observations made by those sensors. Also, any observation related concepts, such as the metadata about the IoT-devices themselves can be included, thus enabling device discovery and even M2M-communication.

Georgakopoulos et. al. [30] observe the SSN ontology, and claim it to consist of ten abstractions, aka. modules. Each of those modules contributes to the overall data representation from different perspectives. Out of these perspectives Georgakopoulos et. al. [30] mention, the following:

- *IoT sensor*: is a view of what and how the sensor senses
- *observation*: is the data that the sensor produces
- *system*: offers a description of the system, to which the IoT sensor belongs to
- *feature*: is a description of what data property is being censored in the observation
- *deployment*: is a view to the systems deployment and lifetime expectancy
- *measuring capability*: provides the range for observations, but might also be the operating or the survival rates of the sensor
- *conditions*: can offer data about the condition where the sensor is, and when linked with data from the measuring capability, the possible measuring distortion could be taken care of

In the SSN ontology, there are modules, which consist of classes which can have properties [86]. These modules and classes together make the SSN ontology functionality, and the properties offer metadata of them. For example, SensingDevice has a class Sensor, which implements Sensing, Property, SensorInput, and MeasurementCapability classes. Some of these have subclasses, and properties like the Sensing class has a Process, which is responsible for any input or output to the Sensor itself. By using these components composition, the pre-mentioned views to the actual sensing are possible.

To enable the M2M-connectivity, the SSN is decoded in OWL (Web Ontology Language) [85]. The OWL and the OWL2 [82] are publicly defined languages for defining ontologies. More especially they commonly use the W3C XML standard (see Subsection 2.3.2) or the RDF (Resource Description Framework) [83] to form documents of the semantics, that the ontology represents. Using the RDF allows the use of the SPARQL [84] query language. The SPARQL is essentially an SQL-like query language for RDF documents [84]. Thus the SSN [86] is essentially a set of documents, which is agnostic about the possible lower layers of interoperability beneath it.

The following example illustrates our previous example of temperature data presented in SSN. This example is heavily influenced by a very informative blog post by Marcus Stocker [77], about how to present observation data in SSN using RDF-triples. Firstly, the sensor and its related sensing properties are defined. What is it and what it measures:

```
TemperatureSensor rdfs:subClassOf ssn:SensingDevice
TMP36 rdfs:subClassOf TemperatureSensor
tampereTemperature rdf:type TMP36
tampereTemperature ssn:observes temperature
temperature rdf:type ssn:Property
airTemperature rdf:type ssn:FeatureOfInterest
airTemperature ssn:hasProperty temperature
```

Each Observation is unique (*temp1*), and is linked to the metadata that describes its domain. Also, the timestamp for the observation is created, and connected to the Observation.

```
temp1 rdf:type ssn:Observation
temp1 ssn:observedBy tampereTemperature
temp1 ssn:observedProperty temperature
temp1 ssn:featureOfInterest airTemperature
temp1 ssn:observationResult senso1
senso1 rdf:type ssn:SensorOutput
senso1 ssn:hasValue value1
value1 rdf:type ssn:ObservationValue
value1 dul:hasRegionDataValue "21.3"^^xsd:double

temp1 ssn:observationResultTime time1
time1 rdf:type dul:TimeInterval
```

```
time1 dul:hasRegionDataValue "2017-09-17"^^xsd:date
```

This technology stack is the core enabler for M2M-interoperability. SSN ontology describes the phenomena and the IoT-domain surrounding it. The SSN is decoded in OWL, that can be transformed into RDF. RDF allows queries using the SPARQL, so (assuming all previous interoperability challenges solved) end-to-end machines can both communicate and be context aware of each others measuring domains.

2.4.2 OGC SensorThings API

The SSN is a popular choice for ontology, but The OGC SensorThings API [59] is also one possible solution when dealing with data in the IoT or WSN domains. The OGC SensorThings API provides an open framework for interoperable sensor data over the Internet using conventional and popular Web technologies [59]. As a critical design principal for SensorThings API, the developers mention [59] that they wanted to create a lightweight method for REST-like connectivity of IoT data and applications.

Whereas the previously presented SSN was mainly a definition on how to create a standardized set of ontology suited documents, the OGC SensorThings API is also intended to support the communication architecture of the IoT-system. According to the OGC SensorThings API specification [61], the primary design purpose of the OGC SensorThings API in to offer a standardized and easy-to-use functionality for unifying IoT-communications. The SensorThingsAPI builds upon broadly used Web-technologies, as the pre-mentioned REST-model.

The standard is designed based on the REST-model, thus its a collection of requests, which have a JSON-encoded payload. Note, that the standard itself isn't bound to any specific message protocol, and while REST is more naturally used in HTTP and CoAP, the OGC SensorThings API also has an MQTT-extension [61]. The request type itself also affects the operations (POST, GET, PATCH and DELETE). Each entity defined by the standard has a unique URI [61]. Each IoT-node or a relating concept also has a unique identifier [61], which is created by the back-end server. The OGC SensorThings API itself is entirely technology agnostic. Thus the programming language or database can be selected by the developers.

To initialize a simple system, the first thing is to send the following standard defined [59] POST-message to the server. This creates the Thing, Sensor, Location, Datastream and the ObservedProperty, which are all linked relevantly to each other.

```
{
  "name": "Temperature Measuring System",
  "description": "Sensor system for monitoring temperature",
  "properties": {
    "Deployment Condition": "Locating in an open and windy spot."
  },
  "Locations": [{
    "name": "The city of Tampere",
    "description": "This is the center at the city of Tampere",
    "encodingType": "application/vnd.geo+json",
    "location": {
      "type": "Point",
      "coordinates": [61.495396, 23.775267]
    }
  }],
  "Datastreams": [{
    "name": "Tampere temperature",
    "description": "Datastream of temperature in the city of Tampere"
    "observationType":
        "http://www.opengis.net/def/observationType/
    OGC-OM/2.0/OM_Measurement",
    "unitOfMeasurement": {
      "name": "Degree Celsius",
      "symbol": "degC",
      "definition": "http://www.qudt.org/qudt/owl/1.0.0/
      unit/Instances.html#DegreeCelsius"
    },
    "ObservedProperty": {
      "name": "Area Temperature",
      "description": "The degree or intensity of heat
       present in the area",
      "definition": "http://www.qudt.org/
      qudt/owl/1.0.0/quantity/Instances.html#AreaTemperature"
    },
    "Sensor": {
      "name": "TMP36",
      "description": "TMP36 temperature sensor",
      "encodingType": "application/pdf",
      "metadata": "https://www.adafruit.com/product/165"
    }
  }]
}
```

After successfully creating the Thing and the Datastream, the IoT-system can start to send Observations to the specific Datastream. The Datastream is created by the Back-end, so the *"@iot.id"* needs to be queried first. When the *"@iot.id"* is known, the Observation can be formed as the following example states:

```
{
  "phenomenonTime": "2017-09-17",
  "resultTime" : "2017-09-17",
  "result" : 21.3,
  "Datastream":{"@iot.id":313}
}
```

As seen, the OGC SensorThingAPI [59] is alternative to the SSN, and they are not interoperable unless an interpreter is used. On the other hand, they both are machine-readable, so to create that interpreter would be possible. Nevertheless, this is an interoperability challenge, that needs to be recognized.

3 Previous interoperability solutions

As a recap from Chapter 2, the interoperability can be seen as a challenge on multiple layers. This stack of layers is omnipresent on our daily lives, much of the time without us even realizing it. There are various solutions about how to solve these interoperability challenges. A comparison of interoperability solutions presented in this thesis is presented in the Table 3.1. In the following sections, we take a closer look, at how these solutions provide interoperability. Many of them solve the interoperability challenges on multiple layers.

Table 3.1: Comparison of previous solutions

Solution	Network	Message protocol	Data format	Semantic
Zhu et. al. [90] - IoT-Gateway	x			
Jin et. al. [45] - WiZi-Cloud	x			
Kruger et. al. [48] - IoT-Gateway	x	x		
Castellani et. al. [11] - Proxy		x		
Bandyopadhyay et. al. [6] - Proxy		x		
Belli et. al. [7] - Message Stream		x	x	
Desai et. al. [18] - SGS		x	x	x
Rozik et. al. [69] - Sense Egypt			x	

x = Interoparable

3.1 Network layer interoperability solutions

Desai et. al. [18] state, that the interoperability challenge among the vertical IoT-silos is a challenge created by the various compositions of hardware and software. Solving the interoperability challenge among radio-technologies requires the use of proper hardware and software. In the following subsections, there are previous solutions, where Network layer interoperability challenge is solved.

3.1.1 IoT Gateway - Bridging radio network to the Internet

Zhu et. al. [90] state, that IoT Gateway can provide interoperability between sensor networks and the Internet. They argue that for an IoT-gateway there are the following three system requirements:

1. Data Forwarding: The core functionality of an IoT-gateway is to receive and forward data from both the Internet and the sensor network. This means forwarding data seamlessly from one network to other.

2. Protocol Conversion: Zhu et. al. [90] claim, that the Internet's network traffic is done mainly using the TCP/IP protocol while the IEEE 802.15.4 based ZigBee is popular radio protocol for sensor networks. The IoT-Gateway is responsible for transferring correct data packets from radio operating sensor network to the correct Internet entity, and vice versa.

3. Management and Control: Zhu et. al. [90] state, that the gateway should offer management and possibly also control of the sensor nodes.

To demonstrate pre-mentioned, Zhu et. al. [90] built an IoT-Gateway using very simple hardware. In their model, they used a very simple computer (ARM9 Samsung S3C2440 processor with 400MHz CPU, 64M both of Flash and SDRAM). At the time of writing this setup could be seen as very constrained both on memory and on processing power. The gateway also included a GPRS-module, that was used for communication with Internet entities. ZigBee radio-module (MSP430, CC2420) was also included, and that was responsible for communications with the sensor network entities. The mainboard and both the ZigBee-, and the GPRS-module was connected by using a serial connection.

Zhu et. al. [90] also presents the workflow of the main program running in the IoT-Gateway. The primary responsibility of the gateway is to listen to the serial ports. If something is received through a serial port, this event creates an interruption, and gateways main program determines from which serial port the interruption was created. After deciding this, the program passes the message received through the serial port to the proper software module. If the message is received from the ZigBee-module, the Protocol conversion module is the next destination. After that, the Messaging platform interaction module posts the message to the TCP Server. If the message is received from the GPRS (or the Ethernet) port, the Command analysis module determines, what actions need to take place. Following this,

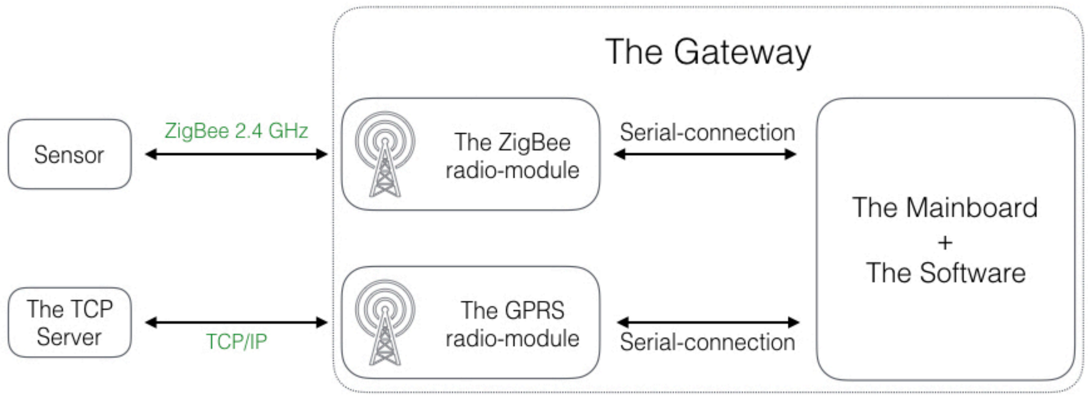


Figure 3.1: The general architecture of the IoT-gateway by Zhu et. al. [90]

the command distribution software module forms proper headers for the ZigBee-messages and sends them through the serial to the ZigBee-radio module.

3.1.2 Multi-radio IoT Gateway

Jin et. al. [45] present the WiZi-Cloud, which is a dual-radio access point to the Internet. It has both a WiFi and a ZigBee radio, and software for handling the WiFi-packet transfers, but also converting IP-packets to suit the ZigBee-network. As mentioned in Section 2.1, network layer interoperability requires the use of proper hardware. Jin et. al. [45] state, that their purpose was to offer very low power consumption data link alternative to the WiFi link. In WiZi-Cloud, Jin et. al. [45] use two different setups to enable both the use of WiFi and the ZigBee radios:

- The Linksys WRT54GL WiFi router with the TI CC2530 ZigBee SoC connected by using the UART interface.
- The Planex Wireless USB router MZK-W04NU with the TI CC2530 ZigBee SoC connected by using a USB-dongle.

These setups are essentially ZigBee-extended WiFi-home routers. But also custom software is needed to provide interoperability among these networking radios. Out of that functionality of the WiZi-Cloud, Jin et. al. [45] present the following system framework:

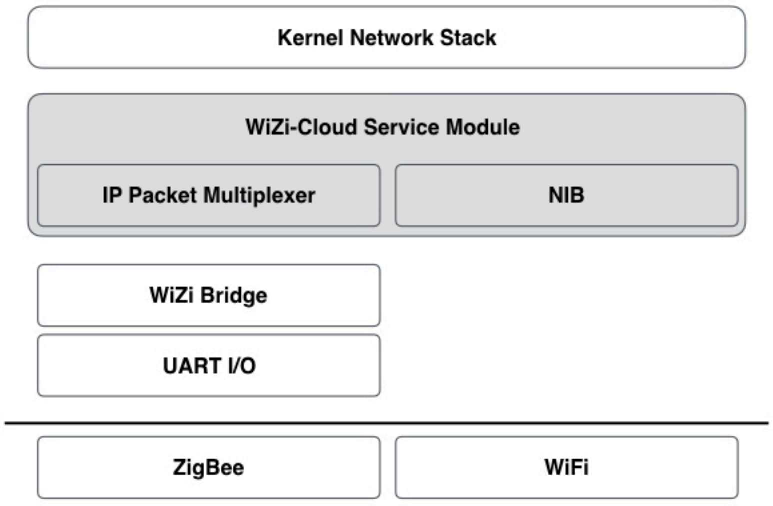


Figure 3.2: The general framework for WiZi-Cloud by Jin et. al. [45]

Jin et. al. [45] state, that the WiZi-Cloud Service Module is responsible for processing and forwarding the messages to the correct network interface (either the Internet, ZigBee or the WiFi). The service module extends the IP-based routing of the WiFi-network to also suit the ZigBee through the WiZi Bridge-component. The IP-address of the message determines whether the message should be forwarded towards the WiFi or the ZigBee radio-module. This naturally means that the Gateway needs to keep track of the nodes in the ZigBee-network and to transform data traffic from and to WiZi Bridge according to the IP-routing used in WiFi. The WiZi Bridge module is responsible for IP-packet fragmentation to suit the ZigBee frame, which is smaller in size. The UART/IO module securely transmits/reads the proper data packet from the UART, which is according to the Jin et. al. [45] a simple bit stream. Finally, the ZigBee modem provides a data link, which is used to the radio-transmission among ZigBee-nodes.

3.1.3 Multi-radio IoT-Gateway from off-the-shelf components

The multi-radio IoT-Gateway by Jin et. al. [45] provided seamless integration with ZigBee and WiFi networks. It, however, required a lot of custom-made software as described earlier. Kruger et. al. [48] state that rapid IoT-Gateway development can be done using what they refer to as the off-the-shelf components. By this, they mean a set of both hardware and software, like the Raspberry Pi computer and open source software like different Linux originated operating systems or network management software. The hardware consist of the Raspberry Pi (with Linux Kernel), STM32W108CC ZigBee-module (with ContikiOS) and the TP-Link Wireless WiFi-dongle.

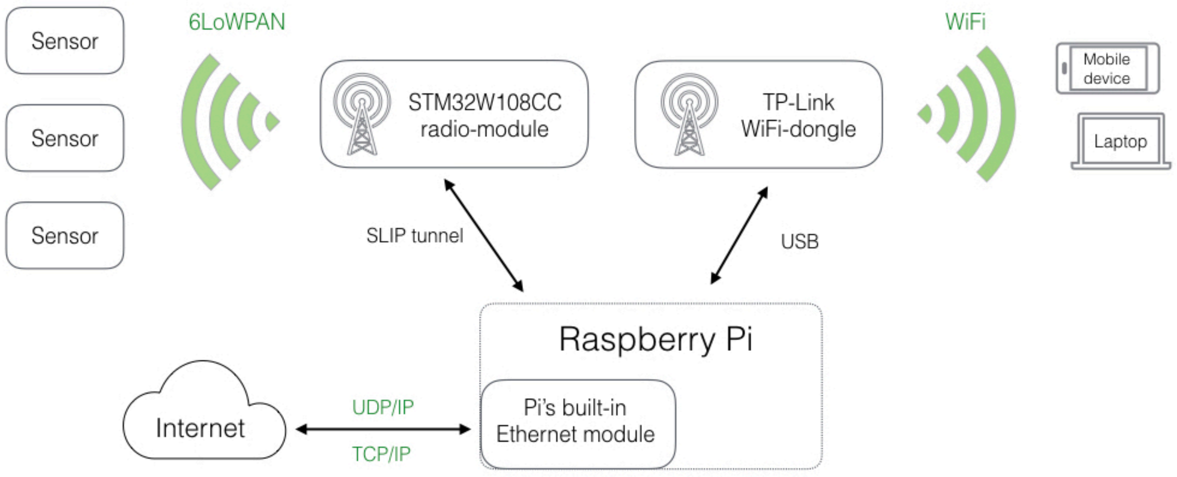


Figure 3.3: The IoT-gateway by Kruger et. al. [48]

Kruger et. al. [48] also built an IoT-Gateway, that enabled interoperability among WiFi and ZigBee radio-networks, and ensured their interoperability towards the Internet. This is accomplished by using IP-connectivity on both the WiFi and on the 6LoWPAN mesh networks. The gateway solution uses IPv6 addressing. This creates problems since most of the Internet traffic is done using the IPv4 addressing. Kruger et. al. [48] claim that IPv6 packets can be transported over the IPv4 network by using tools like the 6to4 tunnels.

According to Kruger et. al. [48], there is also need for fragmenting larger data packets used in IPv6 communication, to suit the smaller size of the 6LoWPAN. Because the Gateway operates on Linux, Kruger et. al. [48] propose a set of network tools for network and gateway management. Also the fact, that gateway can always be reached by using secure SSH-connection, eases management tasks. Kruger et. al. [48] mention that the possibility to install, update and remove software from the gateway on the fly provides confidence that the gateway can be managed even after the install and setup process.

3.2 Message protocol interoperability solutions

Message protocols like HTTP and CoAP have differences that create the interoperability challenge between them. As stated earlier in Section 2.2 the interoperability challenge among messaging protocols is primarily a software challenge. The following section presents some possibilities how to overcome this problem.

3.2.1 Using proxy for protocol conversion

The Multi-radio IoT-Gateway presented by Kruger et. al. [48] also had a feature, that provided interoperability among message protocols. They used off-the-shelf software components to host both a CoAP-proxy. As mentioned earlier (see Subsection 2.2.2), the CoAP is designed to be easily transformed between HTTP messages. They both can use the REST-design model. Kruger et. al. [48] demonstrate this CoAP-feature by presenting the CoAP-proxy. Kruger et. al. [48] state that the proxy is written in Java, but they don't describe it in more detail. But they do state, that the CoAP-proxy has the following responsibilities:

- Translate RESTful HTTP commands to CoAP
- Translate CoAP from IPv4 network to suite CoAP in IPv6 network

Castellani et. al. [11] study the HTTP to CoAP mapping in more detail. They state, that the intermediary between the two different protocol entities is called cross protocol proxy (cross proxy in short). And as a response to the silo-like interoperability challenge presented by Desai et. al. [18], Castellani et. al. [11] argue, that the cross proxy is interoperability enabler.

The HTTP and the CoAP use their own unique Uniform Resource Identifier (URI) plans (see Subsections 2.2.1 and 2.2.2 for more detail). Castellani et. al. [11] present two possibilities to URI mapping techniques between protocols.

- Homogeneous Cross URI: The same resource is named equally in both URI's. The CoaP *coap://server.net/weather/tampere* and equally within the HTTP-domain *http://server.net/weather/tampere*
- Another possibility would be to embed the CoAP-URI within the HTTP-resource: *http://server.net/coap/server.net/weather/tampere* is interpreted by the HTTP server into *coap://server.net/weather/tampere*

Castellani et. al. [11] also argue, that the gateway providing the proxying in IoT-domain is responsible for ensuring that constrained servers are not over-flood. The unconstrained HTTP-based Internet entities cannot expect to interact with the constrained CoAP-devices (battery powered, low processing power) as with other unconstrained entities. Castellani et. al. [11] claim, that the cross proxy is thus responsible for congestion control. They also state that cross proxy can be used to handle the mapping between IPv4 and the IPv6 networks. This is not a necessity, but they argue, that it is commonly needed since most of the HTTP-traffic is done as IPv4.

Castellani et. al. [11] also observe two existing solutions for proxying. The WebThings [47] is an open-source toolkit that consists of many application layer software components. WebThings is not a complete solution but has many modules, that can help to gain interoperability among heterogeneous message protocols. The documentation of the WebThings [47] state that it is designed to be a proxy server for connecting WSN-nodes to Internet using CoAP in REST-like communication. But since CoAP is a message protocol, it can also be used solely on the Internet just like Castellani et. al. [11].

As an another example Castellani et. al. [11] mention the Squid-project [75]. The Squid is essentially an HTTP-proxy server, but Castellani et. al. [11] state, that it can be expanded with an HTTP-CoAP-module to create a cross proxy. They also claim that the easy to use and efficient caching support on the Squid is also a useful feature in the cross proxy. The Squid can also handle the URI addressing needs of proxy service, thus making it suitable for a cross proxy. These are only two examples of available open-source software, but Castellani et. al. [11] argue that their existence eases the development of interoperable, heterogeneous message protocols.

Bandyopadhyay et. al. [6] state, that the similarities of the HTTP and CoAP in RESTful architecture eases their connectivity through proxying. They also observe the possibility to use CoAP in pub/sub like GET-observe to gain interoperability with popular MQTT pub/sub-protocol. They state that despite similarities between CoAP-GET-observe and the MQTT there still is a need for Server to provide logic between the two entities. Figure 3.4 demonstrates this functionality between MQTT and CoAP device. Since the MQTT operates in broker-based fashion, the MQTT-broker won't understand the CoAP-GET-observe unless there is an additional software component (Connection server in Figure 3.4) involved.

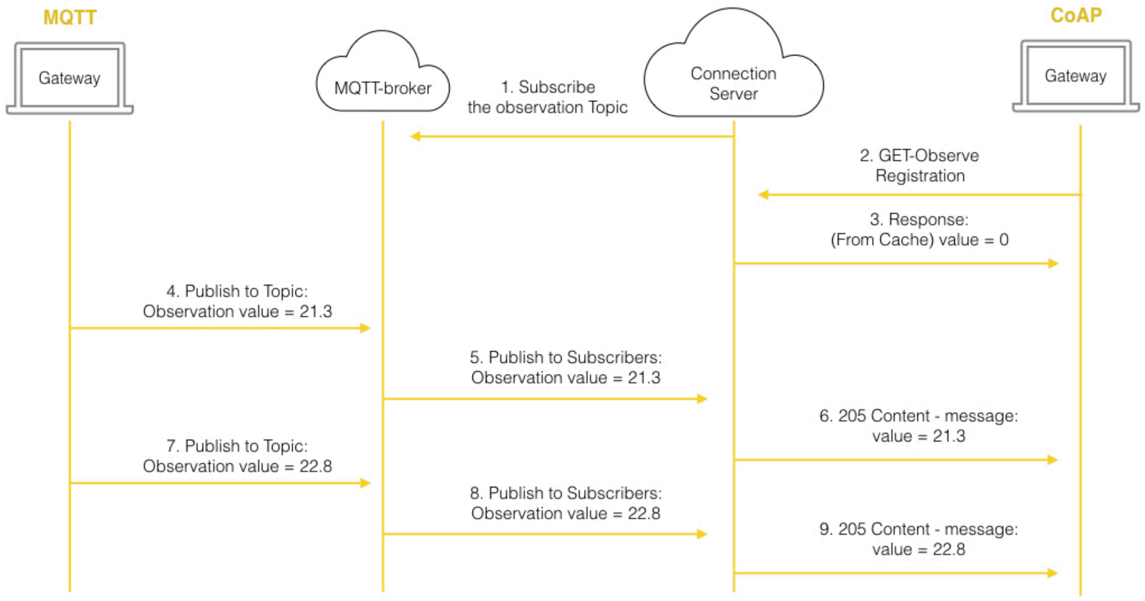
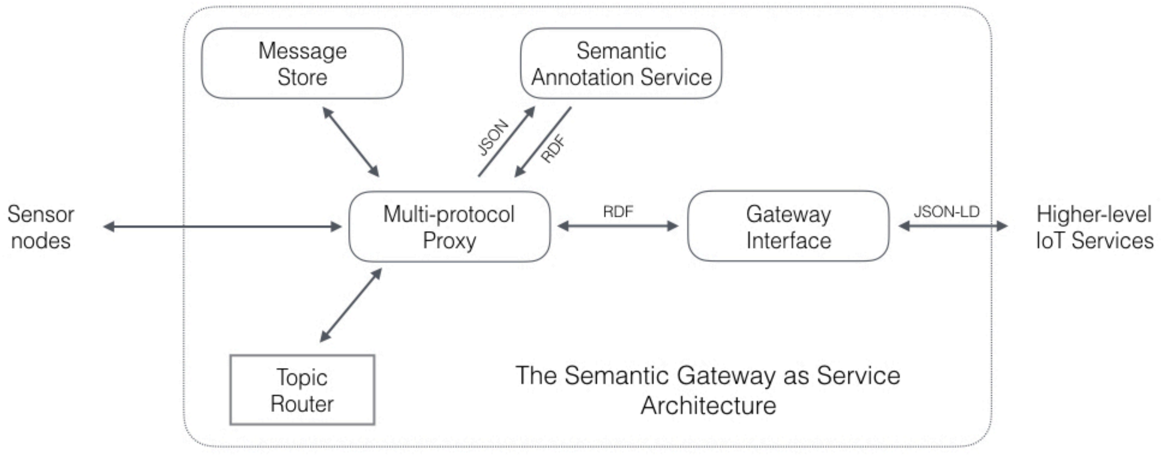

MQTT
Gateway
MQTT-broker
1. Subscribe the observation Topic
Connection Server
CoAP
Gateway
2. GET-Observe Registration
3. Response: (From Cache) value = 0
4. Publish to Topic: Observation value = 21.3
5. Publish to Subscribers: Observation value = 21.3
6. 205 Content - message: value = 21.3
7. Publish to Topic: Observation value = 22.8
8. Publish to Subscribers: Observation value = 22.8
9. 205 Content - message: value = 22.8

Figure 3.4: An example by Bandyopadhyay et. al. [6], of interoperability among MQTT and the CoAP Gateways using Server.

3.2.2 Multi-protocol proxy

As seen in Subsection 3.2.1, achieving interoperability among heterogeneous message protocol can be achieved by proxying. Desai et. al. [18] present the Semantic Gateway as Service (SGS), that would enable interoperability on message protocol, data format and also on semantic layers. The SGS uses Multi-protocol proxy to solve the Message Protocol interoperability challenge. More specifically the proxy offers services in CoAP, MQTT and XMPP Message Protocols. Because these protocols can carry payload presented in various Data Formats, the Proxy also solves the Data Format interoperability by parsing all incoming messages into JSON or in RDF. In Figure 3.5 there is a general architecture of the SGS components. Desai et. al. [18] also present the SGS openly available as a node.js package at [17].

The components of multi-protocol proxy are presented in more detail in Figure 3.6. Desai et. al. [18] also state, that the proxy is not restricted to only CoAP, MQTT, and XMPP, but also other message protocols could be used. All data sources are connected by using message protocol specific Clients. When using the CoAP, the client interacts with the CoAP Interface, which forwards the messages to the Mes-



Message Store
Semantic Annotation Service
JSON
RDF
Sensor nodes
Multi-protocol Proxy
RDF
Gateway Interface
JSON-LD
Higher-level IoT Services
Topic Router
The Semantic Gateway as Service Architecture

Figure 3.5: Semantic Gateway as Service Architecture by Desai et. al. [18]

sage Broker. In Figure 3.6 also MQTT-broker and XMPP-servers are connected to the Message Broker. This design principle allows the simultaneous use of heterogeneous message protocols by multiple IoT-systems. For instance, the communication to an MQTT-specific IoT-system appears to happen solely on MQTT.

The multi-protocol proxy translates messages between different message protocols. To achieve this the Multiprotocol Proxy by Desai et. al. [18] uses three software components.

- *Message Store* is responsible of storing all messages until they are successfully sent to the higher level of IoT-services through the REST-interface on SGS.
- *Topic Router* is responsible for creating, maintaining and sharing knowledge of all connected IoT-devices. Each device has its unique topic and id on the Topic Router. This abstraction allows the harmonization among various message protocols. By using whatever message protocol to communicate with the Message Broker, all IoT-devices appear similarly on the Topic Router.
- *Message Broker* is responsible for proxying different message protocols so that the content of messages won't be disrupted. It is also responsible for using the Semantic Annotation Service, whenever the higher level of IoT-services request a new message.

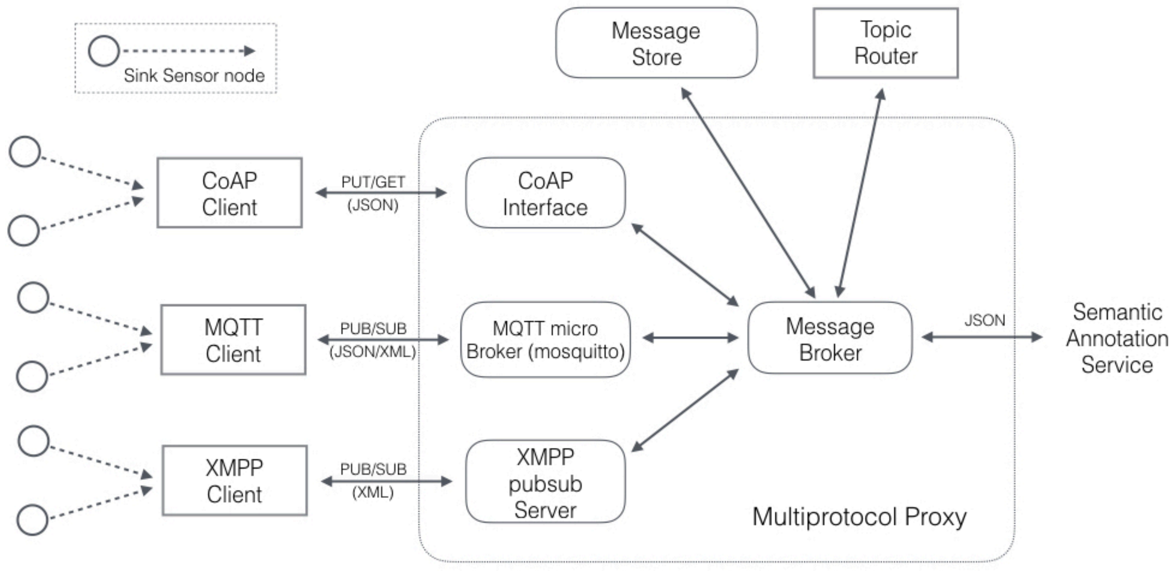


Figure 3.6: The multiprotocol proxy of SGS by Desai et. al. [18].

Desai et. al. [18] present an example of how the multi-protocol proxy operates. In Figure 3.7 there are different logical steps, which show the workflow of the message translation.

1. Message Proxy has subscribed all topics and thus receives all new messages sent by MQTT-publishers.
2. Proxy ensures that the topic is registered and transmits it to the Topic Router. If this topic is not known by the Topic Router, it creates it and gives it a distinguishing id.
3. Proxy forwards the message to the Message Store that stores it.
4. The Multi-protocol proxy waits for a request from higher level IoT-services.
5. Message Proxy receives a request from higher level IoT-services with an id attached to the request.
6. Proxy determines what topic has this id by asking it from the Topic Router.
7. By including the id, the Proxy request the recent message from the Message Store.

8. Message Store returns the original message, which it has stored.
9. Message Proxy sends it to the Data Annotation Service, wherefrom it is sent properly annotated to the requester.

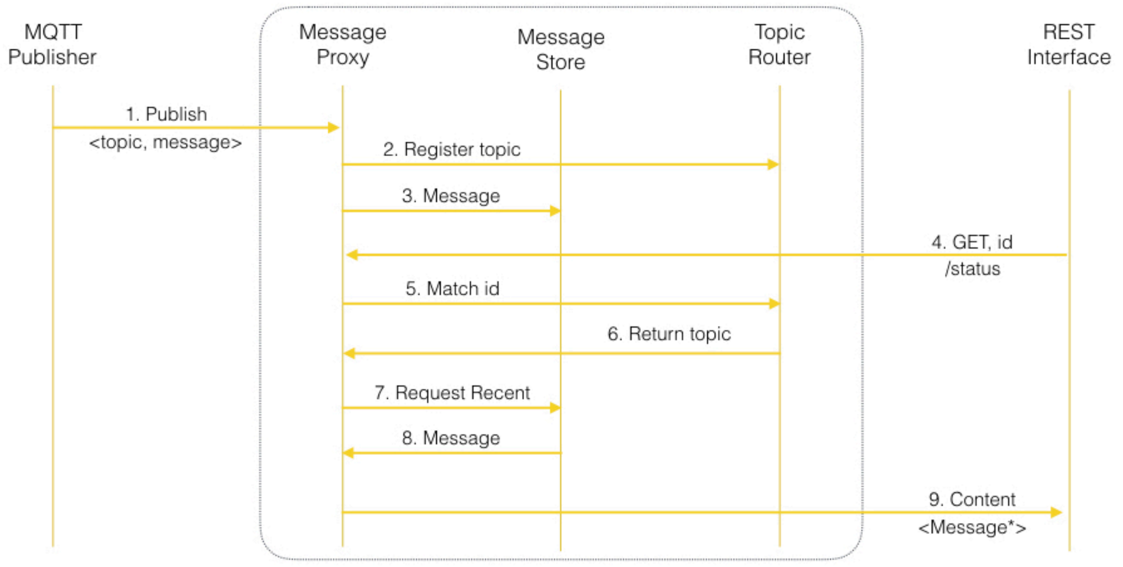


Figure 3.7: The message translation of multiprotocol proxy by Desai et. al. [18].

To achieve Data Format and Semantic interoperability, the SGS also has additional software components. These components are presented in more detail at Section 3.4.

3.2.3 Using message stream to enable interoperability

Belli et. al. [7] present an open-source architecture (see Figure 3.8) for what they refer as Big Stream data flow system. It is capable of handling data flowing into the system from multiple heterogeneous data sources using various message protocols. Belli et. al. [7] argue, that there are many possible messaging protocols (see Section 2.2) which could be used by IoT-devices or systems. Thus they include in their architecture an Acquisition module which consists of multiple connectors, all capable of handling different protocols for the incoming data stream.

By interoperability, Belli et. al. [7] mean the ability to gather data from heterogeneous data sources into single Big Data system, not the capability of providing

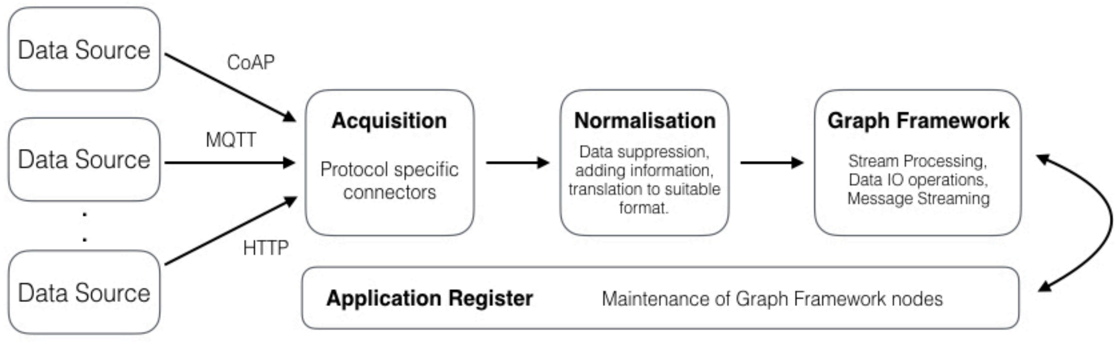

Data Source
Data Source
Data Source
CoAP
MQTT
HTTP
Acquisition
Protocol specific connectors
Normalisation
Data suppression, adding information, translation to suitable format.
Graph Framework
Stream Processing, Data IO operations, Message Streaming
Application Register
Maintenance of Graph Framework nodes

Figure 3.8: General architecture and simplified responsibilites of components in Big Stream application architecture by Belli et. al. [7]

communication among vertical IoT-silos like presented by Desai et. al. [18]. Thus the name, Big Stream. The data only streams into the system, but it does come from various message protocol sources.

Belli et. al. [7] divide the responsibilities to different components. The acquisition component is responsible for message protocol harmonization. This task is achieved by using open source software components, each running on as a separate process. For HTTP the NGINX [40], for CoAP the mjCoAP [56] and for MQTT-messages the ActiveMQ [25] are used. Each of those separate software processes listens to its data source, and simply forwards whatever content it receives to the message protocol dedicated Exchange-module in the RabbitMQ (Advanced Message Queuing Server).

The Exchange-module is merely a listener. It confirms that it has received the message, and is then responsible for placing them into the proper RabbitMQ buffered queue. RabbitMQ-queues store the messages until a separate Java-process called Normalisation receives them. Thus, once the Acquisition node receives a confirmation of the message sent, it shifts the responsibility to the Normalisation block.

The Normalisation nodes in Figure 3.9 are separate Java-processes, but they receive their input from the RabbitMQ-message system, so Belli et. al. [7] present them within the RabbitMQ. All these different Java-Processes are maintained by the Application Register, which is responsible for registering and maintaining the state of the pre-mentioned Java-processes. The Application Register keeps track of what processes are linked to each other. Thus the messaging between them can also be maintained.

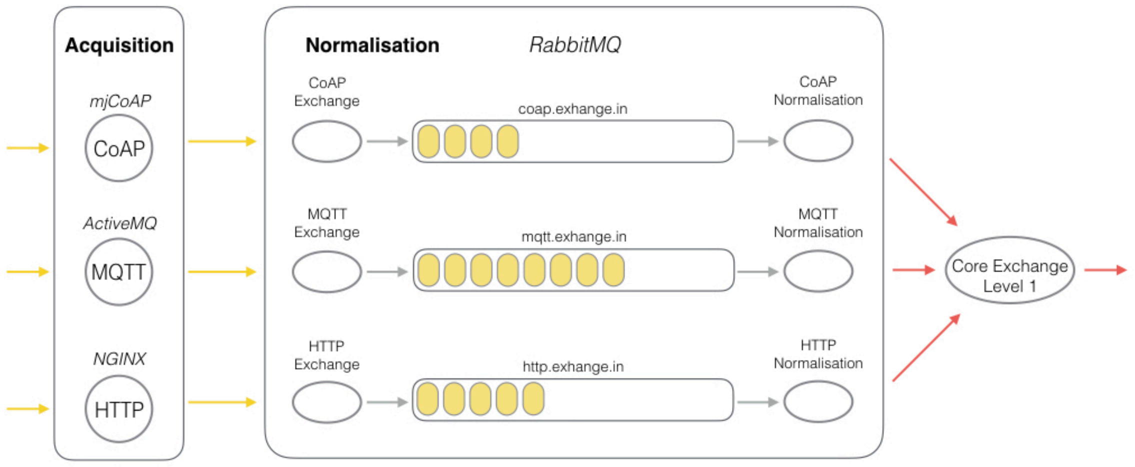


Figure 3.9: Software components of Acquisition and Normalisation modules by Belli et. al. [7]

After the Normalisation, the data flows through the Graph Frameworks entry point, which is the Core Exchange Level 1 node. After this node, the rest of the system assumes to receive messages which are also harmonized on the Data Format layer. Belli et. al. [7] argue that the Core Exchange Level 1 node has the same responsibilities as the Exchange node for each Message protocol, it just forwards the messages to the proper composition of unique Java processes. Finally, the data flow to the desired end-point. Out of these end-points, Belli et. al. mention possible Data Streams of some other Big Data processes, a Data Warehousing system, and also application layer processes. Message protocol interoperability is thus created by the combination of the Acquisition and the Normalisation modules.

3.3 Data format interoperability

Desai et. al. [18] present the Data Format and the Semantic Interoperability layers coupled as one. As mentioned earlier in Subsections 3.3 and 3.4, data can be represented in different Data Formats (XML, JSON, etc.) without it having any declarations about the metadata. But when describing the Semantics of the data (ontologies, standards, etc.), there always needs to be a description also about the used Data Format. Thus these two interoperability layers are linked together, especially when presenting actual solutions.

3.3.1 Message stream for data format interoperability

In the previous Section (see Subsection 3.2.3) Belli et. al. [7] present an example how to solve the Message Protocol interoperability challenge. Their proposition also takes a stance on how to address the challenge created by various Data Formats.

After acquiring the data into the system, Belli et. al. [7] present the Normalisation Java process, which is responsible for filtering out useless data. Out of this Belli et. al. mention the possible headers or undesired meta-data. The Normalisation block is also responsible for annotating the data with additional information, like adding a timestamp if needed. The normalization node can also be used to filter out some IoT-observations, that can be assumed to be corrupted (e.g., an Indoor temperature reading of -200 Celcius).

Belli et. al. [7] argue, that the Normalisation node is responsible for providing the data format interoperability. This is achieved by transforming all messages received from RabbitMQ-queues to JSON-format. Whether received as an XML or a CSV formatted data, the messages transported by the RabbitMQ-system are essentially Strings. To gain data format interoperability custom parsing is required. By this Belli et. al. [7] means that all messages are fragmented and finally encapsulated into JSON-format. When the Core Exchange Level 1 node receives a new message, it would be entirely agnostic about the original message protocol or the data format of the data source, unless there would be additional information implemented about these.

Thus the proposition by Belli et. al. [7] solves both the message protocol and the data format interoperability, but does not offer any interoperable solution for semantical knowledge exchange, but instead creates a semantical vertical silo of its own.

3.3.2 IoT Data Stream platform

Rozik et. al. [69] explore current IoT-platforms like the Thing Speak [53] and the Xively [50], and also propose their design Sense Egypt. These platforms offer great IoT-connectivity and are also capable of storing data and provide different data analysis tools. With IoT-Platform they mean, a cloud-based software system, which offers a set of well defined APIs, which can be used to upload data into the IoT-platform. The purpose of the platform is thus to provide harmonization, visualization, data storage, data analytics, alerts, commands and different custom messages to and from IoT-nodes [69]. By IoT-nodes Rozik et. al. [69] mean both actuators and sensors.

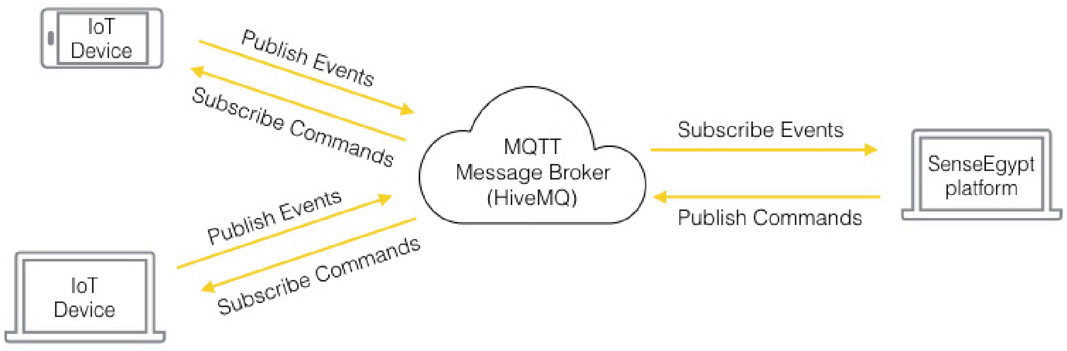


Figure 3.10: The MQTT-broker in Sense Egypt IoT-platform design by Rozik et. al. [69]

According to Rozik et. al. [69] the Thing Speak and the Xively cloud-based platforms use the HTTP-protocol for message protocol connectivity. The Sense Egypt uses the MQTT-protocol. Thus the MQTT Broker is responsible for all communications between the IoT-nodes and the Sense Egypt IoT-platform. In the Sense Egypt, the HiveMQ [16] MQTT-broker is used. The broker communicates with the IoT-nodes, and the Apache Kafka [27] message stream system within the cloud platform. Figure 3.10 illustrates the pub/sub mechanism of the Sense Egypt.

Rozik et. al. [69] state, that the MQTT broker did not provide any buffering mechanism for the messages. This is the reason why the Kafka Messaging System [27] is used. The Apache Kafka is a message stream system that operates according to the publish/subscribe paradigm. The Kafka is used to forward data received from IoT-devices to the Apache Storm [28] analytics engine. Kafka is also used for messaging between other software components of the Sense Egypt platform, and also forwarding commands to the IoT-nodes through the HiveMQ broker. The visualization component also receives its data through the HiveMQ broker.

The heterogeneity on data formats, and also somewhat the semantics, can be solved by using the Apache Storm [69]. As Figure 3.11 illustrates, the data coming from IoT-nodes through the Kafka, is first received within the Apache Storm by Kafka consumer spout. Each data source is connected to specific spout, which fetches the messages from the Kafka.

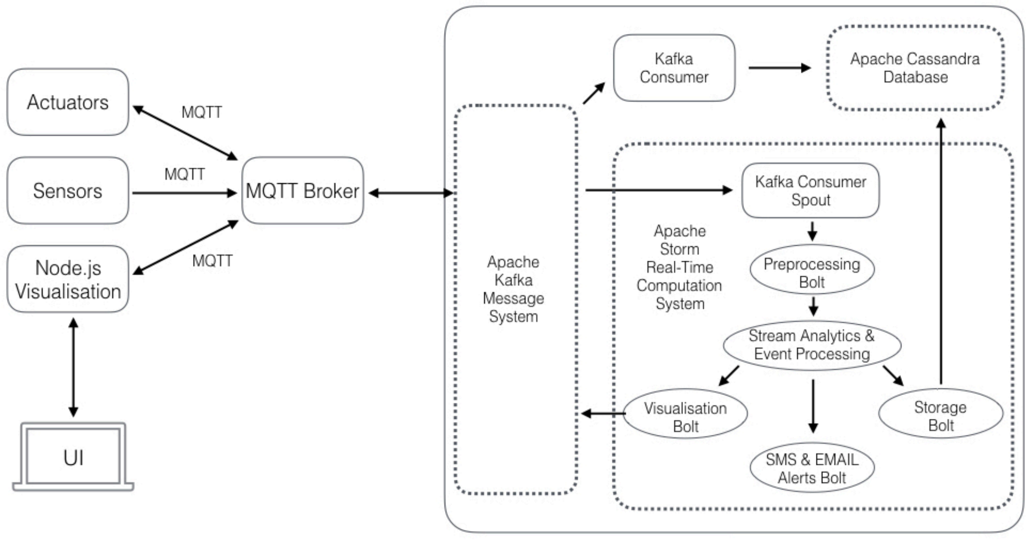


Figure 3.11: General architecture and components of the Sense Egypt IoT-platform according to Rozik et. al. [69]

Sense Egypt is entirely data format agnostic. Thus it is capable of harmonizing different heterogeneous data formats. This is achieved in the Apache Storm and in the Preprocessing bolt, which is responsible for harmonizing the IoT-data. The preprocessing begins with the data cleaning phase, where faulty sensor readings are removed, and missing values can be added. The final preprocessing task is to transform data into optimal machine learning data form. The Sense Egypt expects UTF-8 encoded Strings as input from it's MQTT Broker. To gain comparability among heterogeneous data formats, Sense Egypt uses the machine learning techniques of the Apache Storm.

Apache Storm is Java-based software that has many data classification techniques. All incoming data is dynamically transformed by the Preprocessing bolt into Serialized objects [24], tuples as Storm refers them. These tuples are Java Objects within the Storm, but once exported from the Storm, they are transformed into CSV or other data format. Thus the data format is interoperable once serialized within Storm, but finding semantics and finding features out of data is the responsibility of the following machine learning algorithms.

The Stream Analytics and Event Processing bolt use data analytics and machine learning techniques to find features from IoT-data. By features Rozik et. al. [69]

mean separate events that have a meaningful correlation with one another. Even if there is no correlation found, the Stream Analytics and Event Processing bolt attempts to classify data so that later discoveries can be made. In the design of the Sense Egypt, no topology or standard is applied to the data, but the Stream Analytics and Event Processing bolt could also be used to do that. Nevertheless, this component is where the platform attempts to gain semantic interoperability, find meaning from raw data.

After the Stream Analytics and Event Processing bolt the flows to the Storage Bolt, which is responsible for securely uploading data to the Apache Cassandra database [26]. Also, the data is forwarded towards the user interface, by publishing it on the dedicated UI message stream on the Apache Kafka. Also, the Stream Analytics and Event Processing bolt can trigger different alerts, like send an SMS or email if it finds features, which are preconfigured to do so.

3.4 Semantic interoperability

To enable Message Protocol, Data Format, and semantic interoperability, Desai et. al. [18] present the Semantic Gateway as Service (SGS). The solution for message protocol interoperability was introduced earlier in Subsection 5.3

To build upon Message Protocol interoperability, Desai et. al. [18] present the Semantic Annotation Service (SAS) software component (Figure 3.12), that they argue solves the Semantic interoperability challenge. All messages coming in from the Sensor nodes are first routed to the SAS-component.

The first software-component that receives incoming observations in the SAS is the O&M, SensorML component. Desai et. al. [18] state, that this component is designed to follow the standards for service description defined by the Open Geospatial Consortium (OGC) in the Observation and Measurement (O&M) and the Sensor Model Language (SensorML) specifications. These two provide the XML-schema that unifies all received data from sensors into a standardised format.

After this transformation, the Data Annotation component can annotate the data to suit the SSN Ontology (see Subsection 2.4.1). Desai et. al. [18] also present the possibility to use different more domain-specific ontologies if needed. After the Data Annotation component, the data is sent back to the Proxy, which can send it as forwards as JSON Linked Data (JSON-LD), which according to Desai et. al. [18] suites the REST-model better. The following components use the REST-model to

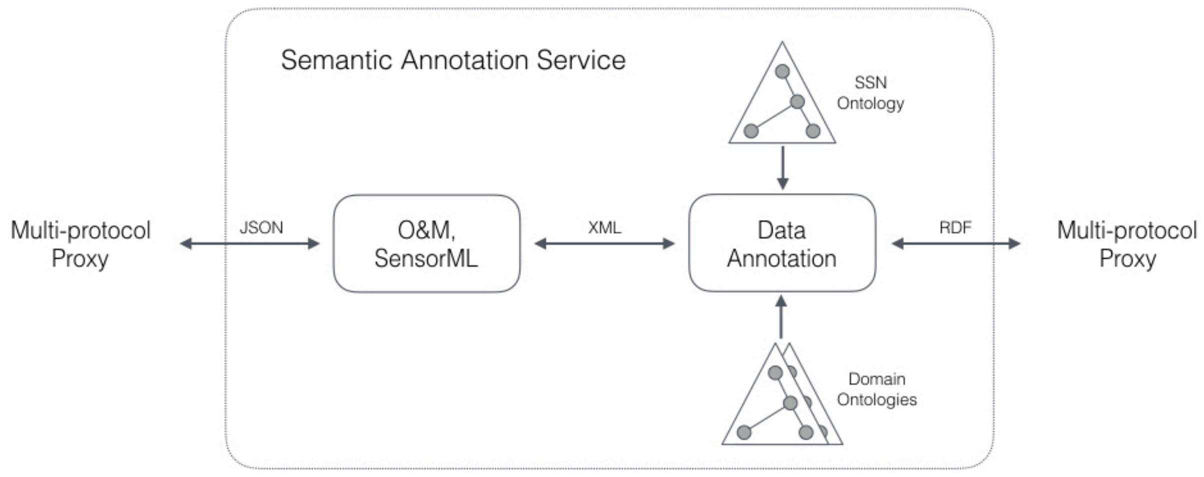


Figure 3.12: Semantic Annotation Service component by Desai et. al. [18]

serve this data to other services.

4 Case: predictive transport demand solution

The interoperability research question of this thesis was derived from a real-world software development process. The customer was a people transport company located in the city of Tampere. The customer wanted to create software, which could predict the people transport demand in the city of Tampere. If the software predicted the future demand for transport requests with reasonable accuracy, the customer would be able to optimise the use of their transport fleet. This would save money due to the more efficient use of labor. When there would be little demand, fewer drivers were waiting for customers. And during the high peaks of demand, there would be sufficient amount of vehicles on duty.

To be able to predict the future transport demand, the development team first needed to gain insight into how the human transport business works. In discussions with the customers, it became evident that multiple data sources were required. There were tens of possible events, that may affect the transport demand. Some most prominent events chosen for the POC (Proof of Concept) like the weather, which increases the demand on some occasions, like when its raining or when it's freezing cold. One event could also be derived from previous transport actions, if there is a lot of transport actions ending to a specific location, there obviously might be more future transport requests from there.

Since there was no certainty about how events changed the demand, few initial data sources were selected in collaboration with the customer. These few would act as a basis for a fully functional proof of concept software. More data sources were planned to be added later if the initial POC was approved by the customer.

The software would be a web application, that would be accessible only by the customer's staff. While the application offers a prediction of the demand, the decisions on how to deploy the vehicle fleet are still made by the customer. The application has a map view of the city of Tampere (this is the core business area of the customer). The map view is divided into a grid, where each block within the grid acts as an indicator for changes on future demand.

The future demand is calculated for each block. This grid division is not fixed but can be changed, if the customer wants to change the size or the number of blocks.

As initial design proposition, each block is either transparent, colored using green if there is increase on demand and colored using red if there is less then average of demand.

Also each of the vehicles actively on-duty are displayed on the map, so that the customers transport controller can redirect vehicles to most profitable locations. Vehicles currently carrying a passenger are colored red, and vehicles which are free are colored green. The Figure 4.1 illustrates this design.

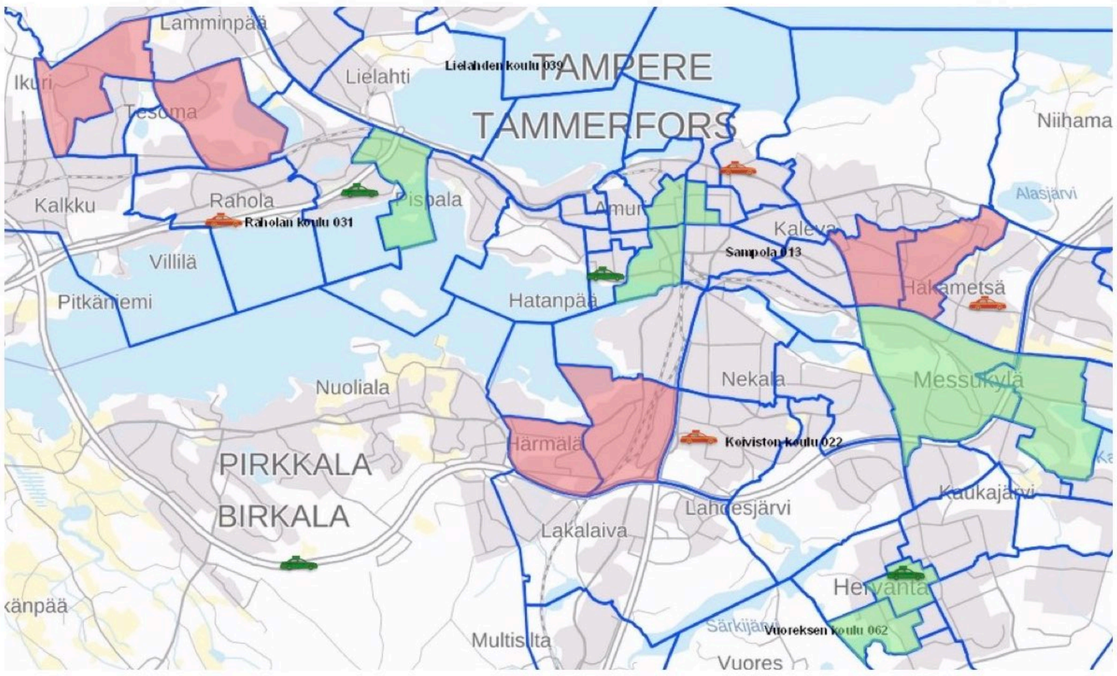


Figure 4.1: An example of the core functionalities for the application. Background map from Avoindata.fi [20].

4.1 Requirements and objectives

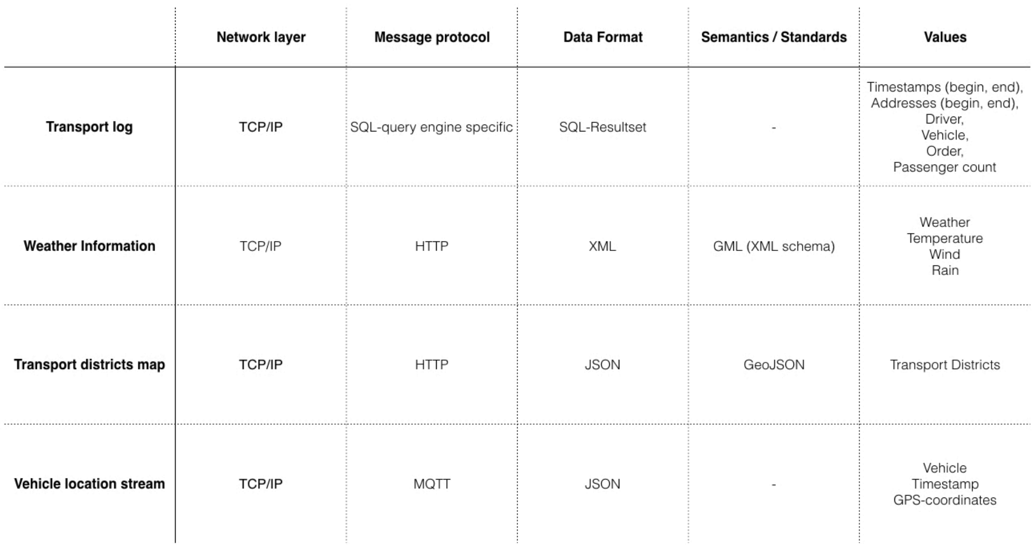
To be able to provide such functionalities for the predictive Dashboard, the design team crafted the following list of initial key-requirements. The initial objective was to build a system, that would integrate several heterogeneous data sources. Data from these sources were gathered into the system and processed using data analysis techniques. The prediction of possible changes in transport demand would be created by the data analysis. To enable data analysis, the following requirements needed to be met:

- The system needs to be able to receive data from several heterogeneous message protocols.
- Data in XML, JSON data formats needs to be handled by the system. Also the system needs to be able to query a database.
- The system needs to easily adjust to any possible changes in the message protocol layer. If the data provider changes message protocol or changes attributes of the data source (etc. the URL for HTTP-GET-request), the system needs to be able to be resilient to these changes.
- The data acquisition should be decoupled from rest of the system so, that any possible errors or changes in data sources won't affect rest of the system. If a data source stops providing data, only the data should be missing, and the rest of the system would continue to operate normally.

4.2 The interoperability challenge of the data sources

The selected data sources were very heterogeneous on many levels. The initial four data sources did have one shared property; they all used the TCP/IP on the network layer, meaning that in this case there is really no network layer interoperability challenge. There were differences in the message protocols, data formats, and semantic representations. These interoperability challenges are presented in the Figure 4.2.

- Transport log is a customer database where all transport actions and orders are stored. It is essentially a relational database, that can be accessed remotely using a remote database connection. This data source was chosen since the customer had confidence that previous transport actions affect future ones. If lots of people are transported into say a football game, lots of people will order transport when the game ends.
- Weather Information is served by the Finnish Meteorological Institute (FMI). This data source was selected due to the customer's assumptions, that there is a correlation between weather phenomena and the number of transport requests. In cold or rainy weather, people attend to take more taxi rides.



	Network layer	Message protocol	Data Format	Semantics / Standards	Values
Transport log	TCP/IP	SQL-query engine specific	SQL-Resultset	-	Timestamps (begin, end), Addresses (begin, end), Driver, Vehicle, Order, Passenger count
Weather Information	TCP/IP	HTTP	XML	GML (XML schema)	Weather Temperature Wind Rain
Transport districts map	TCP/IP	HTTP	JSON	GeoJSON	Transport Districts
Vehicle location stream	TCP/IP	MQTT	JSON	-	Vehicle Timestamp GPS-coordinates

Figure 4.2: Interoperability of the datasources and possible additional data processing

- The map of transport districts is representative of the voting districts in the city of Tampere. This was chosen as the initial data source for providing coordinate information for rendering the grid and the blocks to the map. As mentioned earlier in the requirements, this data source should be easily changed later.
- Vehicle location stream was a GPS-logging system that the customer already had in use. In the stream, each vehicle transmitted its coordinates frequently, so that their current locations could be viewed from a map. This data source was selected because it was important to point each vehicle to the most prominent locations in near real time. Essentially this means moving vehicles away from lower demand blocks to those with higher demand.

4.2.1 Network layer challenges

Due to the fact, that all data sources used the TCP/IP-stack, there really was no network layer challenges. The vehicle location stream system was also out of reach of the developing team, and they provided the GPS-location data by using the MQTT that run over the TCP/IP stack. But there naturally could appear the possibility, that some data source would change it's network layer to UDP/IP.

4.2.2 Message protocol challenges

One of the interoperability challenges was the heterogeneous message protocols. The weather information and the coordinates for the transport district grid were both served by well documented REST-APIs. The weather information was gathered from the Finnish meteorological institute, by using their open data solutions [41]. The FMI-API was used by simply making HTTP-GET queries with API-key and different attributes narrowing the query included in the URL. To receive the API-key, the developer needs to register on the FMI website. For daily weather forecast in the city of Tampere, the following HTTP-GET request was made:

```
http://data.fmi.fi/fmi-apikey/secret-unique-key/wfs?
request=getFeature&storedquery_id=fmi::forecast::
hirlam::surface::point::simple&place=tampere&
```

The same kind of functionality was also used in the transport district map. For the grid information, an open source solution was selected. The voting districts of the city of Tampere were used to enable the grid division [20]. The avoindata.fi provided a REST-API, that was accessible only after registration and logging in to the avoindata.fi web application. The information was shared by using specific HTTP-GET requests.

The customer had a vehicle location service already in use. All vehicles had multiple GPS-logging systems. One of these systems operated on message protocol layer by using the MQTT. The developing team was able to subscribe all location data from this MQTT-broker. Thus the system also needed to be able to use MQTT-pub/sub paradigm.

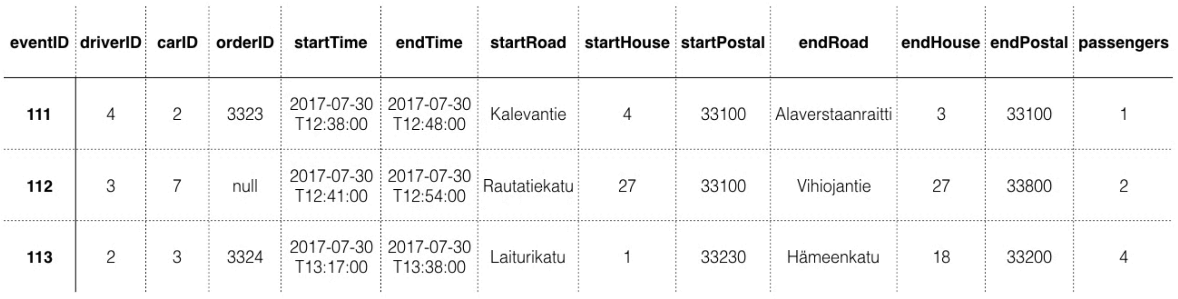
The final data source was the transport logs of the customer. These logs consisted of multiple relational database tables, into which all transport actions were logged in real time. These transport actions stored information of who was the driver, where the transport event began and where it ended. Also, timestamps and passenger counts were included. To access this data, the relational database of the customer was queried with database suitable remote connection. This connection could be achieved in many ways, by using a SQL-querying program, or for example by using custom written program codes.

4.2.3 Data format challenges

The data received from the FMI-API was in XML 1.0 data format encoded in UTF-8. The XML-file was a massive (10779 lines) collection of hourly predictions for the following 36 hours.

The data received from the avoindata.fi was in JSON data format. The JSON-file was a massive collection of coordinate arrays. Each of these arrays had all the corner coordinates for representing the grid on a map. JSON was also the data format in the vehicle location service. Each vehicle had a processing unit that posted continuous location information stream in JSON data format to an unique MQTT-topic .

The transport logs were stored in a relational database. Thus when querying the database, the data could be queried into whatever data format was selected. The Figure 4.3 illustrates the rows and columns of the database.



eventID	driverID	carID	orderID	startTime	endTime	startRoad	startHouse	startPostal	endRoad	endHouse	endPostal	passengers
111	4	2	3323	2017-07-30 T12:38:00	2017-07-30 T12:48:00	Kalevantie	4	33100	Alaverstaanraitti	3	33100	1
112	3	7	null	2017-07-30 T12:41:00	2017-07-30 T12:54:00	Rautatiekatu	27	33100	Vihiojantie	27	33800	2
113	2	3	3324	2017-07-30 T13:17:00	2017-07-30 T13:38:00	Laiturikatu	1	33230	Hämeenkatu	18	33200	4

Figure 4.3: The SQL-table of the transport log.

This meant that the system would have to cope with both the XML and JSON data formats and to be able to query the relational database using a remote connection.

4.2.4 Semantical challenges

All four data sources had different content. Only the FMI (GML [58]) and avoindata.fi (GeoJSON [43]) used standards in their data representation. The transport log system and the vehicle location system were both separately developed, and no standards were applied to their data representation. This created a semantical challenge.

The semantical challenge was the different representation of essentially the same data. As an example, location was represented as a street address in the relational database, whereas the FMI.fi used a GML [58] defined coordinates.

The lack of standards doesn't mean lack of information. Transport log database and the GPS-vehicle location service present their data in a non-standardised way. This semantical challenge needs to be handled.

4.3 Solving the interoperability challenge on different layers

To be able to meet the requirements presented in the Section 4.1, the system needed to have at least the components shown in Figure 4.4.

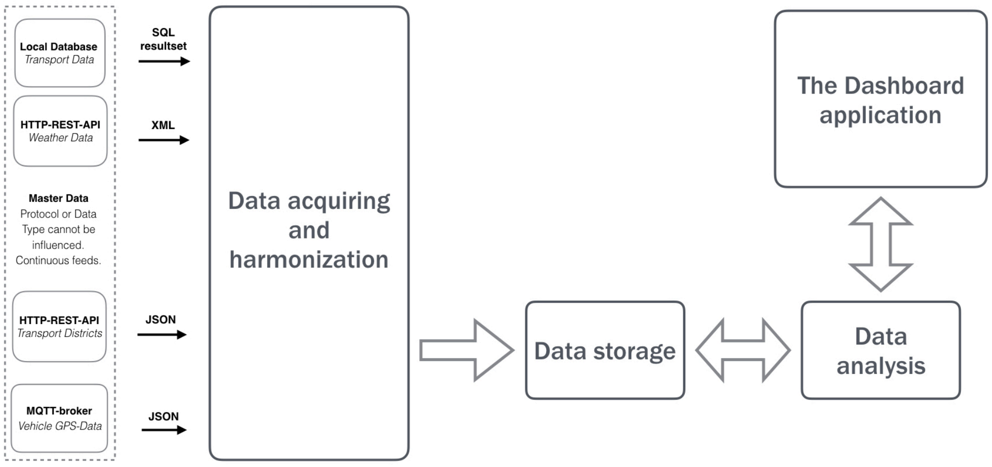


Figure 4.4: The general components of the system.

On the left, the four data sources and their different message protocols and data formats are presented. The first major component of the system needs to be able to cope with these heterogeneous message protocols. In Figure 4.4 this is the Data acquiring and harmonization component. Since the system only receives data, no actions towards the data sources are presented. The Data storage and Data Analysis component is responsible for storing data and providing the feature findings for the Dashboard web application.

As seen in Chapter 3, there are various possibilities on how to achieve interoperability among heterogeneous data sources. Thus the responsibilities of the previously presented components can vary. In the following Subsections, different solutions for providing interoperability among four selected data sources are presented.

In further subsections, there are two different decompositions presented for the Data acquiring and harmonization component.

4.3.1 Message protocol interoperability possibilites

In Chapter 3, there are essentially two variations presented about how to enable message protocol interoperability: proxying or message stream. The four data sources used the HTTP, MQTT and SQL-query engine protocols.

Using proxying (see Subsection 3.2.1 and 5.3) for handling message requests towards the database and the MQTT-broker would achieve interoperability in the system. By using a proxy, all request logic could operate on simply the HTTP. This functionality is presented in Figure 4.5.

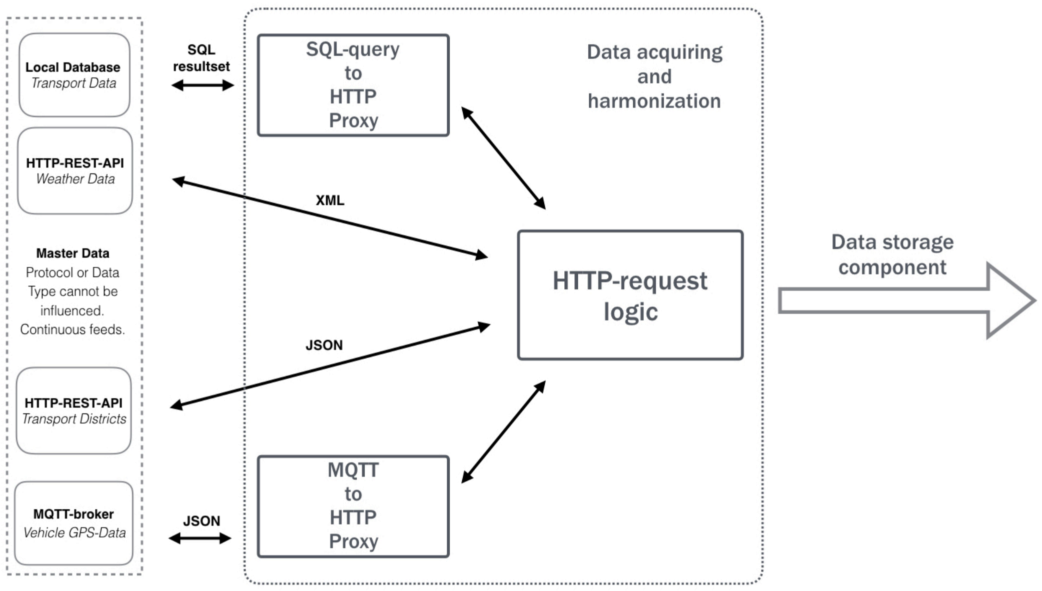


Figure 4.5: Connecting to data sources by using proxies.

Using proxying would simplify the request logic of the HTTP-request logic component. The HTTP-request logic component simply makes a request either directly to the data source or to the dedicated proxy. Thus the HTTP-request logic component is responsible for drawing the data into the system. The component itself is capable of making HTTP-requests, but not making a SQL-query or MQTT-subscription. For solving the message protocol interoperability challenge proxies are used.

When receiving a query, the SQL-query to HTTP Proxy transforms it into suitable SQL-query and retrieves the data from the database. After successfully executing the query, it forms and HTTP-respond and forwards the payload to the HTTP-request logic component. As seen in 2.2.1 there are multiple variations on how to perform these requests.

The MQTT to HTTP Proxy operates somewhat on the same logic, but instead of SQL-queries, it buffers all messages received to a vehicle specific MQTT-topic and responds to the HTTP-request logic component with them as the payload.

There would be a need to create proxies to support both the remote database connection and the MQTT. But the proxy development would also offer a possibility to provide interoperability on the data format layer since all messages are transformed from one message protocol to another. Proxies could for instance response to HTTP-request only by XML or JSON. Proxying would still not entirely solve the data format interoperability challenge since the XML, and the JSON would co-exist even after proxies are in place.

Another possibility would be to use the message streaming (see Subsections 3.2.3 and 3.3.2). In this scenario, all data sources were connected by using a Connector software component. Each component makes message protocol-specific requests/subscriptions and forwards all messages to single message stream. This possibility is presented in Figure 4.6.

Each Connector component could is a separate software process. For example, the HTTP Connector component could make timed requests to the Weather Data REST-API. When the Connector receives the HTTP-respond, it publishes a new message on the Message Stream with the payload from the request. This technique decentralizes the request process to each separate Connector component.

Since the message stream would have all the content of the four data sources, the message protocol interoperability challenge would be solved. On the other hand, this would need four different software components, each connecting to a different data source.

Each Connector component has essentially three main functionalities:

- It connects to the data source using message protocol specific techniques.
- It parses the payload from the message/respond.
- It publishes a new message to the Message Stream component, with the payload attached to it.

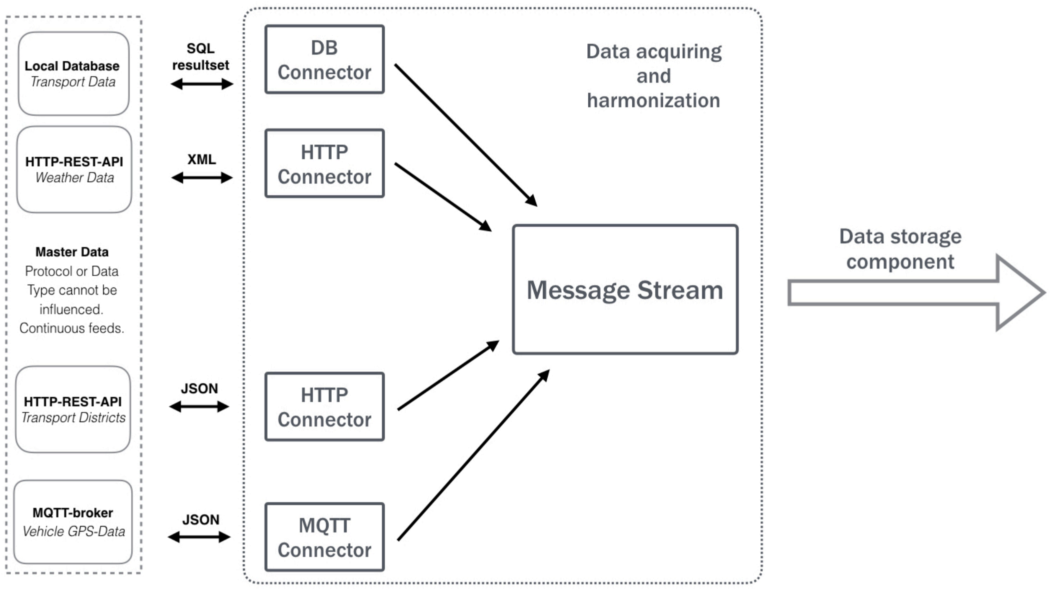


Figure 4.6: Connecting to data sources by message stream.

The connecting and payload parsing is often data source specific, but the message stream publishing is very reusable function among all Connector components.

4.3.2 Data format interoperability possibilities

As seen in Section 3.3, the data format interoperability is often solved by converting heterogeneous data formats into a single one. This can be done within either at the Data acquiring and harmonization or at the Data storage and Data Analysis software component (see Figure 4.4). If all data is stored in the Data storage component as it is received, every data analysis process needs to do data format conversion among heterogeneous data formats. In previous solutions, the data format conversion is done before the data is stored (see Subsections 3.3.1 and 3.3.2). This approach is also used this thesis's proposition.

Depending on which message protocol interoperability approach is used, the data format can be unified in two ways:

- If using the proxying (Figure 4.5), the HTTP-request logic component is also responsible for doing conversions into a single data format. There could be an additional component that would be responsible for data format conversions. Data format conversions always requests unique parsing attributes (e.g. getting the date of an event out from XML and place it into JSON). Since each data source already has unique message protocols properties, it might be feasible to also do all custom parsing in one place, in the Connector component. Since the majority of the data sources already present their data in JSON, this is the data format of choice. The Proxy components can also be used in data format conversion when they receive the content from the original data source; they do the conversion before responding to the HTTP-request.

- If using the message streaming (Figure 4.6), either all message protocol specific Connector components do the conversion into JSON, or the Message Stream component does the data format unifying before moving data to the Data storage component. There are many open source solutions for building the Message Stream. Rozik et. al. [69] use the Apache Kafka [27], which would suit this solution as well.

4.3.3 Semantic interoperability possibilities

As seen in Section 3.4, there are various ways to include metadata alongside the actual data. FMI offers a manual of all their ontologies used [42]. They mainly use different geospatial data representations of OGC Geospatial data models. The use of these ontologies provides an excellent reusability for all data received from FMI's REST-API.

The coordinate data from the Avoindata.fi [20] is presented in a GeoJSON data format. This lightweight data description format includes some metadata in the representation since users can rely on the fact that they receive a set of coordinates represented in the ETRS-GK24 coordinate model. The GeoJSON [43] is essentially an extended version JSON data format, which is intended for representing regions of space. Thus it is very suitable and reusable for the map grid representation.

There are also various alternatives for applying ontologies to the transport information stored in the customer's database. Li et. al. [49] present the taxi operation ontology, which can be used to represent the data from the customer's transport action database.

Also, the vehicle location data has various ontology alternatives, which would suit its content. The coordinates are presented in the WGS84 standard, which is a common standard in GPS-location systems [21]. There would be plenty of alternative coordinate models for the WGS84 like the ETRS-GK24 used previously for providing information to present map grid division.

Using such ontologies is highly useful, especially in a development phase, since once the development team has parsed and suited the various data sources in well-defined ontologies, they can be sure that the data is easily reusable and extensible. This is also important if looking at things from developers point-of-view since if there were changes on the development team during the actual development, the team can be sure, that all used data is well documented for all new team members.

But there are challenges when attempting to use a unifying semantic description model out of all data sources. One alternative would be to use a semantic gateway as presented by Desai et. al. [18]. This would allow the use of various domain specific ontologies or standards. The benefits of the semantic gateway model would be the well documented output data, but the drawback would be building such system for four data sources. The only drawbacks of such a system are practical, building such a system requires money and time.

One alternative would be to attempt to apply some ontology to all data. The OGC SensorThings API (see Subsection 2.4.2) offers an easy to use framework for semantic and data format interoperability of IoT-data. Since the weather observations and the vehicle location data are essentially observations out of the real world, data from these two data sources can be transformed into SensorThings API.

With a little ingenuity, even the data from the transport actions database can be transformed into OGC SensorThings API. The unique value of eventID in the database table (see Figure 4.3) is presented as a Thing according to the SensorThings API. This allows multiple Datastreams to be added to that Thing. Each of these Datastreams represents a column in the database. Each Datastream has only one ObserverdProperty, which describes the content of the Datastream. These ObservedProperties could be the driverID, carID, orderID, etc. of the database schema. The Observation would then be the actual value placed on that specific row in the database.

But to present the GeoJSON polygon data of the district map in OGC SensorThings API would be a challenge. Since the polygons are essentially arrays of coordinates, there is very little metadata included in them. One possibility would be to see the polygon as a Thing and to add each corner coordinate as a Location. Another but a more extensive possibility would be to use the same measures as in representing the transport log data.

The purpose of the solution was to be able to detect features out of the data

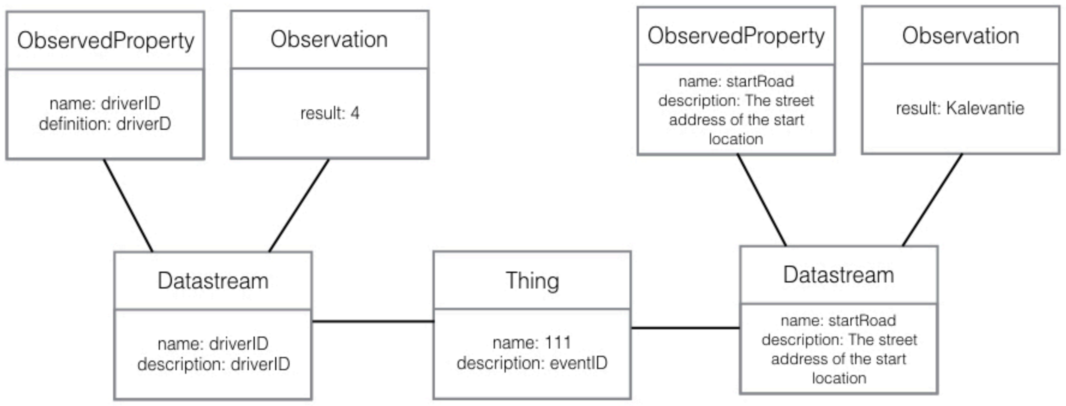


Figure 4.7: Simplified view of two values from the transport action database represented in OGC SensorThings API.

gathered from four heterogeneous data sources. To be able to do this, the Data Analysis component needs to compare values which originate from different data sources. This can be achieved (as seen in Subsections 3.3.1 and 3.3.2) even without the interoperable semantical model.

In previous solutions by Rozik et. al. [69] and Belli et. al. [7] data format interoperability is achieved before data reaches software components responsible for data analysis. Despite the harmonized data format, every data analysis query to the data needs semantical knowledge of the data. As an example, if we assume that previous transport action correlates to future ones, semantical knowledge of these attributes needs to be handled manually by the developers.

5 Evaluation of the proposed solutions

As seen in the previous chapter, there are various possibilities to solve the interoperability problem of the four heterogeneous data sources. In this chapter, an evaluation of these propositions is presented.

5.1 Message protocol interoperability propositions

Both the proxying and message stream was proposed. When considering the system as a whole, the message stream proposition would be more feasible to use. This is mainly because setting up or developing an entirely new proxy for remote database, and the MQTT-broker would require a lot of resources. Adding to this the request logic in the HTTP-request component, there would be much work without a real possibility to reuse components.

When considering the message stream proposition, each Connector component has features, that can be reused in each of the four Connectors like the publishing mechanism to the Message Stream. As a drawback, each of the data sources requires a Connector component, which also needs data source specific setup measures. On the other hand, this is needed even if using the proxying model: the same HTTP-GET request won't work to different data sources.

5.2 Data format interoperability propositions

If using the proxying solution, some of the data format interoperability can be achieved already at the proxies. But some of the data conversion logic remains the responsibility of the HTTP-request logic component (see Figure 4.5). It might be a bad design model to do the same task in two different software components. Thus the HTTP-request logic component would be more suitable for also providing the data format interoperability.

Another possibility was to use the message streaming. Since the Connector modules need to connect to heterogeneous message protocols, it also needs to be able to receive heterogeneous data formats. Thus it might be the most reasonable also to make the next data format harmonization phase and to publish all messages in unified data format to the message stream, in this case, the JSON.

5.3 Semantic interoperability propositions

Solving the semantical interoperability was challenging, since the four used data sources, each using an ontology or standard from entirely different domains. This cross-domain ontology challenge was solved by Desai et. al. [18]. They presented the Semantic Gateway as Service, which was able to utilize multiple ontologies to provide semantical interoperability. When also looking things from the perspective of the original objectives, M2M enabling semantic interoperability is not needed. To be able to do data analysis the data is a sufficient level of interoperability. Thus building a system as presented in the Subsection would be somewhat unnecessary.

Another alternative would be an attempt to unify all these representations to a single semantical model. For this, the use of OGC SensorThings API was proposed. This would offer benefits when querying data from the database. But as seen when describing the transport log data from customers database, there might be an excess of data stored in the database. But when defining the transport logs data or the transport districts data in OGC SensorThings API, a semantical interoperability will not be achieved.

6 Conclusion & Discussion

The scope of this Master's thesis is vast. The main idea is to present the interoperability challenge as a whole; none of the layers individually solve the interoperability problem, but any one of the layers can ruin it. For example, imagine a message received from IoT-device using the shared network and message protocol, but with different data formats. Despite two previous interoperability layers being solved, the lack of shared understanding of the used data format ruins the interoperability as a whole. But as this thesis has shown, this interoperability mismatch could be solved.

This complexity created challenges for writing this thesis since there were many interoperability challenges, on many layers, which needed addressing while remaining on quite a high level of abstraction. Another approach would have been to focus on some specific layer, like, e.g. to the network layer, but this would not present the whole scope of interoperability problems. Despite these challenges, since this thesis was inspired by a real-life development process, the interoperability as a whole was selected as the scope of this thesis.

The layered model for interoperability presented by Desai et. al. [18] was used to create a construction for this Thesis. This model was extended so that the data annotation model was divided into the data format and the semantical interoperability layers. Another approaches and constructions could have been selected, but these four were selected, due to their suitability for both to the academic literature domain, and also due to their easy adaptability for the actual software development offerings.

The actual software development took place during summer 2017. Ambientia Oy had a customer, which was interested in doing explorative research on the business possibilities, which could be produced by a predictive software. The business requirements mentioned in this thesis are thus derived from real-world business planning processes. The author of this thesis was mainly responsible for the conversations with the customer, especially about what the software should do. The development process was iterative. After discussions and planning with the customer, a new proposition was made until the customer approved the general design for the Proof-of-Concept to be built.

Quite early on in this process, the interoperability challenges of heterogeneous data sources was recognized as a challenge. After the initial design and requirement engineering processes, the author of this thesis focused on exploring the common knowledge base surrounding the interoperability challenge. Thus the actual software solution, which was built from the requirements and design principals presented in this thesis, is unknown to the author.

The interoperability challenge was fascinating but a challenging subject to study. This complexity of the research question created challenges for writing this Thesis. There were many interoperability challenges, on many layers, which needed addressing while remaining on quite a high level of abstraction. Another approach would have been to focus on some specific layer, like, e.g. to the network layer, but this would not present the whole scope of interoperability problems. Despite these challenges, since this Thesis was inspired by a real-life development process, the interoperability as a whole was selected as its scope.

Nevertheless, the author of this thesis hopes to have contributed to the challenging task of solving the interoperability problems of data-driven solutions. Solving interoperability would enable massive opportunities to software and IoT-solution developers since the development efforts could be focused on the substance of the solutions, rather than solving the complexities of heterogeneous data sources.

References

[1] AHLGREN, B., HIDELL, M., AND NGAI, E. C.-H. Internet of things for smart cities: Interoperability and open data. *IEEE Internet Computing 20*, 6 (2016), 52–56.

[2] ALLIANCE, L. Lorawan - technology. URL `https://docs.wixstatic.com/ugd/eccc1a_ed71ea1cd969417493c74e4a13c55685.pdf`, accessed 23.9.2017.

[3] ALLIANCE, W.-F. Wi-fi - technology. URL `https://www.wi-fi.org/discover-wi-fi/connect-your-life`, accessed 23.9.2017.

[4] ALLIANCE, Z. The zigbee low-power, low-cost, low-complexity networking for the internet of things. URL `http://www.zigbee.org/zigbee-for-developers/network-specifications/`, accessed 16.9.2017.

[5] ATZORI, L., IERA, A., AND MORABITO, G. The internet of things: A survey. *Computer networks 54*, 15 (2010), 2787–2805.

[6] BANDYOPADHYAY, S., AND BHATTACHARYYA, A. Lightweight internet protocols for web enablement of sensors using constrained gateway devices. In *Computing, Networking and Communications (ICNC), 2013 International Conference on* (2013), IEEE, pp. 334–340.

[7] BELLI, L., CIRANI, S., DAVOLI, L., MELEGARI, L., MÓNTON, M., AND PICONE, M. An open-source cloud architecture for big stream iot applications. In *Interoperability and Open-Source Solutions for the Internet of Things*. Springer, 2015, pp. 73–88.

[8] BIANCHINI, D., DE ANTONELLIS, V., AND MELCHIORI, M. A multi-perspective framework for web api search in enterprise mashup design. In *International Conference on Advanced Information Systems Engineering* (2013), Springer, pp. 353–368.

[9] BORENSTEIN, N. S., AND FREED, N. Multipurpose internet mail extensions (mime) part two: Media types. *Multipurpose internet mail extensions (MIME)*

part two: Media types (1996). URL `https://tools.ietf.org/html/rfc2046`, accessed 16.9.2017.

[10] BORMAN, C. Rfc 7049 concise binary object representation. URL `http://cbor.io`, accessed 16.9.2017.

[11] CASTELLANI, A. P., FOSSATI, T., AND LORETO, S. Http-coap cross protocol proxy: an implementation viewpoint. In *Mobile Adhoc and Sensor Systems (MASS), 2012 IEEE 9th International Conference On* (2012), IEEE, pp. 1–6.

[12] CERN. Cern open data portal. URL `http://opendata.cern.ch/about`, accessed 9.6.2017.

[13] CHEN, C. P., AND ZHANG, C.-Y. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences 275* (2014), 314–347.

[14] CHEN, H., CHIANG, R. H., AND STOREY, V. C. Business intelligence and analytics: From big data to big impact. *MIS quarterly 36*, 4 (2012), 1165–1188.

[15] COMMITTEE, T. J. P. E. G. J. Overview of jpeg. URL `https://jpeg.org/jpeg/index.html`, accessed 9.6.2017.

[16] DC SQUARE. Hivemq - enterprise mqtt broker). URL `https://www.hivemq.com`, accessed 16.10.2017.

[17] DESAI, P. Node-sgs 0.9 (semantic gateway as service for iot). URL `https://github.com/chheplo/node-sgs`, accessed 11.10.2017.

[18] DESAI, P., SHETH, A., AND ANANTHARAM, P. Semantic gateway as a service architecture for iot interoperability. In *Mobile Services (MS), 2015 IEEE International Conference on* (2015), IEEE, pp. 313–319.

[19] DÍAZ, M., MARTÍN, C., AND RUBIO, B. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer Applications 67* (2016), 99–117.

[20] DUKPA, J. The voting districts of tampere. URL `https://www.avoindata.fi/data/fi/dataset/tampereen-aanestysalueet`, accessed 23.6.2017.

[21] EPSG. Wgs84 - world geodetic system 1984. URL `https://epsg.io/4326`, accessed 1.12.2017.

[22] FIELDING, R., AND RESCHKE, J. Hypertext transfer protocol (http/1.1): Message syntax and routing. *Hypertext transfer protocol (HTTP/1.1): Message syntax and routing* (2014). URL `https://tools.ietf.org/html/rfc7230`, accessed 16.9.2017.

[23] FIELDING, R., AND RESCHKE, J. Hypertext transfer protocol (http/1.1): Semantics and content. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* (2014). URL `https://tools.ietf.org/html/rfc7231#section-4.3.1`, accessed 16.9.2017.

[24] FOUNDATION, A. S. Storm v. 1.1.1 - serialization. URL `http://storm.apache.org/releases/1.1.1/Serialization.html`, accessed 20.10.2017.

[25] FOUNDATION, T. A. S. Apache activemq (tm) is the most popular and powerful open source messaging and integration patterns server. URL `http://activemq.apache.org`, accessed 9.10.2017.

[26] FOUNDATION, T. A. S. Apache cassandra. URL `http://cassandra.apache.org`, accessed 19.8.2017.

[27] FOUNDATION, T. A. S. Apache kafka tm. URL `https://kafka.apache.org`, accessed 3.5.2017.

[28] FOUNDATION, T. A. S. Apache storm tm. URL `http://storm.apache.org`, accessed 3.5.2017.

[29] FREED NED, K. M. Media types. URL `https://www.iana.org/assignments/media-types/media-types.xhtml`, accessed 18.9.2017.

[30] GEORGAKOPOULOS, D., AND JAYARAMAN, P. P. Internet of things: from internet scale sensing to smart services. *Computing 98*, 10 (2016), 1041–1058.

[31] GERACI, A., KATKI, F., MCMONEGAL, L., MEYER, B., LANE, J., WILSON, P., RADATZ, J., YEE, M., PORTEOUS, H., AND SPRINGSTEEL, F. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*. IEEE Press, 1991.

[32] GOLDSTEIN, S. M., JOHNSTON, R., DUFFY, J., AND RAO, J. The service concept: the missing link in service design research? *Journal of Operations management 20*, 2 (2002), 121–134.

[33] GOOGLE. Unicode nearing 50 URL `https://googleblog.blogspot.fi/2010/01/unicode-nearing-50-of-web.html`, accessed 9.6.2017.

[34] GROUP, I. . W. Official ieee 802.11 working group project timelines - 2017-09-20. URL `http://www.ieee802.org/11/Reports/802.11_Timelines.htm`, accessed 23.9.2017.

[35] GROUP, N. W. Ascii format for network interchange. URL `https://www.rfc-editor.org/rfc/pdfrfc/rfc20.txt.pdf`, accessed 9.6.2017.

[36] GROUP, N. W. Hypertext transfer protocol – http/1.1 - rfc 2616. URL `https://tools.ietf.org/html/rfc2616`, accessed 6.6.2017.

[37] GROUP, N. W. A tcp/ip tutorial. URL `https://tools.ietf.org/html/rfc1180`, accessed 8.6.2017.

[38] HENSON, C. A., PSCHORR, J. K., SHETH, A. P., AND THIRUNARAYAN, K. Semsos: Semantic sensor observation service. In *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on* (2009), IEEE, pp. 44–53.

[39] IETF. Rfc 7252 constrained application protocol. URL `http://coap.technology`, accessed 6.6.2017.

[40] INC., N. Introducing the nginx application platform. URL `https://nginx.org/en/`, accessed 9.10.2017.

[41] INSTITUTE, F. M. The finnish meteorological institute's open data. URL `https://en.ilmatieteenlaitos.fi/open-data`, accessed 27.6.2017.

[42] INSTITUTE, F. M. Open data manual. URL `http://en.ilmatieteenlaitos.fi/open-data-manual`, accessed 29.11.2017.

[43] INTERNET ENGINEERING TASK FORCE (IETF), R.-. The geojson format. URL `https://tools.ietf.org/html/rfc7946#page-12`, accessed 23.6.2017.

[44] ISO/IEC. Final committee draft iso/iec fcd 8859-16. URL `http://www.open-std.org/JTC1/SC2/WG3/docs/n411.pdf`, accessed 9.6.2017.

[45] JIN, T., NOUBIR, G., AND SHENG, B. Wizi-cloud: Application-transparent dual zigbee-wifi radios for low power internet access. In *INFOCOM, 2011 Proceedings IEEE* (2011), IEEE, pp. 1593–1601.

[46] Ecma-404 the json data interchange standard - introduction. URL `http://json.org`, accessed 18.5.2017.

[47] KOANLOGIC. Webthing - core protocols implementation and ramblings. URL `https://github.com/koanlogic/webthings`, accessed 7.10.2017.

[48] KRUGER, C., ABU-MAHFOUZ, A., AND HANCKE, G. Rapid prototyping of a wireless sensor network gateway for the internet of things using off-the-shelf components. In *Industrial Technology (ICIT), 2015 IEEE International Conference on* (2015), IEEE, pp. 1926–1931.

[49] LI, Y., ZHAO, J., AND WANG, H. An ontology-based taxi operation query system. In *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on* (2015), IEEE, pp. 832–836.

[50] LOGMEIN, I. Xively. URL `https://www.xively.com/xively-iot-platform`, accessed 19.8.2017.

[51] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on* (2012), IEEE, pp. 177–182.

[52] MAINETTI, L., PATRONO, L., AND VILEI, A. Evolution of wireless sensor networks towards the internet of things: A survey. In *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on* (2011), IEEE, pp. 1–6.

[53] MATHWORKS. Thingspeak. URL `https://se.mathworks.com/help/thingspeak/?requestedDomain=www.mathworks.com`, accessed 19.8.2017.

[54] MCAFEE, A., BRYNJOLFSSON, E., DAVENPORT, T. H., PATIL, D., AND BARTON, D. Big data. *The management revolution. Harvard Bus Rev 90*, 10 (2012), 61–67.

[55] MIKE USHOLD, C. M. Semantic integration and interoperability using rdf and owl. URL `https://www.w3.org/2001/sw/BestPractices/OEP/SemInt/`, accessed 9.6.2017.

[56] MJCOAP.ORG. mjcoap. URL `http://www.mjcoap.org`, accessed 9.10.2017.

[57] MURDOCK, P., BASSBOUSS, L., BAUER, M., ALAYA, M. B., BHOWMIK, R., BRETT, P., CHAKRABORTY, R. N., DADAS, M., DAVIES, J., DIAB, W., ET AL. Semantic interoperability for the web of things, 2016.

[58] O.G.C. Geography markup language. URL `http://www.opengeospatial.org/standards/gml`, accessed 28.10.2017.

[59] O.G.C. Ogc sensorthings api. URL `https://github.com/opengeospatial/sensorthings`, accessed 11.6.2017.

[60] O.G.C. Sensor web enablement (swe). URL `http://www.opengeospatial.org/ogc/markets-technologies/swe`, accessed 15.9.2017.

[61] O.G.C. Sensorthings api part 1: Sensing. URL `http://docs.opengeospatial.org/is/15-078r6/15-078r6.html`, accessed 22.9.2017.

[62] PALFREY, J., AND GASSER, U. *Interop: The promise and perils of highly interconnected systems*. Basic Books (AZ), 2012.

[63] POSTEL, J. User datagram protocol, 1980.

[64] PROGRAMMABLEWEB. Programmableweb - api's, mashups and the web as aplatform. URL `https://www.programmableweb.com`, accessed 24.4.2017.

[65] PROVOST, F., AND FAWCETT, T. *Data Science for Business: What you need to know about data mining and data-analytic thinking*. O'Reilly Media, Inc., 2013.

[66] RANABAHU, A., NAGARAJAN, M., SHETH, A. P., AND VERMA, K. A faceted classification based approach to search and rank web apis. In *Web Services, 2008. ICWS'08. IEEE International Conference on* (2008), IEEE, pp. 177–184.

[67] REPICI, D. J. How to: The comma separated value (csv) file format. create or parse data in this popular pseudo-standard format. URL `http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm`, accessed 18.9.2017.

[68] ROWLAND, C., GOODMAN, E., CHARLIER, M., LIGHT, A., AND LUI, A. *Designing Connected Products: UX for the Consumer Internet of Things*. O'Reilly Media, Inc., 2015.

[69] ROZIK, A., TOLBA, A., AND EL-DOSUKY, M. Design and implementation of the sense egypt platform for real-time analysis of iot data streams. *Advances in Internet of Things 6*, 4 (2016), 65–91.

[70] SCHWARTZ, M. *Mobile wireless communications*. Cambridge University Press, 2004.

[71] SHAFRANOVICH, Y. Rfc 4180: Common format and mime type for comma-separated values (csv) files. - (2005). URL `https://tools.ietf.org/html/rfc4180`, accessed 20.9.2017.

[72] SHELBY, Z., HARTKE, K., AND BORMANN, C. The constrained application protocol (coap). *The Constrained Application Protocol (CoAP)* (2014). URL `https://tools.ietf.org/html/rfc7252`, accessed 16.9.2017.

[73] SIG, B. The bluetooth (r) core specification. URL `https://www.bluetooth.com/specifications/bluetooth-core-specification`, accessed 16.9.2017.

[74] SIGMA DESIGNS, I. Z-wave: The basics. URL `http://www.z-wave.com/faq`, accessed 24.9.2017.

[75] SQUID CACHE.ORG. squid-cache.org - optimising web delivery. URL `http://www.squid-cache.org`, accessed 7.10.2017.

[76] STANDARD, O. Mqtt version 3.1.1. URL `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`, accessed 8.6.2017.

[77] STOCKER, M. Sensor observations, ssn example in rdf. URL `http://markusstocker.com/sensor-observations/`, accessed 22.9.2017.

[78] THANGAVEL, D., MA, X., VALERA, A., TAN, H.-X., AND TAN, C. K.-Y. Performance evaluation of mqtt and coap via a common middleware. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on* (2014), IEEE, pp. 1–6.

[79] The unicode consortium standard: A technical introduction. URL `http://www.unicode.org/standard/principles.html`, accessed 8.6.2017.

[80] TSE, D., AND VISWANATH, P. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[81] W3C. Extensible markup language (xml) 1.1 (second edition). URL `https://www.w3.org/TR/2006/REC-xml11-20060816/`, accessed 18.5.2017.

[82] W3C. Owl 2 web ontology language document overview (second edition). URL `https://www.w3.org/TR/owl2-overview/`, accessed 21.9.2017.

[83] W3C. Rdf 1.1 concepts and abstract syntax. URL `https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`, accessed 18.5.2017.

[84] W3C. Sparql 1.1 query language. URL `https://www.w3.org/TR/sparql11-query/`, accessed 18.5.2017.

[85] W3C. Web ontology language (owl). URL `https://www.w3.org/OWL/`, accessed 16.6.2017.

[86] W3C, O. . Semantic sensor network ontology. URL `https://www.w3.org/TR/vocab-ssn/`, accessed 15.9.2017.

[87] WPAN, I. . Ieee 802.15 wpan task group 4 (tg4). URL `http://www.ieee802.org/15/pub/TG4.html`, accessed 23.9.2017.

[88] ZANELLA, A., BUI, N., CASTELLANI, A., VANGELISTA, L., AND ZORZI, M. Internet of things for smart cities. *IEEE Internet of Things journal 1*, 1 (2014), 22–32.

[89] ZHANG, X., AND SCHULZRINNE, H. Voice over tcp and udp. *Department of Computer Science, Columbia University, Tech. Rep. CUCS-033-04* (2004).

[90] ZHU, Q., WANG, R., CHEN, Q., LIU, Y., AND QIN, W. Iot gateway: Bridgingwireless sensor networks into internet of things. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on* (2010), IEEE, pp. 347–352.