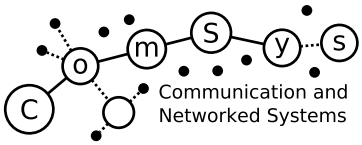
FACULTY OF
COMPUTER SCIENCE



Communication and Networked Systems

Master Thesis

Enhanced CoAP resource discovery - Application layer interfaces and highly targeted query filters

Jawad Ahmad

Matr. 209873

Supervisor: Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor: MSc. Marian Buschsieweke

Institute for Intelligent Cooperating Systems, Otto-von-Guericke-University Magdeburg

March 11, 2019

Abstract

Abstract

Constrained Application Protocol (CoAP) has the potential to be a key actor in improving the Internet of Things (IoT) ecosystem. Despite its strengths, CoAP's standardization is still under development; thus, CoAP suffers from a lack in maturity in certain regards. Resource discovery is such an aspect where CoAP's standard methodology exhibits inadequacy. This deficiency is exemplified by a use-case scenario where employing standard methodology of CoAP's resource discovery is shown to be inefficient in terms of both time and space. One of the most significant shortcomings of the standard methodology, in particular, is its incapability to filter resources based on individual attribute values. This thesis presents a solution to this limitation by implementing a query processing layer on top of a standard CoAP server, allowing highly targeted resource filtering. In addition to decreasing the payload sizes through the use of complex filters, the proposed implementation exercises the use of Concise Binary Object Representation (CBOR). The performance evaluation has shown that CBOR reduces payload sizes by as much as 45% when compared to plain-text encodings such as JavaScript Object Notation (JSON). The performance evaluation also revealed that for queries that targeted 92% of the total resources, a reduction of up to 54% response sizes as compared to a standard CoAP server was seen. The proposed implementation is shown to have successfully filtered resources by attribute values while being time and space efficient.

Contents

List of Figures

List of Tables

Listings

Acronyms

AWS Amazon Web Services. 31

CBOR Concise Binary Object Representation. iii, 17–19, 21–24, 27, 29, 32
CoAP Constrained Application Protocol. iii, 4–11, 13, 16, 18, 19, 26, 27, 29, 31, 32
CoRE Constrained RESTful Environments. 32

HTTP HyperText Transfer Protocol. 2, 4, 5, 9

IANA Internet Assigned Numbers Authority. 29
IoT Internet of Things. iii, 3, 9

JSON JavaScript Object Notation. iii, 1, 7, 17, 18, 21, 24, 27, 30, 32

NCP Network Control Protocol. 1

QoS Quality of Service. 10

RD Resource Directory. 7
REST Representational State Transfer. 2

TCP Transmission Control Protcol. 4, 5
TCP/IP Transmission Control Protocol / Internet Protocol. 1, 2

UDP User Datagram Protocol. 4, 5, 18
URI Uniform Resource Identifier. 5, 13, 14, 19, 21, 22, 29

WWW World Wide Web. 2

CHAPTER 1

Introduction

Invention of Transmission Control Protocol / Internet Protocol (TCP/IP) [1] marked the advent of modern internet, enabling communication between computers that were thousands of miles apart. The layer-based nature of the protocol allowed gradual and seamless integration of existing networks that were running on older and inefficient protocols like Network Control Protocol (NCP) [2]. The design principle of TCP/IP enables the application layer of the host at one end of the communication channel to assume a direct connection to the application layer of the host at the other end. The transport layer can safely assume so as well. This is depicted in Figure 1.1. The solid lines describe the actual path of the data packets, whereas the dashed lines are virtual direct data paths.

In an effort to demonstrate a peculiar characteristic of TCP/IP, an imaginary email service, that resides at the sender side is illustrated. It is also imagined that the email client resides at the receiver end. The email message, shown in Listing 1.1 as JSON, is generated at the application layer of the sender.

```
{
  "From": "Device1",
  "To": "Device2",
  "Subject": "Hello",
  "Body": "Hello from Device1"
}
```

Listing 1.1: Email from sender device

Even though this payload goes through various transformations and encapsulations while passing through different layers, the data that the application layer of receiver eventually gets is exactly the same [1] as Listing 1.1. The receiver side is free to parse the JSON structure and extract the fields. This design scheme has some interesting outcomes. The sender could have sent *any* kind of data and the receiver would have received it unaltered. This aspect of TCP/IP gave rise to Application Layer Protocols, where the applications are free to encapsulate/interpret data in whatever way they desire.

[1] assuming a reliable transmission protocol

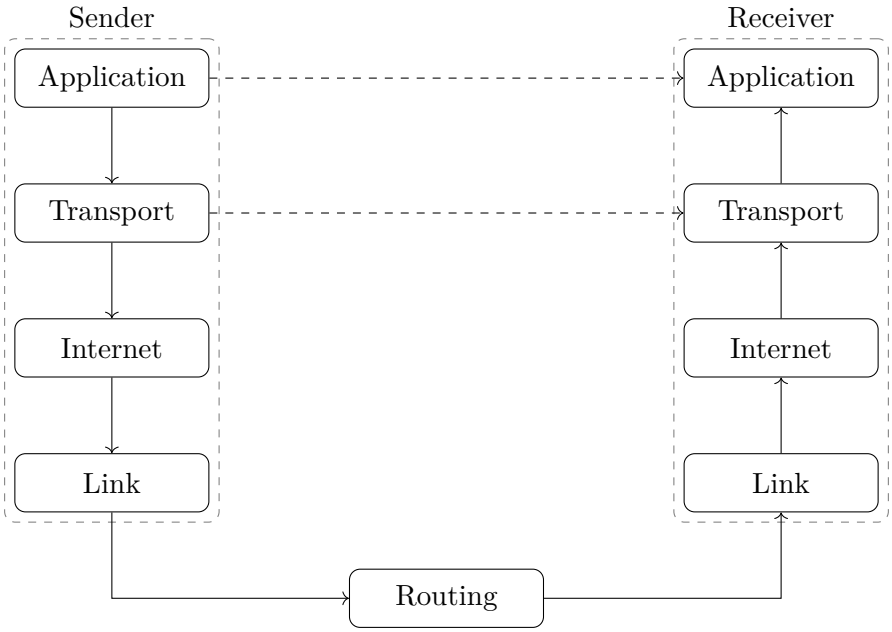


Figure 1.1: TCP/IP layers

1.0.1 Rise of World-Wide-Web

TCP/IP allowed basic services, like file-sharing and email, to exist between computers on the internet. However, the "web" as we know it today didn't exist until the invention of World Wide Web (WWW) in late 1980s and formalization of HyperText Transfer Protocol (HTTP) [3] in late 1990s. HTTP is an application layer protocol that introduces the concept of *resources* hosted / served on web-servers and requested by web-clients. A resource encompasses any form of content that a server is willing to offer to a client. For example, a list of names of employees of a company is a resource that the company's server could offer.

HTTP operates on a request-response model. The clients are responsible for initiating a *request* for a resource, while the servers are responsible for returning the requested resource as a *response*. HTTP is also a *stateless* protocol, i.e. every new request by clients is independent of all past and future requests. In other words, requests are self-contained transactions that include all client-side information that servers require in order to process the request. This leads to server designs that do not require the servers to store the client information. This form of interaction between clients and servers is referred to as Representational State Transfer (REST) [4]. Servers that operate in RESTful style support HTTP verbs such as: `GET`, for retrieval of resources; `POST`, for creation of resources; `PUT`, for replacement of current resource representation with some other representation; `PATCH`, for updating the resources; and `DELETE` for deletion of resources.

This begs a question: How does a client know what resources are available on server? The answer to this question is manifold. A common approach among web-servers is to present the client with an `index.html` page, which lists all the available links. Nonetheless, because the *clients* are supposed to be humans, any technique that makes it easy for a client to find the relevant content can be implemented. This is the basis of User Experience field in the

recent web technology. However this changes quickly when the clients are not humans, but machines.

1.0.2 Internet of Things

The term IoT refers to the eco-system of small, low-cost and low-power computers that are connected to the internet. These computers, generally called embedded devices, can be found everywhere nowadays; like inside home-appliances, vehicles, wearables, and buildings to name a few. These devices mostly have very low processing power and available memory space, are powered using low capacity batteries, and operate in lossy networks. On the other hand, these devices are expected to last for years while reliably transmitting and receiving data, which is made possible by virtue of the increased power efficiency [5]. The ever-increasing usage of these devices can be attributed to the fact that the prices of low-end microprocessors have been dropping for decades [6]. Most of these devices have data to offer to entities that inquire for it.

In order to explore possibilities of ways for a device to transmit its data, an example of a sensor that can measure temperature in both Celsius and Fahrenheit units is given. As an embedded device, the sensor is capable of transmitting the temperature values to other devices in its network. There are two ways this sensor can transmit its data, listed below with their advantages and disadvantages.

1. Broadcast the temperature values to all devices with a certain frequency.
    a) Pros:
        i. Easy to implement.
        ii. Energy consumption is constant regardless of number of devices in the network.
        iii. Sensor device can sleep between broadcasts to conserve power.
    b) Cons:
        i. Broadcasts are a wastage of energy and network bandwidth if none of the devices in the network needs sensor data.
        ii. High Network traffic in case where there are a lot of similar devices that broadcast data.
        iii. The consumer devices may not agree with the broadcast frequency.
        iv. Requesting devices have to be in a listening state all the time in order to capture all broadcasted data.
2. Wait for a request from a device in the network, and respond with a temperature value.
    a) Pros:
        i. Energy is conserved when there are no requesting devices.
        ii. Low network traffic if the number of requesting devices, or the request frequency, is low.

iii. Requesting devices have control over the data transmission frequency.

b) Cons:

i. Relatively harder to implement.

ii. High network traffic if the number of requesting devices, or the request frequency, is high.

iii. Sensor device has to be in a listening state all the time in order to capture all requests.

As evident, the choice of data transmission methodology depends on the application scenario. In most practical applications, however, there are numerous such sensor devices. Because the first method is inefficient for many broadcasting nodes, the second method is almost always the preferred one. This method, where the sensor device waits for request before responding with data, is reminiscent of an HTTP server as discussed in Section 1.0.1. The sensor device could define the following two HTTP resources:

1. `GET /temp/celsius`
2. `GET /temp/fahrenheit`

As discussed earlier, embedded devices such as the sensor device in question, have low computing power and available memory. The idea of implementing a full-fledged HTTP server inside this kind of devices is far-fetched. Furthermore, in practical cases like these the networks are lossy. HTTP uses Transmission Control Protcol (TCP) as the underlying transport mechanism which is unsuitable in a lossy environment. Instead, User Datagram Protocol (UDP) [7] is better suited than TCP in these scenarios for the reasons explained next.

1. UDP is connection-less, as opposed to TCP which is connection based. This means that in UDP, the two communicating devices don't have to go through a lengthy connection negotiation process.
2. UDP messages are "unreliable", as opposed to TCP whose messages are reliable. This may sound a negative aspect for UDP at first, but really it simplifies and quickens the process of messaging between two simple nodes.
3. UDP header size is 8-bytes, versus TCP's 20-bytes header size, making UDP payloads inherently smaller.
4. UDP has a concept of "message boundary". This means that when a device sends a UDP message, the receiving device either gets the full message or nothing at all, further simplifying the messaging process. In TCP, there is no message boundary. So an application-level message can get partially transmitted or received before the connection drops.

1.0.3 CoAP

As stated in Section 1.0.2, HTTP is unsuitable for constrained embedded devices that offer resources. A protocol that is somewhat similar to HTTP but with only basic features, and uses UDP will be ideal. CoAP [8] is an attempt at designing such a protocol. CoAP uses HTTP-like request-response model, and has the concept of resources. It specifies basic

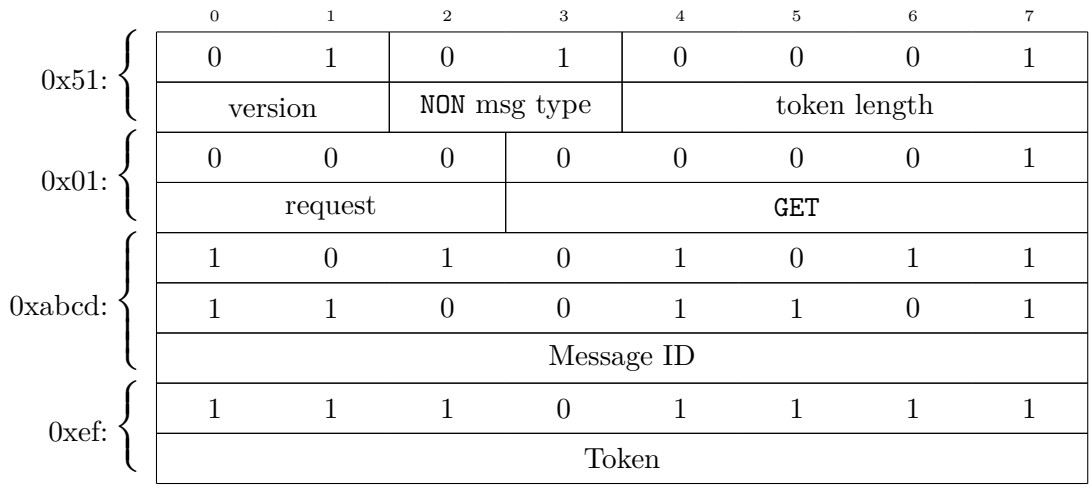


Figure 1.2: CoAP GET message

HTTP verbs like `GET` and `POST`. It runs atop UDP bringing all the benefits of UDP as discussed in Section 1.0.2 along with it. CoAP does allow using TCP too in case the application design absolutely demands it [9]. CoAP uses binary components for the messages, as opposed to plain-text in HTTP. Listing 1.2 shows the smallest possible HTTP `GET` vs CoAP `GET` message.

```
HTTP: GET / \gls{HTTP}/1.1\r\nHost:www.example.com\r\n\r\n
CoAP: 0x51 0x01 0xab 0xcd 0xef
```

Listing 1.2: HTTP GET vs CoAP GET

HTTP needs 25 bytes, excluding `www.example.com`, versus 5 bytes for CoAP. It can already be seen that CoAP messages are not meant be human-readable. It is harder to construct a CoAP message by hand. Figure 1.2 shows a quick breakdown of the CoAP `GET` message.

As mentioned earlier, CoAP uses UDP as the transport. So the missing reliability mechanism is built into CoAP itself at the application layer. In Figure 1.2, the message type is `NON`, or non-confirmable. `NON` messages are fire-and-forget kind of messages, where the sender doesn't expect an acknowledgement from the receiver. The sender still needs to temporarily keep track of the *Token* values sent in case there is any response from the receiver, which will have the matching *Token* value. Another kind of message type is `CON`, or confirmable. These messages expect an `ACK` type of message as response from the receiver, and are repeatedly sent until they get the acknowledgement. Message IDs are used to match the sent message with their `ACK` responses, so the sender needs to keep the IDs for `CON` messages in memory. Responses to requests of type `CON` are normally `CON` themselves. This means the original requester needs to send back an `ACK` after receiving a response. `NON` responses are usually sent for `NON` requests, but can be `CON` too. Shown in Table 1.1 is a comparison between TCP, UDP, and CoAP.

Another useful feature of CoAP is the discovery of all services offered by a server. Any inquiring entity can call `GET /.well-known/core` to get a list of URIs that are available.

	TCP	UDP	CoAP
Semantics	Connection oriented	Datagram oriented	Request/response semantics
Reliability	Yes	No	Yes, optionally
Order of Data	FIFO (order preserved)	Datagrams may arrive out of order	May arrive out of order for NSTART > 1 (see [8])
Duplicates	Detected and ignored	A datagram may arrive more than once	Detected and ignored
Congestion Control	Yes	No	Yes, but basic
16-bit Port Number	Yes	Yes	Yes (as it uses UDP)
CRC Checksum	Yes	Yes	Yes (as it uses UDP)

Table 1.1: Comparison of TCP, UDP and CoAP

Here's an example of resource discovery request-response pair.

```
GET /.well-known/core

</temperature/celsius>;rt="temperature-c";if="sensor",
</temperature/fahrenheit>;rt="temperature-f";if="sensor"
```

CoAP's standard resource discovery mechanism is useful as far as listing *all* resources offered by a server is concerned. In most practical applications, however, the inquiring entities are not interested in everything a server has to offer. The requesting devices need a way to filter the resources list, absence of which has serious consequences for practicality of a device network. Because of this, CoAP describes a methodology to tailor resource discovery requests for the server to return only the resources that match a certain criteria. As will be explained in Section 1.1, this criteria specification is basic at best. In other words, CoAP's resource discovery is useful for trivial situations, but proves to be mundane in complex scenarios such as the use case discussed in Section 1.1.

1.1 Motivation

The following use case is used to exemplarily point out the challenges and limitations of CoAP's standard resource discovery. The use-case conceptualizes a 15 storey building with 25 rooms on each floor. A company is given a contract to install sensors of different kinds throughout the building. The contract states that sensor management user-interface (UI) needs to have the following capabilities:

1. The UI needs to act as a CoAP client, with all communication going to a CoAP resource directory.
2. Ability to see the sensors location: *floor* and *room*

3. Ability to see the sensors type: *unit*
4. Ability to see and configure a *gain* parameter for each sensor that will allow the management team to calibrate sensors regularly.
5. Ability to service queries of the following nature: *Show all sensors that are in room 1 to 4 of floor 3 and floor 11 with gain smaller than 2.1, or sensors in room 1 of any floor with gain bigger than 3.3*

A design engineer would deem the standard CoAP Resource Directory (RD) [10] model an *almost* perfect fit, with a few limitations. The RD can present a "lookup interface", which can enlist all sensors registered in the RD. Here's an example of how a resource lookup for 5 different sensors could look like based on [11]:

```
REQ:    GET /rd-lookup
RES:    2.05 Content (application/link-format)
        </rd-lookup/s>;if="core.b"

REQ:    GET /rd-lookup/s
RES:    2.05 Content (application/senml+json)
        {"e":[
                {"n": "accel", "r": 19, "f": 11, g: 1.97, u: "g"},
                {"n": "temp", "r": 2, "f": 14, g: 4.71, u: "degC"},
                {"n": "humid", "r": 21, "f": 0, g: 3.72, u: "%"},
                {"n": "press", "r": 10, "f": 1, g: 1.35, u: "Pa"},
                {"n": "light", "r": 12, "f": 9, g: 0.7, u: "lx"}
        ]}

REQ:    POST /rd-lookup/s/accel/g
        1.5
RES:    2.04 Changed

REQ:    GET /rd-lookup/s/accel
RES:    2.05 Content (application/senml+json)
        {"n": "accel", "r": 19, "f": 11, g: 1.5, u: "g"}
```

As evident from the request/response pairs above, company specification number 1 through 4 are well covered. All sensor attributes can be listed and printed on UI screen after parsing JSON output from 2nd request. 3rd request is used to change *gain* for acceleration sensor. 4th request is used to confirm if the gain has indeed been changed.

Note that the UI software is already fully aware of the sensor resource type and its interface. In other words, it knows what *n, r, f, g* and *u* mean in response for 2nd request. It also knows that a POST request is required at `/rd-lookup/SENSOR/g` to change gain value of a sensor.

One could argue that specification number 5 is also fully satisfied because the UI software already has all the data for sensors. All it has to do is apply filters to data it received from the above requests, and present to the user. That is entirely true. However, this approach has some dire scalability issues. In the case where there is not one but several UI clients, if each client downloads all data each time it is powered on for simple queries like *Show sensors from room 1 of floor 1*, there is going to be huge data overheads and wastage of bandwidth. Not only that, but energy consumption also increases due to more wireless activity. Increasing the number of sensors worsens these costs even more, leading to a high probability of network congestion. Figure 1.3 shows the increase in packets quantity as the number of clients is increased from 1 to 100, and number of sensors is increased from 5 to

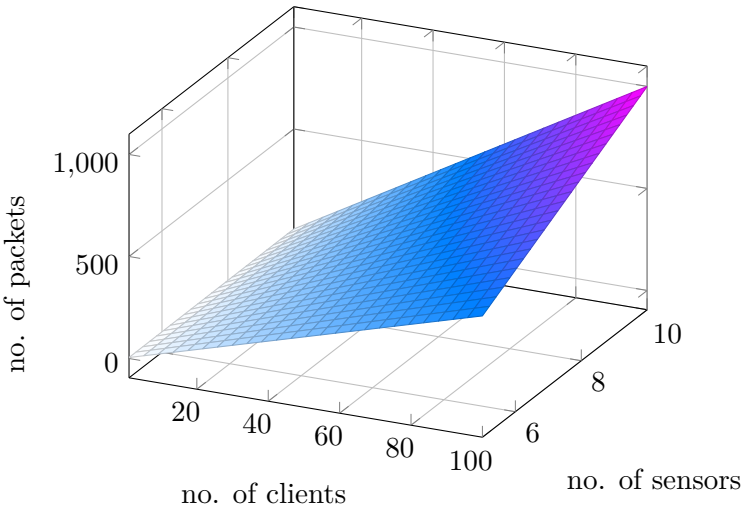


Figure 1.3: Quantity of packets vs number of clients and sensors

100.

One possible solution to this problem is to let the resource directory do the filtering, and return the filtered sensors that match the query. In fact, [10] states that each registered resource can have arbitrary attributes which can be used for lookup using a query string. For example, the following request/response pairs lookup sensors that are in room 7 of floor 10:

```
REQ:    GET /rd-lookup/s?r=7&f=10
RES:    2.05 Content
        {"e":[
                {"n": "humid", "r": 7, "f": 10, g: 3.72, u: "%"},
                {"n": "light", "r": 7, "f": 10, g: 0.7, u: "lx"}
        ]}
```

The query string `r=7&f=10` in the above request could be interpreted in two ways by the resource directory:

1. Show sensors from room 7 AND floor 10.
2. Show sensors from room 7 OR any room of floor 10.

The resource directory is free to interpret the query in any one of the above ways, but not both. This severely limits the complexity of queries that a CoAP client can issue. The reason is that there is no standard way of incorporating `AND` / `OR` operators in the query string. Furthermore, the `=` token, acting as a mere key-value separator, only allows for *is equal to* type of comparison. There is no way to incorporate *is less than* or *is greater than* comparisons.

This immediately presents a problem for the design engineer trying to express the query mentioned in the specification number 5 in a standard way. The following expression fully describes the query in question:

```
(((room<5&floor=3)|(room<5&floor=11))&gain<2.1)|(room=1&gain>3.3)
```
(1.1)

The proposed implementation covers this particular problem, while keeping the data transactions involved in the process as energy-efficient and network-friendly as possible.

1.2 Thesis Structure

This thesis is structured to build up a case, starting from Chapter 1, towards a better CoAP resource discovery. Introduction to IoT, importance of CoAP, and explanation of how CoAP is a counterpart of HTTP in embedded ecosystem are transcribed in Chapter 1 section 1.0.1 through 1.0.3. A use-case scenario, explaining how CoAP's standard resource discovery falls short in certain situations, is described in Section 1.1. Review of current research regarding CoAP's discovery mechanisms is done in Section 1.3. The section also explores how current research's focus is aligned to the use-case described in the thesis.

The solutions to the problems that the use-case puts forth are presented in Chapter 2. The software architecture and its interface to the outside world is described in Section 2.1. Experimentation employing random environmental configurations is documented in Section 2.2.

The proposal is completed by evaluation of the experimental results in Chapter 3. The conclusion of the thesis is also presented in this chapter.

1.3 Related Work

D. Pfisterer et al. [12] describe SPITFIRE: a globally linked network of devices with well-defined "Semantics", which allows machines to deduce logic based on a globally accessible knowledge base. They describe a use-case scenario where empty rooms in a building need to be found. SPARQL is used to search in the knowledge base based on basic node attributes, e.g. sensor location. Their work focuses on harnessing simple device states for complex logic inference.

C. Perera et al. [13] propose CASSARAM: a context-aware sensor search system where user can describe attribute values for exact and ranked matching in their queries. The paper focuses on decreasing sensor search processing time and memory usage by optimizing search algorithm at the server end. The use cases include large scale sensor databases, where multiple powerful machines are involved. They describe sensing-as-a-service application as their primary motivation.

M. Ruta et al. [14] describe semantic matchmaking on the data generated by participating nodes. The results are ranked by similarity to the search criteria. They propose CoAP based gateways that keep record of local nodes activity, and use that record to respond to resource discovery requests. The discovery requests are composed in Manchester Syntax, incorporating discrete values for the attributes. However, it is not clear how continuous attributes are handled. The data overheads, as compared to standard CoAP resource discovery, for complex queries involving multiple resource types is not explained either. Another two papers by M. Ruta et al. [15, 16] employ similar methodology with a very specific use case. The following is an example query from one of the papers:

```
coap://193.204.59.75:5683/.well-known/core?&ro=SSN-XG-IRI&sd=yyyyyy=&at=30004&lg
    =16.763571&lt=41.079769&md=800&st=2&sr=70
```

The query mentions latitude *lt*, longitude *lg*, and a maximum distance *md* to search for sensors. The server responds with the following 2 sensors that lie within that radius

```
</Hts2030HumidSens>;ct=0;ct=41;at=30004;lg=16.768277; lt=41.077286;md=480;ro=SSN-XG-
    IRI;sd=aaaaaaa; title="Humidity-Sensor-2030", </BitLineAnemomSens>;ct=0;ct=41;at
    =30004;lg=16.758347; lt=41.081983;md=500;ro=SSN-XG-IRI;sd=bbbbbbb; title="
    Anemometer-Sensor-111",
</H>;sd=ccccccc;sr=9.12
```

A. Yachir et al. [17] propose a resource directory model where the resources have four custom attributes: *entity*, *reference ontology*, *Service Quality of Service (QoS)*, and *Device QoS*. In a resource discovery request the attributes *entity* and *reference ontology* can acquire a value from a set of predefined values. The article presents an example for a user requesting temperature for kitchen with high energy level, very high reliability, a medium response time, a very low energy consumption, and a "matching threshold" of 0.6. The request looks like this

```
coap://addressRD? ent="kitchen"; rt="temperature"; dqos="Energy_level: high |
    Reliability: very high"; sqos="Response_Time:medium | Energy_Cost: low"; sr="0.6"
```

The response contains a description of the sensor of the following nature

```
/</pathRes1>;ep="Imote2Sensor";et="http://emp.org/Ontologies/Device.owl";ent="Kitchen"
    ;entro="http://emp.org/Ontologies/Space.owl";dqos="Energy_level:70:1|Reliability
    :0.6:1";rt="getTemperature|temperature|http://emp.org/Ontologies/"
```

This proposal meets most of the requirements of the use-case scenario. However, it does not address the scenario where the request could contain attributes with multiple values tied together with other attributes through a relational logic.

There has been a lot of work pertaining resource discovery in embedded systems in general, and CoAP in particular. The design goal of CoAP is to warrant the usage of low-end devices, both in the server and client roles, while keeping the functionality between them feature-rich. Unfortunately, the majority part of this research only focuses on the latter part, e.g. making *automatic* resource discovery in very large scale networks work. Although these techniques may still be used in a network of relatively smaller scale, the elements involved in the architecture of these solutions are often too big to be incorporated in practical scenarios. For example, the assumption that a small building's energy monitoring system has access to the same kind of data or information as the energy management system of a whole city does has practical limitations. This also applies to the processing power; the majority of the research gives little to no attention to the fact the most embedded servers are unable to run a full-fledged query language servers, such as SPARQL servers. Not only that, but it is often inaccurately assumed that all nodes in a network have access to internet or servers in an external network.

The network traffic related aspects, i.e. keeping the number and size of data transactions to a minimum, are mostly ignored. Although using CoAP as the primary communication protocol automatically mitigates these problems to an extent, it does not however eliminate

them completely. M. R. Khaefi et al. [18] present a solution to reduce message payloads pertaining to resource discovery among a large number of nodes. The approach, named CoAP-PBF, employs Partitioned Bloom Filters to keep record of each node's services. The paper shows that employing a technique as simple as using binary messages, instead of plain-text, decreases the payloads by orders of magnitude.

The proposal presented in this thesis will keep all these limitations in perspective while suggesting solutions to the problems as they surface.

CHAPTER 2

Thesis Contribution

The standard CoAP discovery procedure proved to be insufficient in practice for a number of use cases. This can easily be observed when looking at the use described in the scenario in Section 1.1. It was asserted that in order to fully satisfy the building owner's requirements, the resource directory needs to be able to process queries of the form 1.1. However, it was made evident in Section 1.1 that there is no standard way of processing the tokens that appear in the query expressions of this nature.

The symbols `|` and `&` represent logical operators `AND` and `OR` respectively. The usage of comparison / relational operators `<` (*smaller than*) and `>` (*bigger than*) in addition to `=` (*is equal to*) can be seen. The implementation proposed in this section will enhance the query processing capabilities of a standard CoAP server by adding parsing, processing and response handling for the query expressions of the form similar to 1.1.

2.1 Implementation

The implementation proposed in this thesis offers two CoAP URIs:

1. `POST /cbor` (with query expression as request body)
2. `GET /cbor/attr`

2.1.1 Architecture of `/cbor/`

Figure 2.1 shows an overview of the architecture of the first URI (`/cbor/`). Afterwards, each block's inner functionality will be explained.

Query Parser

The request to `/cbor/` starts with a `POST` call, with the query expression as the request body. The raw expression is fed to a query parser. The purpose of the query parser is to transform a complex expression into a form that could easily be processed by an engine

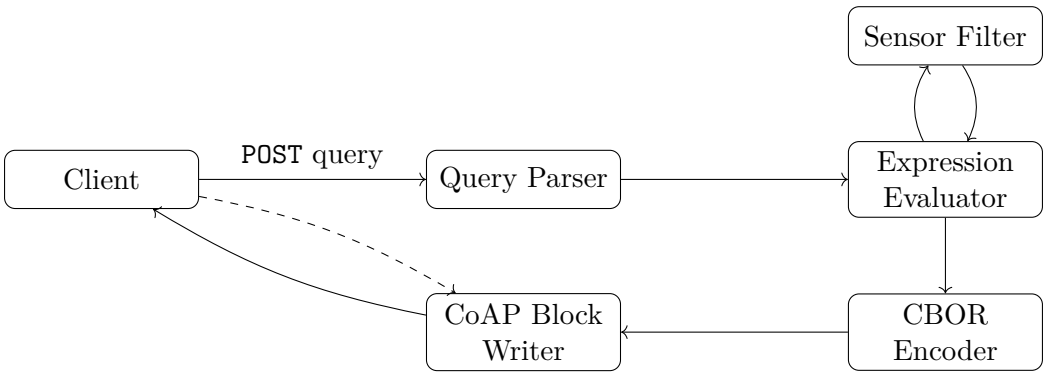


Figure 2.1: Request Handling For `/cbor/` URI

that is only capable of doing elementary operations with only two operands and a single operator, i.e. `Operand1 operator Operand2 = Result`.

In the example case where the client `POST`s expression 1.1, the job of the query parser is to lay it down into the following sequence of operations:

1. Evaluate `room<5`
2. Evaluate `floor=3`
3. `AND` the results of 1 and 2.
4. Evaluate `floor=11`
5. `AND` the results of 1 and 4.
6. `OR` the results of 3 and 5.
7. Evaluate `gain<2.1`
8. `AND` the results of 6 and 7.
9. Evaluate `room=1`
10. Evaluate `gain>3.3`
11. `AND` the results of 9 and 10.
12. `OR` the results of 8 and 11.
13. Return the result of 12.

The rearranged expression on the basis of the above sequence is shown here.

```
room<5 floor=3 & room<5 floor=11 & | gain<2.1 & room=1 gain>3.3 & |   (2.1)
```

To process expression 2.1 each sub-expression (starting from the left), like `room<5` or `gain>3.3`, is evaluated into a result. The sub-expression is then replaced with its evaluated result. This is continued until an operator appears in the sequence. When an operator appears, the corresponding operation is performed on the last two results, and then whole `(Operand1 Operand2 operator)` trio is replaced with the operation's result. The process is continued until the last operator in the expression, eventually reducing the whole expression into a single result.

Such an expression, where the operator appears after the two operands is called "Postfix notation" [19], as opposed to "Infix notation" where the operator lies between the two operands. Postfix notations are much easier to be processed by a simple computing engine because, while an infix notation requires parentheses to describe operator precedence, postfix notations do not need parentheses. For example, an infix expression "`(3 + 4) * 5`" is described in its postfix notation counterpart as "`3 4 + 5 *`". One such algorithm that computers use to transform these complex infix notations to simpler postfix notation is called Shunting-Yard algorithm [20]. The query parser in the implementation uses a modified form of the algorithm where the operands are sets instead of simple integers.

The "operands" in the expression 1.1 are not simple numbers. Instead, the result of a sub-expression, like `gain<2.1`, is a set like `{sensor13, sensor22, sensor34}`. Consequently, the operations `OR` and `AND` translate to Set-Union and Set-Intersection respectively. Terminal output from this block in one of the test runs is shown in 2.1.

```
(((room<5&floor=3)|(room<5&floor=11))&gain<2.1)|(room=1&gain>3.3) stack built with 13
    elements
0 1 & 2 3 & | 4 & 5 6 & |
0: room<5
1: floor=3
2: room<5
3: floor=11
4: gain<2.1
5: room=1
6: gain>3.3
```

Listing 2.1: Stdout From Query Parser

Sensor Filter

The input to this block is a sub-expression like `gain<2.1`, and the output from this block is a set of sensors that fulfil the criteria. Terminal output from this block in one of the test runs is shown in 2.2.

```
room<5 has 218 matches
floor=3 has 56 matches
room<5 has 218 matches
floor=11 has 76 matches
gain<2.1 has 454 matches
room=1 has 42 matches
gain>3.3 has 341 matches
```

Listing 2.2: Stdout From Sensor Filter

The set of filtered sensors is represented by a bitfield of 64 bits, where each bit represents a sensor. If a bit is set, the corresponding sensor is considered to have matched the sub-expression criteria. Non-matching sensors have their corresponding bits set to 0. As an example, imagine there are 73 sensors in total. A 64-bit bitfield can only accommodate 64 sensors, so we'll need another 64-bit bitfield to represent the remaining 9 sensors. Figure 2.2 shows how a resultant filtered set looks like:

Although these two bitfields could accommodate 128 sensors, having only a maximum of 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1	x	x	x	x	x	x	x
0	0	1	0	1	0	1	1	1	0	0	0	1	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1	1	0	1	1	1	0	0	1	1	0	1	1	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

1st bitfield 2nd bitfield

Figure 2.2: A Set of Sensors Returned By Sensor Filter

sensors in the system, the rest of the bits will be unused. If the number of sensors had been over 128, 3 bitfields would have been required. In general, the number of 64-bit bitfields, N_B, required for N_S number of sensors is:

$$N_B = \lceil \frac{N_S}{64} \rceil \tag{2.2}$$

There are two major advantages of using bitfields instead of ordinary integer arrays. The first being the better memory space efficiency. Instead of wasting 73 bytes for 73 sensors, assuming the smallest integer is 1 byte (8-bit) long, only 16 bytes are needed by using bitfields. The other advantage will be discussed shortly.

Expression Evaluator

The input to this block is a "stack", like 2.1, that was generated by the query parser previously. This block also forwards the sub-expressions to the "Sensor Filter" block. The sensor filter block returns the results in the form of an array of 64-bit long bitfields, as discussed in Section 2.1.1. Finally, this block performs the union and intersection operations on the returned sets.

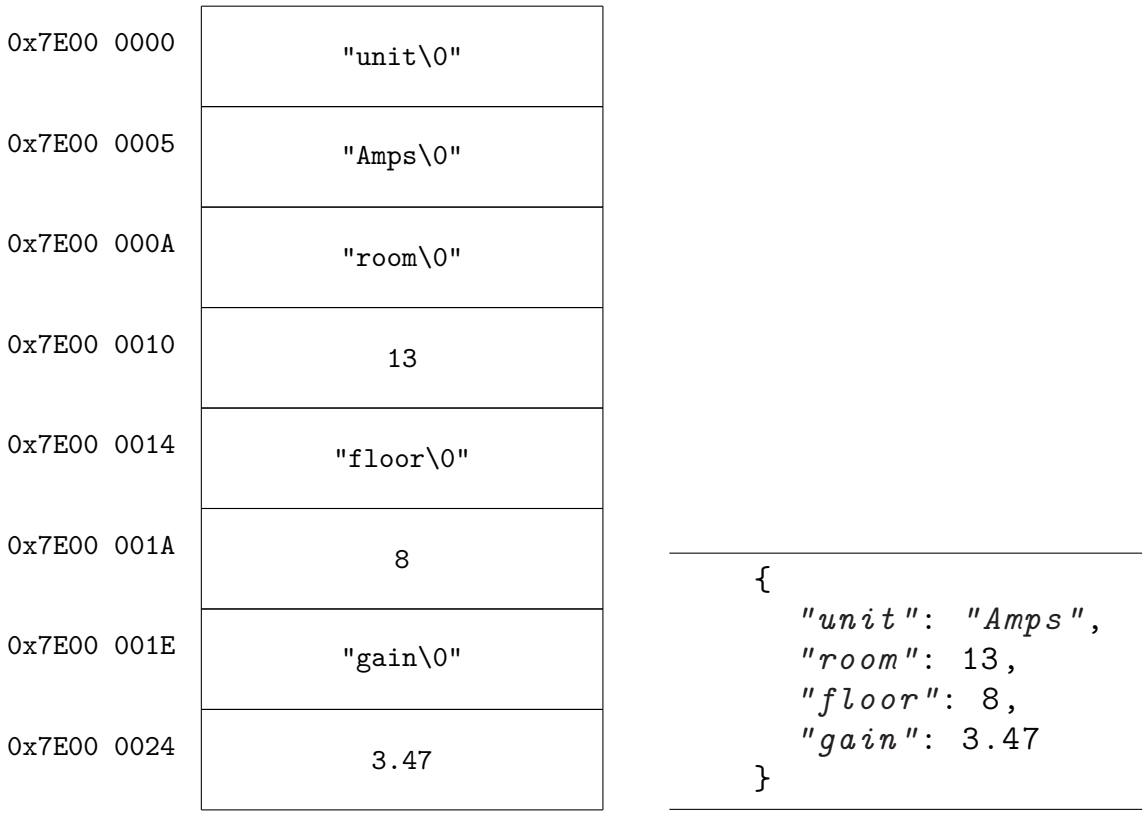
As mentioned earlier, there's more to the usage of bitfields over ordinary arrays than simply memory space efficiency. The bitfields, being 64-bit integers, are extremely easy to perform bitwise operations on. The set operations union and intersection simply translate to bitwise-or and bitwise-and in a programming language. The bitwise operations are done on two arrays of bitfields with a one-to-one correspondence.

After performing all of the set operations in the stack, the resultant bitfield array describes the sensors that match query 1.1.

CBOR Encoder

The resulting bitfield from the "Expression Evaluator" block only contains the flags representing the sensors, not the actual sensors data itself. Some methodology is needed to package all sensors into a data structure that can be returned in a CoAP response, and later be easily parsed by the client. This block needs to be able to return an array of sensors, with all their attribute names and values into one big hierarchical format.

The process of transforming a data structure from a computer program's memory into some



a) Sensor Layout in Memory

b) JSON Encoded Sensor

Figure 2.3: Serializing Sensor Data

form for later processing is called serialization. Figure 2.3 shows a sensor being serialized into the JSON.

JSON is probably the most commonly used serialization format in the web ecosystem. It is human-readable, can be hand written, has extensive software library support for encoding/parsing, and can represent any kind of structural hierarchy. However, these advantages come at the cost of verbosity. Each character in the JSON encoding takes up a whole byte. For example, the value of `room` attribute, i.e. `13`, is represented using two bytes: one byte for `1` and one byte for `3`. This value could have easily fit into a single byte if JSON could represent binary values, but it can't.

CBOR is a serialization format that fits perfectly for the use case scenario. CBOR allows for packaging hierarchical data, e.g. array of arrays of maps, into a representation that has very low encoding overhead, requires minimal code for encoding and decoding, and fully supports binary data. The binary representations used by CBOR are extremely small, resulting in much smaller payloads as compared to JSON. Even when compared to other binary serialization formats, CBOR has much shorter encodings. Shown in Listing 2.3 is the same sensor from Figure 2.3, this time in CBOR encoding:

```
A4 64 75 6E 69 74 64 41 6D 70 73 64 72 6F 6F 6D 0D 65 66 6C 6F 6F 72 08 64 67 61 69 6E
```

```
    FB 40 0B C2 8F 5C 28 F5 C3
```

Listing 2.3: CBOR Encoding of a Sensor

The byte count for CBOR encoding is 38 bytes, as compared to JSON encoding's 47 bytes. The difference of almost 10 bytes per sensor increases multiplicatively with a high number of sensors. Another thing to notice in the CBOR packet is that the `gain` attribute's value is a double-precision float value. That means even if the value had multiple digits after the decimal point, e.g. `3.47123123`, it would still have resulted in the same packet size, whereas in JSON it would have taken additional 6 bytes.

There are a lot of CBOR software libraries to choose from. The proposed implementation uses TinyCBOR (https://github.com/intel/tinycbor) due to its small code size and an easy to use C API.

CBOR allows creation of "indefinite length arrays", which is extremely useful when the total number of items in an array are unknown at the time of array creation/initialization. The input to "CBOR Encoder" block is the bitfield generated by "Expression Evaluator" block. The "CBOR Encoder" block could count the number of bits that are set to `1` to find out the required size of the array. But that would've resulted in two passes over the bitfield: one for counting bits, and one for using bits to put respective sensors into the array. Having the option to declare an array of indefinite length allows us to add sensors to it as the bitfield is being passed over the first time.

The final output of this block is CBOR encoded array of sensor structures with their attributes names and values.

CoAP Block Writer

CoAP imposes a limit on the PDU size of messages. According to [21] the reasons for this limit are as follows:

- To fit a message in a single UDP packet.
- To avoid IP fragmentation that is caused by breaking down a large message into smaller ones.
- To avoid adaptation-layer fragmentation for 6LoWPAN.

The output from the CBOR Encoder block could get large, if the filtered array contains many sensors. If the encoded packet gets larger than the allowed limit, it will have to be broken down into pieces and transmitted in several messages piece by piece. CoAP allows this by using Block-Wise Transfers [21].

The CoAP Block Writer block manages large-body payloads in the following fashion:

1. Wait for request.
2. Check for `Block2` option in the request header.
3. If there is no `Block2` option value go to the next step. If there is, set `N` equal to `Block2` value and go to step 5.
4. Initialize block and set `N`=0.

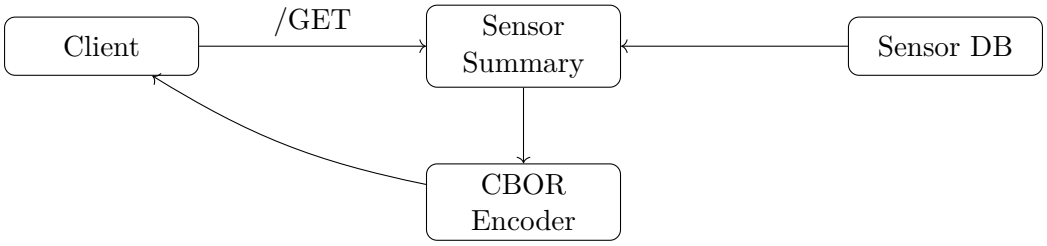


Figure 2.4: Request Handling For `/cbor/attr` URI

5. Send block number `N`, with `Block2` value set to `N` and `Size2` value set to the total size of the CBOR packet.
6. Go to step 1.

It is the job of the CoAP client to appropriately set `Block2` value with its next request whenever it sees `Block2` with the `M` bit set in the request header. It can also check the total size of the CBOR packet through `Size2` option of the response.

2.1.2 Architecture of `/cbor/attr`

Imagine a scenario where the technician from the company that installed the sensors is called for routine examination of the sensors. Now because the building owners are free to change the sensors configuration on their own, i.e. add new sensors, remove existing sensors, or change the location of sensors, the tech doesn't have the latest information on the building's state. It would be extremely helpful for him if he could get a summary of currently operational sensors' types and locations.

This URI is more of a utility than a core functional unit. The objective of this URI is to aid the user in creating informed queries. This URI presents a summary of all sensor attributes, which enables the user to see all possible values the attributes can have. It also informs if the attributes are read-only values or read-write values.

Figure 2.4 shows the birds-eye view of the second URI (`/cbor/attr`). The details of each block will follow afterwards.

Sensor DB

This block hosts all of the sensor data in its entirety. It is just a memory store where each sensor's latest `unit`, `room`, `floor` and `gain` value is available. For the sake of simplicity we'll assume that the stored data is up-to-date, even though it is statically defined and compiled in the implementation's source code.

In the proposed implementation a random sensor database is generated using the python script A.1. The script generates a C source file and a header file. The number of sensors to be generated is given as an argument to the script. 1000 sensors were generated with the following configuration:

- 28 unit types.

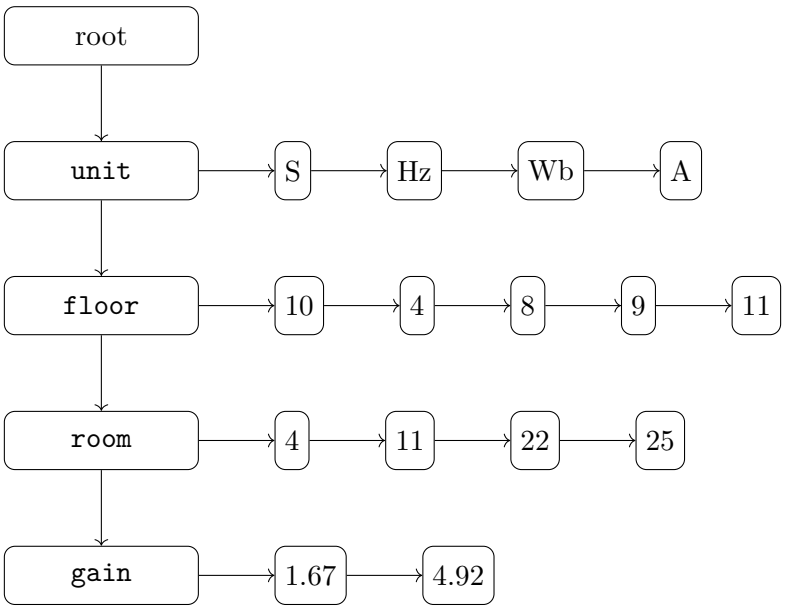


Figure 2.5: Depiction of Sensor Summary as Linked List of Linked Lists

- 15 floor levels.
- 25 rooms per floor.
- Gain values from 1.0 to 5.0.

Sensor Summary

This is the block where the sensor data is summarized and stored. The data structure is in the form of a linked list of linked lists. Shown in Table 2.1 is an example database of 5 sensors in the Sensor DB block.

Unit	Floor	Room	Gain
S	10	4	4.92
Hz	4	11	4.05
S	8	22	4.02
Wb	9	4	1.67
A	11	25	2.31

Table 2.1: Sensor DB Contents

The purpose of the data structure is to store only unique attribute values, preventing duplicates. Figure 2.5 shows the structure for the sensors from Table 2.1.

The main branch of the structure lists the attribute names. Each node then has a sub-branch that lists unique values for that particular attribute. Looking carefully, it can been seen that there are only 4 nodes in the `unit` sub-branch. This is because the unit type `S` appears twice in the sensor DB, but is only included once in linked list. Similarly, `room`

value 4 is included only once, resulting in 4 nodes.

The `gain` attribute, being a floating point value, is treated differently than string and numerical attributes. The data structure only stores the minimum and maximum values for floating point attributes. Such attributes have very distinct values for each sensor, and storing each value in the linked list will make it very large which defeats the purpose of a summary.

This list is generated only once, and only changes when anything in the sensor database changes.

CBOR Encoder

This block works similarly as the "CBOR Encoder" of the first URI as explained in Section 2.1.1, i.e. it converts a data structure from program's memory into a CBOR encoded packet. The JSON equivalent of the data structure of Figure 2.5 is show in Listing 2.4. The output of this block is similar, but in CBOR instead of JSON.

```
{
  "unit": {
    "wr": false,
    "type": "str",
    "vals": ["S","Hz","Wb","A"]
  },
  "floor": {
    "wr": false,
    "type": "int",
    "vals": [10,4,8,9,11]
  },
  "room": {
    "wr": false,
    "type": "int",
    "vals": [4,11,22,25]
  },
  "gain": {
    "wr": true,
    "type": "float",
    "vals": [1.67,4.92]
  }
}
```

Listing 2.4: Sensor Summary in JSON

2.1.3 URI response handling

As is apparent at this point that the output from the two URIs is a binary CBOR encoded packet which needs to be *decoded* at the UI (CoAP client) end. The library that used to encode binary data into CBOR at the server end, i.e. TinyCBOR, has all the functions to decode CBOR packets back to binary data as well.

The client is passed raw CBOR packet through its standard input, or *stdin* in Linux terminology, with the size of the packet as an argument. The client then allocates buffer for the raw packet, and copies the packet into it. Finally, a simple TinyCBOR function `cbor_value_to_pretty` is used to print the decoded data on standard output (*stdout*).

For the configuration with 1000 sensors, as described in section “Sensor DB”, Listing 2.5 shows the raw CBOR packet for `/cbor/attr` URI:

```
bf64756e6974bf627772f4647479706563737472647661 6c739f62487a61
4a626364627372626b67614e626c78614b6141637261646253766464656 7
43625061626c6d636f686d6156625762624779636d6f6c61546143624271
6146ffff65666c6f6f72bf627772f4647479706563696e746476616c739f
6138613661326231306231336231356137613961336135623131613462 31
34623132ffff64726f6f6dbf627772f4647479706563696e746476616c73
9f623232613462323362323061356231336231316231386231356133623 2
346231396231306232316232356231326138613961366137623137623136
623134ffff646761696ebf627772f5647479706565666c6f61746476616c
739f64312e303364352e3030ffffff
```

Listing 2.5: Raw CBOR Response From /cbor/attr

The corresponding decoded output at the client end is shown in Listing 2.6:

```
{
  "unit": {
    "wr": false,
    "type": "str",
    "vals": [
      "Hz", "J", "cd", "sr", "kg", "N", "lx", "K", "A", "rad", "Sv", "degC", "Pa",
      "lm", "ohm", "V", "Wb", "Gy", "mol", "T", "C", "Bq", "F"
    ]
  },
  "floor": {
    "wr": false,
    "type": "int",
    "vals": [
      "8", "6", "2", "10", "13", "15", "7", "9", "3", "5", "11", "4", "14", "12"
    ]
  },
  "room": {
    "wr": false,
    "type": "int",
    "vals": [
      "22", "4", "23", "20", "5", "13", "11", "18", "15", "3", "24", "19", "10",
      "21", "25", "12", "8", "9", "6", "7", "17", "16", "14"
    ]
  },
  "gain": {
    "wr": true,
    "type": "float",
    "vals": [
      "1.03",
      "5.00"
    ]
  }
}
```

Listing 2.6: Decoded Response For /cbor/attr

When the URI `/cbor/` is called with the query expression 1.1, the raw CBOR packet returned is shown in Listing 2.7:

```
9fa464756e6974614865666c6f6f720364726f6f6d01646761696efa4096
147ba464756e6974636f686d65666c6f6f720e64726f6f6d01646761696e
fa4081eb85a464756e69746065666c6f6f720b64726f6f6d02646761696e
fa3fd5c28fa464756e6974615465666c6f6f720264726f6f6d0164676169
6efa40847ae1a464756e6974615465666c6f6f720364726f6f6d01646761
```

```
696efa3f8ccccda464756e69746372616465666c6f6f720364726f6f6d01
646761696efa408f5c29a464756e6974616d65666c6f6f720264726f6f6d
01646761696efa4077ae14a464756e69746065666c6f6f720264726f6f6d
01646761696efa407147aea464756e6974614365666c6f6f720764726f6f
6d01646761696efa407851eca464756e6974614a65666c6f6f720464726f
6f6d01646761696efa4055c28fa464756e6974617365666c6f6f720b6472
6f6f6d01646761696efa4058f5c3a464756e69746065666c6f6f72036472
6f6f6d03646761696efa3fc147aea464756e697462537665666c6f6f7203
64726f6f6d01646761696efa3f90a3d7a464756e6974614e65666c6f6f72
0864726f6f6d01646761696efa408851eca464756e6974616d65666c6f6f
720a64726f6f6d01646761696efa4059999aa464756e697462576265666c
6f6f720764726f6f6d01646761696efa409f5c29a464756e697461566566
6c6f6f720b64726f6f6d03646761696efa3fb9999aa464756e6974626364
65666c6f6f720c64726f6f6d01646761696efa40900000a464756e697462
477965666c6f6f720364726f6f6d01646761696efa3f8e147ba464756e69
74636f686d65666c6f6f720f64726f6f6d01646761696efa4087ae14a464
756e6974626c6d65666c6f6f720964726f6f6d01646761696efa408eb852
a464756e697462477965666c6f6f720b64726f6f6d04646761696efa4000
a3d7a464756e6974614e65666c6f6f720964726f6f6d01646761696efa40
79999aff
```

Listing 2.7: Raw CBOR Response From /cbor

And the decoded data at the client end (with `gain` values shown until 2 decimal places) is shown in Listing 2.8:

```
[
  { "unit": "H", "floor": 3, "room": 1, "gain": 4.69 },
  { "unit": "ohm", "floor": 14, "room": 1, "gain": 4.05 },
  { "unit": "T", "floor": 2, "room": 1, "gain": 4.13 },
  { "unit": "T", "floor": 3, "room": 1, "gain": 1.1 },
  { "unit": "rad", "floor": 3, "room": 1, "gain": 4.48 },
  { "unit": "m", "floor": 2, "room": 1, "gain": 3.86 },
  { "unit": "C", "floor": 7, "room": 1, "gain": 3.88 },
  { "unit": "J", "floor": 4, "room": 1, "gain": 3.33 },
  { "unit": "s", "floor": 11, "room": 1, "gain": 3.39 },
  { "unit": "Sv", "floor": 3, "room": 1, "gain": 1.12 },
  { "unit": "N", "floor": 8, "room": 1, "gain": 4.26 },
  { "unit": "m", "floor": 10, "room": 1, "gain": 3.4 },
  { "unit": "Wb", "floor": 7, "room": 1, "gain": 4.98 },
  { "unit": "V", "floor": 11, "room": 3, "gain": 1.45 },
  { "unit": "cd", "floor": 12, "room": 1, "gain": 4.5 },
  { "unit": "Gy", "floor": 3, "room": 1, "gain": 1.11 },
  { "unit": "ohm", "floor": 15, "room": 1, "gain": 4.23 },
  { "unit": "lm", "floor": 9, "room": 1, "gain": 4.46 },
  { "unit": "Gy", "floor": 11, "room": 4, "gain": 2 },
  { "unit": "N", "floor": 9, "room": 1, "gain": 3.9 }
]
```

Listing 2.8: Decoded Response from /cbor/

2.2 Experiments: Correctness

This section documents the experimentation that will be used to validate the usefulness of the proposed resource discovery model. In these experiments, the model is expected to serve discovery results based on complex query expressions with 100% correctness. In other words, the sensors in the returned array are expected to fulfil all the criteria described in the query expression. Furthermore, the returned sensor array is expected to contain all the sensors that fit the query criteria.

A set of three experiments was done for testing the proposed implementation under different sensor configurations. The following steps are taken for each experiment.

1. Generate a new sensor configuration, using script A.1, for 100 sensors.
2. Run the following queries:
   a) Show all sensors that are between floors 5 and 10 (5 and 10 included) and are in room 7 or 8, or all the sensors that are below floor 5 and are in room 1 or 3, or all the sensors that are above floor 10 and are in room 9. The query expression for this query is:

```
((floor>4&floor<11)&(room=7|room=8))|(floor<5&(room=1|room=3))|(floor
    >10&room=9)
```

   b) Show all the sensors that are on the first 8 floors with gain either smaller than 1.5 or greater than 3.5, or all the sensors that are in room 10 of first 8 floors with gain between 1.5 and 3.5. The query expression for this query is:

```
floor<9&((gain<1.5|gain>3.5)|(room=10&(gain>1.5&gain<3.5)))
```

3. Verify that all the sensors returned fulfil the criteria, and mark the ones that don't.

Instead of actual CBOR binary packets, the results for both queries for all three experiments are listed below in JSON for convenience.

2.2.1 Experiment 1

Test Query 1

```
[
  {"unit": "Gy", "floor": 6, "room": 8, "gain": 1.12},
  {"unit": "sr", "floor": 13, "room": 9, "gain": 3.84},
  {"unit": "Hz", "floor": 11, "room": 9, "gain": 1.09},
  {"unit": "C", "floor": 1, "room": 1, "gain": 2.06},
  {"unit": "Gy", "floor": 5, "room": 7, "gain": 2.29},
  {"unit": "V", "floor": 10, "room": 7, "gain": 3.40},
  {"unit": "lm", "floor": 8, "room": 7, "gain": 2.25}
]
```

Test Query 2

```
[
  {"unit": "S", "floor": 1, "room": 2, "gain": 4.78},
  {"unit": "s", "floor": 4, "room": 23, "gain": 4.84},
  {"unit": "Gy", "floor": 6, "room": 8, "gain": 1.12},
  {"unit": "H", "floor": 3, "room": 12, "gain": 4.80},
  {"unit": "Pa", "floor": 5, "room": 2, "gain": 4.98},
  {"unit": "Wb", "floor": 3, "room": 5, "gain": 1.04},
  {"unit": "Bq", "floor": 1, "room": 17, "gain": 3.57},
  {"unit": "J", "floor": 1, "room": 12, "gain": 4.21},
  {"unit": "T", "floor": 6, "room": 9, "gain": 1.04},
  {"unit": "W", "floor": 5, "room": 1, "gain": 4.13},
```

```
  {"unit": "Sv", "floor": 3, "room": 5, "gain": 3.73},
  {"unit": "Hz", "floor": 4, "room": 23, "gain": 4.30},
  {"unit": "lm", "floor": 7, "room": 10, "gain": 2.98},
  {"unit": "cd", "floor": 8, "room": 12, "gain": 3.72},
  {"unit": "Hz", "floor": 3, "room": 11, "gain": 4.59},
  {"unit": "mol", "floor": 4, "room": 14, "gain": 4.23},
  {"unit": "Pa", "floor": 7, "room": 20, "gain": 4.11},
  {"unit": "Gy", "floor": 8, "room": 2, "gain": 3.70},
  {"unit": "ohm", "floor": 3, "room": 25, "gain": 4.80},
  {"unit": "mol", "floor": 4, "room": 25, "gain": 4.96},
  {"unit": "s", "floor": 8, "room": 16, "gain": 3.80},
  {"unit": "T", "floor": 2, "room": 13, "gain": 4.32},
  {"unit": "cd", "floor": 2, "room": 2, "gain": 3.96},
  {"unit": "m", "floor": 3, "room": 21, "gain": 4.25},
  {"unit": "sr", "floor": 5, "room": 19, "gain": 1.05},
  {"unit": "rad", "floor": 5, "room": 18, "gain": 4.44},
  {"unit": "C", "floor": 3, "room": 7, "gain": 1.16},
  {"unit": "T", "floor": 3, "room": 4, "gain": 4.28},
  {"unit": "s", "floor": 6, "room": 19, "gain": 3.90},
  {"unit": "K", "floor": 2, "room": 4, "gain": 4.94},
  {"unit": "Pa", "floor": 2, "room": 14, "gain": 3.81}
]
```

2.2.2 Experiment 2

Test Query 1

```
[
  {"unit": "s", "floor": 2, "room": 3, "gain": 2.33},
  {"unit": "m", "floor": 14, "room": 9, "gain": 4.65},
  {"unit": "C", "floor": 3, "room": 1, "gain": 4.48},
  {"unit": "rad", "floor": 12, "room": 9, "gain": 1.30},
  {"unit": "H", "floor": 10, "room": 8, "gain": 3.28},
  {"unit": "ohm", "floor": 8, "room": 7, "gain": 3.95}
]
```

Test Query 2

```
[
  {"unit": "Wb", "floor": 8, "room": 17, "gain": 4.78},
  {"unit": "rad", "floor": 7, "room": 15, "gain": 1.42},
  {"unit": "A", "floor": 3, "room": 5, "gain": 3.65},
  {"unit": "lm", "floor": 8, "room": 14, "gain": 3.89},
  {"unit": "", "floor": 4, "room": 22, "gain": 1.11},
  {"unit": "Gy", "floor": 6, "room": 9, "gain": 4.34},
  {"unit": "sr", "floor": 6, "room": 18, "gain": 4.80},
  {"unit": "F", "floor": 5, "room": 6, "gain": 4},
  {"unit": "C", "floor": 3, "room": 1, "gain": 4.48},
  {"unit": "C", "floor": 7, "room": 22, "gain": 4.07},
  {"unit": "", "floor": 8, "room": 21, "gain": 1.34},
  {"unit": "H", "floor": 6, "room": 21, "gain": 3.67},
  {"unit": "Pa", "floor": 6, "room": 9, "gain": 1.27},
  {"unit": "S", "floor": 2, "room": 23, "gain": 4.67},
  {"unit": "S", "floor": 5, "room": 1, "gain": 4.86},
  {"unit": "Wb", "floor": 5, "room": 21, "gain": 4.88},
  {"unit": "cd", "floor": 5, "room": 23, "gain": 4.61},
  {"unit": "V", "floor": 4, "room": 17, "gain": 1.32},
  {"unit": "kg", "floor": 2, "room": 4, "gain": 4.48},
```

```
  {"unit": "cd", "floor": 8, "room": 20, "gain": 1.37},
  {"unit": "Bq", "floor": 8, "room": 15, "gain": 3.83},
  {"unit": "ohm", "floor": 8, "room": 7, "gain": 3.95}
]
```

2.2.3 Experiment 3

Test Query 1

```
[
  {"unit": "V", "floor": 4, "room": 3, "gain": 2.23},
  {"unit": "ohm", "floor": 11, "room": 9, "gain": 3.59},
  {"unit": "F", "floor": 10, "room": 7, "gain": 1.29},
  {"unit": "lm", "floor": 4, "room": 3, "gain": 3.20},
  {"unit": "F", "floor": 9, "room": 8, "gain": 3.41}
]
```

Test Query 2

```
[
  {"unit": "S", "floor": 3, "room": 7, "gain": 1.45},
  {"unit": "Wb", "floor": 7, "room": 1, "gain": 4.21},
  {"unit": "cd", "floor": 5, "room": 10, "gain": 4.51},
  {"unit": "T", "floor": 7, "room": 4, "gain": 4.94},
  {"unit": "rad", "floor": 8, "room": 11, "gain": 4.44},
  {"unit": "J", "floor": 5, "room": 11, "gain": 3.83},
  {"unit": "C", "floor": 3, "room": 25, "gain": 4.61},
  {"unit": "Sv", "floor": 2, "room": 2, "gain": 3.95},
  {"unit": "", "floor": 3, "room": 15, "gain": 1.42},
  {"unit": "K", "floor": 1, "room": 16, "gain": 4.38},
  {"unit": "C", "floor": 4, "room": 23, "gain": 3.84},
  {"unit": "C", "floor": 3, "room": 9, "gain": 4.19},
  {"unit": "W", "floor": 8, "room": 15, "gain": 3.61},
  {"unit": "A", "floor": 1, "room": 9, "gain": 1.25},
  {"unit": "Gy", "floor": 5, "room": 11, "gain": 3.60},
  {"unit": "F", "floor": 3, "room": 20, "gain": 4.28},
  {"unit": "F", "floor": 5, "room": 25, "gain": 4.32},
  {"unit": "Sv", "floor": 8, "room": 12, "gain": 1.28},
  {"unit": "V", "floor": 7, "room": 3, "gain": 4.98},
  {"unit": "H", "floor": 8, "room": 24, "gain": 1.20},
  {"unit": "Sv", "floor": 1, "room": 16, "gain": 1.40}]
```

2.3 Experiments: Performance

Another set of experiments was done for gauging the new implementation's performance statistics. The objective was to visualize how the response time of the CoAP server is affected by increasing number of sensors, and also by increasing the complexity of the query expression. For these experiments the server was deployed in a machine in Oregon USA, and the client machine was in Braunschweig Germany. For each performance statistic 100 measurements were taken. Appendix A.4 lists the script primarily used to capture the measurements.

2.3.1 Experiment 4

The following steps were taken for this experiment. N ranges from 10 to 10,000.

1. Generate N number of sensors for the configuration.
2. Run query from Listing 2b.
3. Measure response time and the total size of the response.
4. Increase N, and go to step 1.

2.3.2 Experiment 5

The following steps were taken for this experiment. The server was forced to respond with a fixed size response of 4 bytes for each query. N ranges from 50,000 to 1,000,000.

1. Generate N number of sensors for the configuration.
2. Run query from Listing 2b.
3. Measure response time.
4. Increase N, and go to step 1.

2.3.3 Experiment 6

The following steps were taken for this experiment. The number of total sensors was kept at 1,000,000. The server was forced to respond with a fixed size response of bytes for each query.

1. Run a query expression of a specific complexity.
2. Measure response time.
3. Change query expression with a greater complexity, and go to step 1.

The following query expressions were evaluated:

1. `floor<9`
2. `gain<1.5|gain>3.5`
3. `gain>1.5&gain<3.5`
4. Full expression from Listing 2b

2.3.4 Experiment 7

For sensor configurations of 10 to 100,000 sensors, the query expression `gain>0.5` was used to target 92% of the resources[1]. Responses from the following server types were captured:

1. Standard CoAP server returning JSON response
2. Proposed implementation returning JSON response
3. Proposed implementation returning CBOR response

[1]The `gain` attribute ranges from 0.1 to 5.0, distributed uniformly.

CHAPTER 3

Thesis Outcome

In Section 1.1 a use case scenario was discussed. The use case put forward several requirements, or specifications, that the standard CoAP resource discovery mechanism was unable to satisfy. The resource discovery methodology proposed in this thesis is able to discover resources by their specific attribute values. This is especially useful in scenarios where the resources have arbitrary attributes, and are not easily defined using standard / popular interfaces and resource types such as described by Internet Assigned Numbers Authority (IANA) [22]. The proposed methodology allows CoAP client nodes to tailor the discovery criteria in an expressive way through the usage of logical comparison operators, which is impossible to do in standard discovery procedure.

In Chapter 2 it was discussed that it is still possible to do attribute-level filtering using standard discovery procedure, but how it comes at a heavy cost of data overheads and power wastage. The objectives of the proposed methodology is to eliminate, or at the very least minimize, those costs, all while catering to the requirements of the use case scenario. As a summary, the following are the objectives the proposed methodology aims for:

- Resource discovery using attribute-level filtering.
- Reduction in number of data transactions.
- Reduction in size of data transactions.
- Reduction in radio power consumption.
- Reduction in processing cost.

To work towards the listed objectives, the proposed methodology defines two server-side URIs. These URIs can be used to discover resources using complex query expressions that target the resources' attribute values. These URIs can be used by standard CoAP clients after an addition of CBOR decoding layer. The use case describes sensors that have only four attributes, including attributes with string and floating point values. However, the proposed implementation is designed to work with any number of attributes.

3.1 Evaluation

In Chapter 2 it was shown that server-side filtering is more scalable than client-side filtering. It was was argued that if server is able to serve the requested queries with correct results, all of the above stated objectives will be delivered. The purpose of this section is to evaluate if the server was able to accurately return requested data based on experimentation documented in Section 2.2.

The results of the experiments are listed in JSON for ease of processing using common tools. For example, listings 2.2.1 and 2.2.1 show the results for experiment 1 for the two test queries mentioned at 2a and 2b respectively. Similarly, the sensor configuration used in experiment 1 is listed, again in JSON, at A.2. For verification, a Javascript snippet A.3 was written to import sensor configuration A.2 and run the test queries on it. The results from Javascript snippet were then matched with the results from listings 2.2.1 and 2.2.1. This snippet was run in a Node.js [23] environment. The following is the output of the snippet:

```
Test Query 1 Results
{ unit: 'Gy', floor: 6, room: 8, gain: 1.12 }
{ unit: 'sr', floor: 13, room: 9, gain: 3.84 }
{ unit: 'Hz', floor: 11, room: 9, gain: 1.09 }
{ unit: 'C', floor: 1, room: 1, gain: 2.06 }
{ unit: 'Gy', floor: 5, room: 7, gain: 2.29 }
{ unit: 'V', floor: 10, room: 7, gain: 3.4 }
{ unit: 'lm', floor: 8, room: 7, gain: 2.25 }
Test Query 2 Results
{ unit: 'S', floor: 1, room: 2, gain: 4.78 }
{ unit: 's', floor: 4, room: 23, gain: 4.84 }
{ unit: 'Gy', floor: 6, room: 8, gain: 1.12 }
{ unit: 'H', floor: 3, room: 12, gain: 4.8 }
{ unit: 'Pa', floor: 5, room: 2, gain: 4.98 }
{ unit: 'Wb', floor: 3, room: 5, gain: 1.04 }
{ unit: 'Bq', floor: 1, room: 17, gain: 3.57 }
{ unit: 'J', floor: 1, room: 12, gain: 4.21 }
{ unit: 'T', floor: 6, room: 9, gain: 1.04 }
{ unit: 'W', floor: 5, room: 1, gain: 4.13 }
{ unit: 'Sv', floor: 3, room: 5, gain: 3.73 }
{ unit: 'Hz', floor: 4, room: 23, gain: 4.3 }
{ unit: 'lm', floor: 7, room: 10, gain: 2.98 }
{ unit: 'cd', floor: 8, room: 12, gain: 3.72 }
{ unit: 'Hz', floor: 3, room: 11, gain: 4.59 }
{ unit: 'mol', floor: 4, room: 14, gain: 4.23 }
{ unit: 'Pa', floor: 7, room: 20, gain: 4.11 }
{ unit: 'Gy', floor: 8, room: 2, gain: 3.7 }
{ unit: 'ohm', floor: 3, room: 25, gain: 4.8 }
{ unit: 'mol', floor: 4, room: 25, gain: 4.96 }
{ unit: 's', floor: 8, room: 16, gain: 3.8 }
{ unit: 'T', floor: 2, room: 13, gain: 4.32 }
{ unit: 'cd', floor: 2, room: 2, gain: 3.96 }
{ unit: 'm', floor: 3, room: 21, gain: 4.25 }
{ unit: 'sr', floor: 5, room: 19, gain: 1.05 }
{ unit: 'rad', floor: 5, room: 18, gain: 4.44 }
{ unit: 'C', floor: 3, room: 7, gain: 1.16 }
{ unit: 'T', floor: 3, room: 4, gain: 4.28 }
{ unit: 's', floor: 6, room: 19, gain: 3.9 }
{ unit: 'K', floor: 2, room: 4, gain: 4.94 }
{ unit: 'Pa', floor: 2, room: 14, gain: 3.81 }
```

The output matches completely with the experiments' results. This shows that all of the returned sensors in results 2.2.1 and 2.2.1 fulfil the criteria of the test query expressions. It

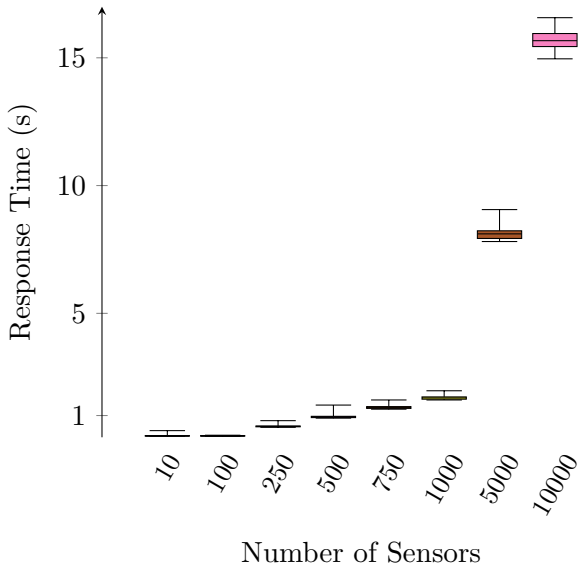


Figure 3.1: Response Time for Filtered Data versus Number of Sensors

also shows that all sensors fulfilling the criteria are returned.

Experimentation done in Section 2.3 showed that the implementation scales better than the standard CoAP discovery in every scenario.The results for the experiment performed in Section 2.3.1 is shown in Figure 3.1 as a box plot. The response time scales linearly with the number of sensors. Increasing number of sensors results in more filtered sensors per query. This in turn increases the total size of data to be returned per query. Figure 3.2 shows number of bytes returned for each configuration with a different number of sensors as a log-log plot.

In order to better visualize server's performance degradation with increasing number of sensors, it is necessary to normalize the data with the number of response packets returned. The maximum packet size was set to 1024 bytes for the server. Figure 3.3 shows response times for 100 packets with increasing number of total sensors as a box plot. Server's response time in terms of pure processing period stays constant for a realistic[1] number of total sensors.

The experiment done in Section 2.3.1 showed that the server's performance was not hindered as the number of sensors was increased from 10 to 10,000. The objective of the experiment performed in Section 2.3.2 was to verify if the results from Section 2.3.1 hold true for very large number of sensors. To eliminate the response time variations due to network related variables, i.e. response size, the server was modified to sent a fixed 4 character response for each configuration. The results are shown in Figure 3.4. It can be seen that for very large number of sensors, the proposed server's response time starts to increase slightly.

[1]Realistic in terms of server machine's computing power. The machine used in the experiments was a single-core 3.0GHz processor, 1GB RAM Amazon Web Services (AWS) t2.micro Virtual Private Server

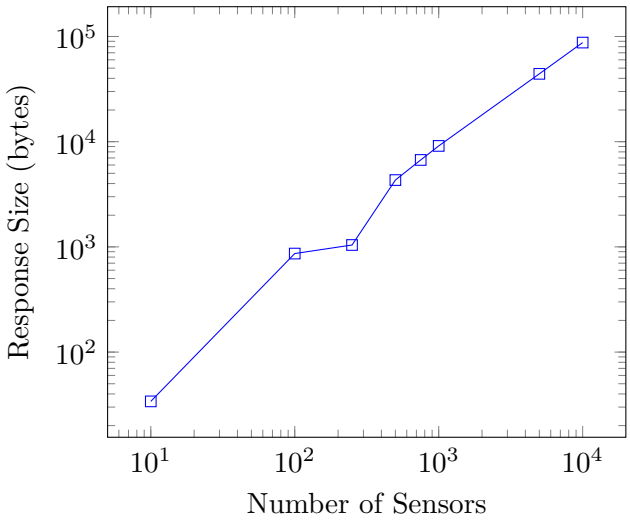


Figure 3.2: Size of Response versus Number of Sensors

The objective of the experiment performed in Section 2.3.3 was to measure server's performance degradation with increasing complexity of the query expression. The results plotted in Figure 3.5 show response times for each query. The server's response time does appear to increase with the increasing complexity of the query expression.

This experiment done in Section 2.3.4 visualizes the data efficiency of the proposed implementation. Because the standard CoAP server does not have the filtering capability, it will simply return all of the sensors in the database. The results shown in Figure 3.6 prove that the proposed implementation performs much better with CBOR as the encoding scheme. Responses were nearly half in size as compared to the standard CoAP server. Using CBOR instead of JSON saved more than 45% in the network traffic.

3.2 Conclusion

In this thesis, a new approach towards CoAP discovery is proposed. It is asserted that the standard CoAP discovery mechanism proves to be insufficient or even incapable in certain scenarios. This argument is strengthened by postulating a real world use-case scenario. It is shown that to effectively fulfil the use-case requirements, some way of filtering the discovered resources is needed. This filtering mechanism is implemented using fast bit-wise representations of resources, and a query parsing engine that supports complex query expressions. The implementation uses CBOR as the encoding scheme which is shown to be superior to Constrained RESTful Environments (CoRE) Link Format, as well as to other encoding schemes such as JSON and other binary encodings. The implementation's correctness and performance superiority is verified using a number of experiments. The outcome of this thesis is a discovery model that can be used in real world applications that need fine-grained resource discovery based on attributes that can have values of types strings, integers and floating-point numbers.

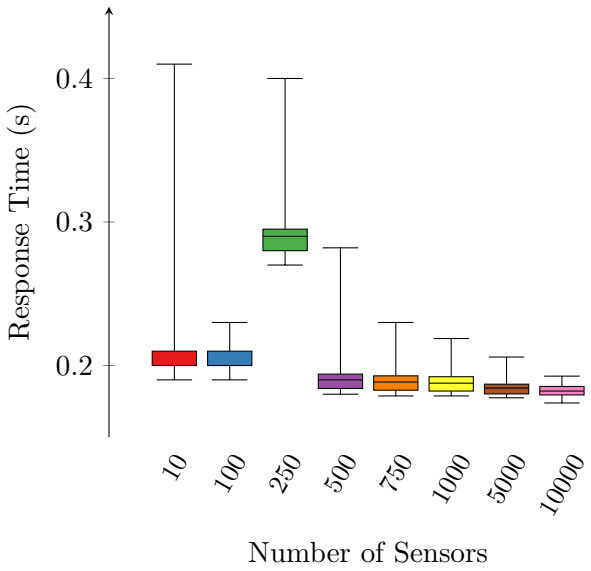


Figure 3.3: Normalized Response Time for Filtered Data versus Number of Sensors

3.3 Future Work

The proposed implementation can be improved in a few ways to make it more performant for capable machines. For instance, machines with more than one cores could be able to improve response times if the implementation supported parallelism. In expression 2.1, the sub-expressions `room<5`, `floor=3`, `floor=11`, `gain<2.1`, `room=1` and `gain>3.3` can be evaluated in parallel in different cores without affecting the final result.

Machines with memory to spare could take advantage of a caching layer. Referring back to expression 2.1, it can be seen that sub-expression `room<5` appears twice. A meticulously designed caching layer that stores results of last N sub-expressions can greatly enhance response times especially in a system with hundreds of thousands of sensors.

The query parser engine can be improved to support relational operators `>=` (*bigger than or equal to*), `<=` (*smaller than or equal to*), and `!=` (*not equal to*). Furthermore, it could be useful to support all operators for string attributes.

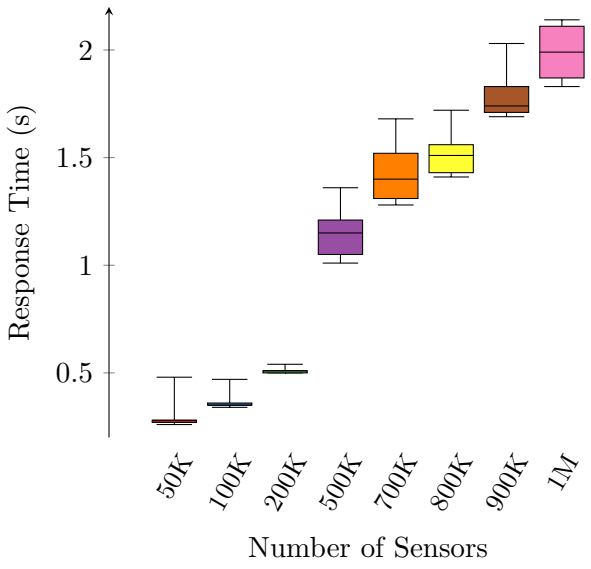


Figure 3.4: Response Time for A Fixed Response Size versus Number of Sensors

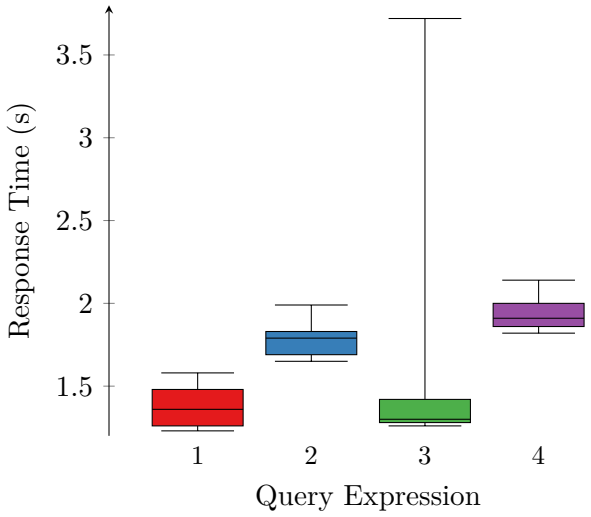


Figure 3.5: Response Time vs Query Expression

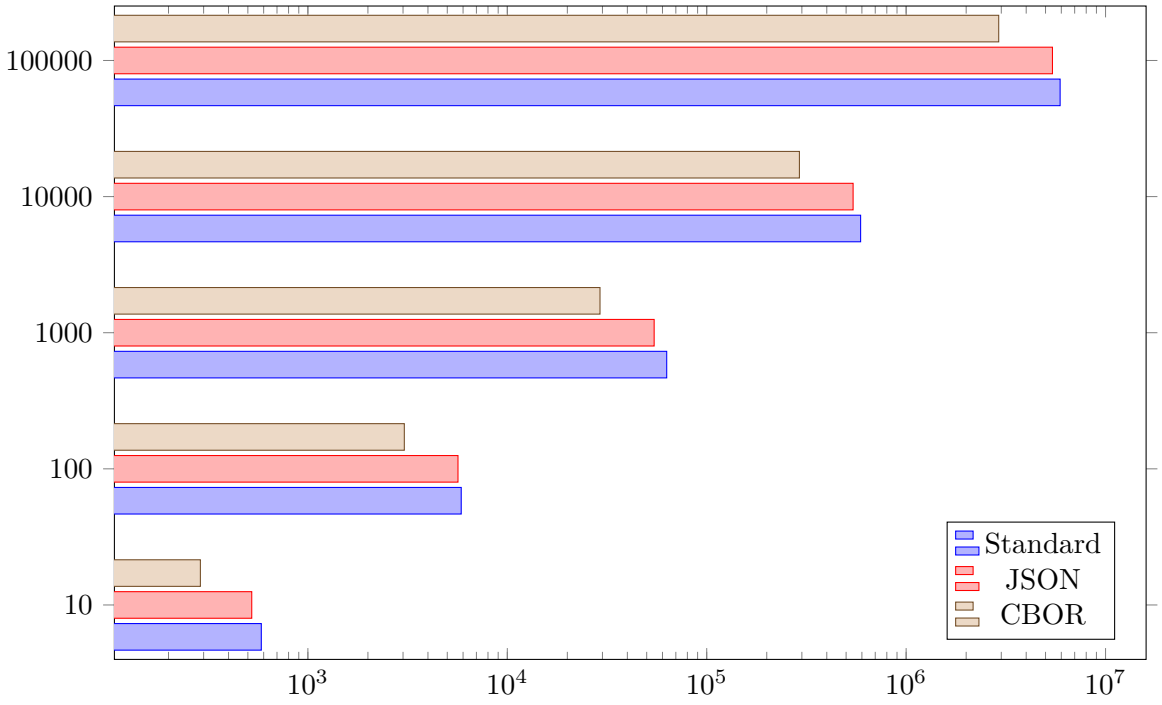


Figure 3.6: Response Sizes for Different Server Types

Bibliography

[1] T. Socolofsky and C. Kale. A TCP/IP Tutorial. `https://tools.ietf.org/html/rfc1180`. [Online; accessed 14-Feb-2019].

[2] R. Kalin. A Simplified NCP Protocol. `https://tools.ietf.org/html/rfc60`. [Online; accessed 14-Feb-2019].

[3] R. Fielding, J. Gettys, J. Mogul, F. Frystyk, L. Masinter, P. Leach, and T. Berners Lee. Hypertext Transfer Protocol – HTTP/1.1. `https://tools.ietf.org/html/rfc1180`. [Online; accessed 17-Feb-2019].

[4] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7, chapter Representational State Transfer (REST). University of California, Irvine Irvine, USA, 2000.

[5] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of Historical Trends in the Electrical Efficiency of Computing. *IEEE Annals of the History of Computing*, 33(3):46–54, March 2011.

[6] Gordon Bell. Moore's Law evolved the PC industry; Bell's Law disrupted it with players, phones, and tablets: New Platforms, tools, and sevices. Technical report, January 2014.

[7] J. Postel. User Datagram Protocol. `https://tools.ietf.org/html/rfc768`. [Online; accessed 17-Feb-2019].

[8] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). `https://tools.ietf.org/html/rfc7252`. [Online; accessed 18-Feb-2019].

[9] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, and B. Raymor. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. `https://tools.ietf.org/html/rfc8323`. [Online; accessed 18-Feb-2019].

[10] Z. Shelby, M. Koster, C. Bormann, P. van der Stok, and C Amsuess. CoRE Resource Directory. `https://tools.ietf.org/html/draft-ietf-core-resource-directory-19`. [Online; accessed 25-Jan-2019].

[11] Z. Shelby, M. Koster, C. Groves, J. Zhu, and B Silverajan. Reusable Interface Definitions for Constrained RESTful Environments. `https://tools.ietf.org/html/draft-ietf-core-interfaces-13`. [Online; accessed 25-Jan-2019].

[12] D. Pfisterer, K. Romer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. Kröller, M. Pagel, M. Hauswirth, M. Karnstedt, M. Leggieri, A. Passant, and R. Richardson. Spitfire: toward a semantic web of things. *IEEE Communications*

Magazine, 49(11):40–48, November 2011.

[13] Perera Charith, Zaslavsky Arkady, Liu Chi Harold, Compton Michael, Christen Peter, and Georgakopoulos Dimitrios. Sensor search techniques for sensing as a service architecture for the internet of things. *IEEE Sensors Journal*, 14, 09 2013.

[14] Michele Ruta, Floriano Scioscia, Agnese Pinto, Filippo Gramegna, Saverio Ieva, Giuseppe Loseto, and Eugenio Di Sciascio. A coap-based framework for collaborative sensing in the semantic web of things. *Procedia Computer Science*, 109:1047 – 1052, 2017. 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.

[15] Michele Ruta, Floriano Scioscia, A Pinto, Eugenio Di Sciascio, Filippo Gramegna, Saverio Ieva, and Giuseppe Loseto. Resource annotation, dissemination and discovery in the semantic web of things: A coap-based framework. pages 527–534, 08 2013.

[16] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Filippo Gramegna, A Pinto, Saverio Ieva, and Eugenio Di Sciascio. A logic-based coap extension for resource discovery in semantic sensor networks. *CEUR Workshop Proceedings*, 904:17–32, 01 2012.

[17] Ali Yachir, Badis Djamaa, Kheireddine Zeghouani, Marwen Bellal, and Mohammed Boudali. Semantic resource discovery with coap in the internet of things. pages 75–82, 01 2017.

[18] M. R. Khaefi and D. Kim. Bloom filter based coap discovery protocols for distributed resource constrained networks. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 448–453, July 2015.

[19] Arthur W. Burks, Don W. Warren, and Jesse B. Wrights. An analysis of a logical machine using parenthesis-free notation. mathematical tables and other aids to computation, vol. 8 (1954), pp. 53–57. *Journal of Symbolic Logic*, 20, 1955.

[20] An algol 60 translator for the x1. *Annual Review in Automatic Programming*, 3:329 – 345, 1963. Annual Review in Automatic Programming.

[21] C. Bormann and Z. Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). `https://tools.ietf.org/html/rfc7959`. [Online; accessed 30-Jan-2019].

[22] Constrained RESTful Environments (CoRE) Parameters. `https://www.iana.org/assignments/core-parameters/core-parameters.xhtml`. [Online; accessed 10-Feb-2019].

[23] About | Node.js. `https://nodejs.org/en/about/`. [Online; accessed 11-Feb-2019].

Appendix

A.1 gen_sensors.py

```
#!/usr/bin/python

import sys
from random import randint
from random import uniform

nsen = int(sys.argv[1])
units = ["m","kg","s","A","K","mol","cd","rad","sr","Hz","N","Pa","J","W",
         "degC","C","V","ohm","S","F","H","Wb","T","lm","lx","Bq","Gy","Sv"]
floors = 15
rooms = 25
gains = 5.0

units_len = len(units)

f = open('server/include/cfg.h','w')
f.write('#define NSEN %s\n' % (nsen))
f.close()

f = open('server/src/cfg.c','w')

f.write('#include "cfg.h"\n')
f.write('#include "sensor.h"\n')

f.write('char *units[NSEN] = {\n')
for i in range(nsen):
    r = randint(0,units_len-1)
    f.write('    "%s",\n' % (units[r]))
f.write('};\n')

f.write('char *floors[NSEN] = {\n')
for i in range(nsen):
    r = randint(1,floors)
    f.write('    "%s",\n' % (r))
f.write('};\n')

f.write('char *rooms[NSEN] = {\n')
for i in range(nsen):
    r = randint(1,rooms)
    f.write('    "%s",\n' % (r))
f.write('};\n')

f.write('char *gains[NSEN] = {\n')
for i in range(nsen):
    r = round(uniform(0.1,gains),2)
    f.write('    "%s",\n' % (r))
```

```
f.write('};\n')
f.close();
```

"Script for generating random sensor database"

A.2 Sensor Configuration


```
[
  {"unit": "m", "floor": 9, "room": 9, "gain": 4.90},
  {"unit": "N", "floor": 8, "room": 20, "gain": 3.04},
  {"unit": "S", "floor": 1, "room": 2, "gain": 4.78},
  {"unit": "s", "floor": 4, "room": 23, "gain": 4.84},
  {"unit": "Gy", "floor": 6, "room": 8, "gain": 1.12},
  {"unit": "Wb", "floor": 13, "room": 6, "gain": 2.28},
  {"unit": "m", "floor": 15, "room": 18, "gain": 4.65},
  {"unit": "H", "floor": 3, "room": 12, "gain": 4.80},
  {"unit": "sr", "floor": 5, "room": 6, "gain": 2.64},
  {"unit": "Pa", "floor": 4, "room": 9, "gain": 2.50},
  {"unit": "F", "floor": 15, "room": 18, "gain": 2.18},
  {"unit": "cd", "floor": 10, "room": 22, "gain": 3.57},
  {"unit": "Pa", "floor": 5, "room": 2, "gain": 4.98},
  {"unit": "Gy", "floor": 7, "room": 4, "gain": 2.25},
  {"unit": "sr", "floor": 13, "room": 9, "gain": 3.84},
  {"unit": "Wb", "floor": 3, "room": 5, "gain": 1.04},
  {"unit": "sr", "floor": 11, "room": 6, "gain": 2.22},
  {"unit": "H", "floor": 8, "room": 24, "gain": 2.17},
  {"unit": "Bq", "floor": 1, "room": 17, "gain": 3.57},
  {"unit": "lm", "floor": 2, "room": 6, "gain": 3.36},
  {"unit": "Gy", "floor": 13, "room": 15, "gain": 1.23},
  {"unit": "J", "floor": 1, "room": 12, "gain": 4.21},
  {"unit": "T", "floor": 6, "room": 9, "gain": 1.04},
  {"unit": "W", "floor": 5, "room": 1, "gain": 4.13},
  {"unit": "Sv", "floor": 3, "room": 5, "gain": 3.73},
  {"unit": "Pa", "floor": 13, "room": 13, "gain": 2.36},
  {"unit": "Hz", "floor": 4, "room": 23, "gain": 4.30},
  {"unit": "lm", "floor": 7, "room": 10, "gain": 2.98},
  {"unit": "S", "floor": 2, "room": 24, "gain": 3.18},
  {"unit": "mol", "floor": 14, "room": 15, "gain": 3.20},
  {"unit": "cd", "floor": 8, "room": 12, "gain": 3.72},
  {"unit": "Hz", "floor": 3, "room": 11, "gain": 4.59},
  {"unit": "Hz", "floor": 11, "room": 9, "gain": 1.09},
  {"unit": "Gy", "floor": 14, "room": 24, "gain": 1.27},
  {"unit": "mol", "floor": 8, "room": 9, "gain": 2.15},
  {"unit": "Gy", "floor": 10, "room": 24, "gain": 2.30},
  {"unit": "C", "floor": 1, "room": 1, "gain": 2.06},
  {"unit": "rad", "floor": 12, "room": 17, "gain": 2.5},
  {"unit": "Gy", "floor": 15, "room": 12, "gain": 4.05},
  {"unit": "lm", "floor": 4, "room": 17, "gain": 2.81},
  {"unit": "Bq", "floor": 3, "room": 13, "gain": 2.80},
  {"unit": "mol", "floor": 4, "room": 14, "gain": 4.23},
  {"unit": "H", "floor": 14, "room": 12, "gain": 4.76},
  {"unit": "N", "floor": 8, "room": 15, "gain": 1.72},
  {"unit": "T", "floor": 5, "room": 2, "gain": 3.04},
  {"unit": "Pa", "floor": 7, "room": 20, "gain": 4.11},
  {"unit": "Gy", "floor": 8, "room": 2, "gain": 3.70},
  {"unit": "kg", "floor": 9, "room": 3, "gain": 2.22},
  {"unit": "Gy", "floor": 5, "room": 7, "gain": 2.29},
  {"unit": "lm", "floor": 11, "room": 1, "gain": 2.51},
  {"unit": "Bq", "floor": 8, "room": 3, "gain": 3.02},
  {"unit": "ohm", "floor": 3, "room": 25, "gain": 4.80},
  {"unit": "mol", "floor": 4, "room": 25, "gain": 4.96},
  {"unit": "s", "floor": 8, "room": 16, "gain": 3.80},
```

```
  {"unit": "T", "floor": 2, "room": 13, "gain": 4.32},
  {"unit": "Wb", "floor": 11, "room": 10, "gain": 3.26},
  {"unit": "cd", "floor": 2, "room": 2, "gain": 3.96},
  {"unit": "Pa", "floor": 12, "room": 4, "gain": 3.90},
  {"unit": "Gy", "floor": 15, "room": 20, "gain": 2.11},
  {"unit": "m", "floor": 3, "room": 21, "gain": 4.25},
  {"unit": "sr", "floor": 12, "room": 21, "gain": 1.58},
  {"unit": "S", "floor": 11, "room": 11, "gain": 3.67},
  {"unit": "S", "floor": 2, "room": 4, "gain": 1.51},
  {"unit": "sr", "floor": 5, "room": 19, "gain": 1.05},
  {"unit": "rad", "floor": 5, "room": 18, "gain": 4.44},
  {"unit": "C", "floor": 3, "room": 7, "gain": 1.16},
  {"unit": "kg", "floor": 2, "room": 5, "gain": 3.31},
  {"unit": "Sv", "floor": 13, "room": 3, "gain": 1.15},
  {"unit": "Bq", "floor": 10, "room": 18, "gain": 3.99},
  {"unit": "T", "floor": 3, "room": 4, "gain": 4.28},
  {"unit": "degC", "floor": 9, "room": 12, "gain": 1.82},
  {"unit": "A", "floor": 11, "room": 4, "gain": 4.88},
  {"unit": "V", "floor": 10, "room": 7, "gain": 3.40},
  {"unit": "Sv", "floor": 14, "room": 4, "gain": 4.05},
  {"unit": "Wb", "floor": 4, "room": 6, "gain": 1.54},
  {"unit": "lm", "floor": 8, "room": 7, "gain": 2.25},
  {"unit": "kg", "floor": 14, "room": 11, "gain": 1.88},
  {"unit": "Hz", "floor": 12, "room": 11, "gain": 3.71},
  {"unit": "V", "floor": 2, "room": 24, "gain": 2.45},
  {"unit": "degC", "floor": 10, "room": 19, "gain": 3.78},
  {"unit": "H", "floor": 10, "room": 10, "gain": 3.88},
  {"unit": "N", "floor": 11, "room": 12, "gain": 2.42},
  {"unit": "kg", "floor": 13, "room": 1, "gain": 2.16},
  {"unit": "Sv", "floor": 11, "room": 23, "gain": 1.87},
  {"unit": "Bq", "floor": 14, "room": 1, "gain": 3.65},
  {"unit": "s", "floor": 6, "room": 19, "gain": 3.90},
  {"unit": "mol", "floor": 11, "room": 19, "gain": 2.77},
  {"unit": "N", "floor": 9, "room": 23, "gain": 3.22},
  {"unit": "lm", "floor": 10, "room": 10, "gain": 3.64},
  {"unit": "Sv", "floor": 5, "room": 18, "gain": 1.5},
  {"unit": "K", "floor": 2, "room": 4, "gain": 4.94},
  {"unit": "m", "floor": 13, "room": 21, "gain": 4.96},
  {"unit": "s", "floor": 2, "room": 13, "gain": 2.94},
  {"unit": "kg", "floor": 7, "room": 1, "gain": 2.41},
  {"unit": "ohm", "floor": 9, "room": 5, "gain": 4.38},
  {"unit": "A", "floor": 8, "room": 25, "gain": 2.82},
  {"unit": "rad", "floor": 12, "room": 25, "gain": 3.77},
  {"unit": "T", "floor": 1, "room": 8, "gain": 2.61},
  {"unit": "N", "floor": 5, "room": 3, "gain": 3.05},
  {"unit": "Pa", "floor": 2, "room": 14, "gain": 3.81}
]
```

”Experiment 1”

A.3 verify_results.js

```
var fs = require('fs');
var cfg = JSON.parse(fs.readFileSync('sen_cfg.json','utf8'));

function query1(s)
{
  if(
    ((s.floor>4 && s.floor<11) &&
    (s.room==7 || s.room==8)) ||
    (s.floor<5 && (s.room==1 || s.room==3) ||
    (s.floor>10 && s.room==9))
```

```
    )
  {
    return true;
  }
  return false;
}

function query2(s)
{
  if(
    s.floor<9 &&
    ((s.gain<1.5 || s.gain>3.5) ||
    (s.room==10 && (s.gain>1.5 && s.gain<3.5)))
  )
  {
    return true;
  }
  return false;
}

console.log("Test Query 1 Results");
cfg.forEach(function(sensor){
  if(query1(sensor))
  {
    console.log(sensor);
  }
});

console.log("Test Query 2 Results");
cfg.forEach(function(sensor)
{
  if(query2(sensor))
  {
    console.log(sensor);
  }
});
```

"Snippet for verifying experiment results"

A.4 benchmark.sh

```
#!/bin/bash

SERVER=34.220.195.165
RUNS=99

declare -a reals

echo -n "$1" > req
for i in $(seq 1 $RUNS); do
  out=$(/usr/bin/time coap-client -m post -f req \
    "coap://$SERVER/.well-known/cbor" 2>&1 > /dev/null)
  echo -ne "$i/$RUNS"\\r
  reals[$i]=$(echo "$out" | awk '{ print $1 }')
done

IFS=$'\n' sorted=($(sort <<<"${reals[*]}"))
unset IFS

echo "median = ${sorted[($RUNS+1)/2]}"
echo "upper quartile = ${sorted[3*($RUNS+1)/4]}"
echo "lower quartile = ${sorted[($RUNS+1)/4]}"
```

```
22  echo "min = ${sorted[0]}"
23  echo "max = ${sorted[$RUNS-1]}"
```

”Script for benchmarking server”

I herewith assure that I wrote the present thesis titled *Enhanced CoAP resource discovery - Application layer interfaces and highly targeted query filters* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, March 11, 2019

(Jawad Ahmad)