DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Smart Building Control with XMPP for IoT

Fabian Sauter

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Smart Building Control with XMPP for IoT

Smart Building Verwaltung mit XMPP für IoT

Author:	Fabian Sauter
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	M.Sc. Teemu Kärkkäinen
Submission Date:	13.12.2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 13.12.2019 Fabian Sauter

Abstract

Traditionally, XMPP was never a solution for interacting or controlling home-user IoT devices. Until now, a full TCP/IP stack introduced too much overhead compared to the available processing power on small IoT devices. Even in the professional space XMPP, in combination with IoT, was never widely used since the existing protocol extension (XEP) is to complex to implement on such low-power devices. However, we try to change this. With our approach, even the smallest of all IoT devices (e.g., light bulbs) can be controlled using XMPP. Our approach focuses on using only already existing extensions like Publish-Subscribe (XEP-0060) or XEP-0004 Data Forms as possible to reduce time spent implementing duplicate features that already exist in another form. Also, we made sure to keep the overall design as simple as possible, forcing developers to keep their applications as simple as possible for home-users to use. Our result demonstrates how simple it can be to use and implement an XMPP extension targeting IoT devices. In a broader sense, this thesis introduces a basic protocol with all the requirements satisfied for a working extension to XMPP that allows home-users to interact with their IoT devices. Later, this basic protocol can be extended to include features like access control or integrating support for IFTTT, Alexa and Google Assistant.

Contents

1 Introduction

In the last few years, we have seen a drastic increase in how many IoT devices are connected to the world wide web. Therefore the need arose, how home-users can interact with those in a standardized way using XMPP. Previous solutions are either to complex or mainly focus on large scale IoT installations. These then only mention home users in a side node.

However, minimal effort has been invested in giving home-users an option to control their IoT devices using their already installed XMPP chat clients.

The purpose of this thesis is to develop a basic XMPP extension protocol (XEP), which allows home-users to interact with their IoT devices purely by using their already installed XMPP chat clients. All of this without having to go through the trouble of having to interact with IoT devices in a purely text-based manner. Additionally, we want to keep the interface for the home-user as simple as possible to use and use as may existing building blocks (XEPs) as possible.

Therefore this thesis starts with a brief overview of all the required technologies. After that, we continue with our system design. There we are outlining the overall system design and all required system components for our protocol. Following the system design, we are introducing our protocol design. Here we have a detailed look at the actual XMPP stanzas send between all participants. To make sure our approach is functioning, we then continue with a detailed look at our reference implantation. In the end, we are concluding with an evaluation of our protocol, including performance measurements (e.g., how much time passes before an action is executed).

2 Background

In this chapter, we will give a brief overview of all the required technologies for this thesis. The first section will talk about IoT in a general manner. After that, we will continue with a brief introduction to XMPP and its terminology. In the end, we will have a look at wireless technologies and compare their key selling points like range, energy usage and data rates.

2.1 Internet of Things

In a general manner Internet of Things (IoT) is an umbrella term for all kinds of devices that are connected to the internet. They do not require any active human interaction to function or report data.

Such devices usually collect and process some kind of sensor data, which then gets either actively sent to other devices (i.e., a central server) or is stored locally until another device/user accesses the data via the internet. An example IoT device could be a thermostat that reports the current temperature and humidity at regular intervals to a central air conditioner controller. The controller then decides based on the data received from our and perhaps a bunch of other connected IoT devices, whether it should turn on and off the local air conditioner to react to temperature or humidity changes.

On the other hand, actuators like light bulbs that are connected to the internet and can be turned on and off or even dimmed are considered an IoT device too.

In this thesis, we mainly focus on IoT devices that are designed to run in buildings owned by home-users and do not focus too much on large scale commercial IoT sensor networks.

2.1.1 Smart Home

As suggested by Ricquebourg, Menga, Durand, et al. [Ric+06], Smart Home is a term for buildings or homes that have so-called *smart* objects or devices. Those smart objects or devices control and monitor properties and or people of that building. This ranges from a simple light bulb, connected to a light sensor turning on when it is getting dark

outside, to a fridge that tracks its contents and informs its owner, in case something is missing for the next meal.

Usually, all devices connect to a central hub where the data then gets processed further. Those hubs often act as a gateway between multiple ways of signal transmission. At the moment, it is common for really small devices like light bulbs or thermometers to send data via low-power, bandwidth and overhead connections (e.g., ZigBee) to their hub. There the data gets relayed. This mostly happens through the use of TCP/IP to other hubs or connected devices like the owner's mobile phone.

At the time of writing, a full TCP/IP (and TLS) stack for data exchange is simply not feasible for the smallest devices. Especially for power constraint devices, a full TCP/IP and TLS stack would introduce too much overhead and Therefore, too much cost in terms of energy.

2.1.2 Data Processing

In general, two places exist where data can be processed. This can happen on the edge (i.e., the device itself) or in the cloud (i.e., on some central server (farm)).

Edge Computing

The process of moving computational efforts to the edge of the network is called edge computing. Here, IoT devices (sensor devices) do not store/send every individual data point (value) directly. Instead, they interpret or process multiple data points to *one* single value that then gets stored for later use or gets send to connected devices and services.

Since we are usually only interested in the overall change of a value, this on the one side drastically reduces the amount of network bandwidth required by the sensor device. Especially in data constraint environments, like if there is only a fixed data cap in combination with a rather small network bandwidth available that multiple devices have to share. On the other hand, we increase the processing power required on those devices which might get relevant, especially in power constraint application cases (e.g., the device is running on battery power).

Cloud Computing

Besides Edge Computing, Cloud Computing is another alternative to how data can be processed. Here, the raw data gets packaged and then send to some kind of remote server (farm). There the data gets stored, processed and evaluated. This is widely used if, for example we want/need to apply complex models on top of our date like it is done for weather forecasts.

For this, small stations spread around the world, collect and report current information about the weather and then send it to a central server. Once the data arrives, supercomputers apply complex algorithms/models to the data to predict tomorrow's weather.

Cloud and Edge Computing are by far not mutually exclusive to each other. Both technologies can cooperate. For example, the IoT devices (sensor devices) already do some kind of light data preprocessing (e.g., smoothing values over time) and then send those data packages to a cloud server (farm), where they get archived and processed even further.

2.2 XMPP

XMPP (Extensible Messaging and Presence Protocol) [XMPa], formerly known as Jabber, is a text-based communication protocol, enabling real-time message exchange between multiple clients. It leverages *XML* (Extensible Markup Language) as its underlying data encoding. The development of Jabber started in 1999 with a focus on instant messaging (IM). With more than one billion users split across multiple services like WhatsApp, Nimbuzz, ChatMe and Kontalk, IM was the largest application field for XMPP [XMPb]. Nevertheless, in the past couple of years this has changed. Application fields like online gaming (about 500 million users [XMPc]), social (e.g., push notifications) (over 2 billion users [XMPd]) and IoT (no concrete statistics) started to get important as well.

The basic protocol structure is defined by RFC 6120 [Sai11a], RFC 6121 [Sai11b] and RFC 7622 [Sai15]. Besides that, additional extensions are defined by so called *XMPP Extension Protocols* (XEPs) [Spe][SCM]. Those XEPs introduce features like *Multi-User Chat* [Saic], *Bookmarks* [BMS] and *Data Forms* [Eat+].

Typically XMPP is used in a distributed client-server architecture environment. Once a connection between two entities (*client-to-server* (c2s) or *server-to-server* (s2s)) has been established, small pieces of structured data ("XML stanzas") are transmitted [Sai11a]. Clients are uniquely identified by so called JIDs (Jabber-IDs). Those JIDs (e.g., *romeo@example.com*) are included in almost every XML stanza. The following example shows a stanza for sending a message with the contents "Neither, fair saint, if either thee dislike.", send from *romeo@example.org* to *juliet@example.com*:

```
<message from='romeo@example.org' to='juliet@example.com'>
<body>Neither, fair saint, if either thee dislike.</body>
</message>
```

2.2.1 Jabber-ID

XMPP uses *Jabber-IDs* (JIDs) to identify individual users and their devices. A *bare JID* is structured like an e-mail address. Such a bare JID consists of two parts, the local and domain part connected by an "@" symbol. As a concrete example, we use the bare JID *romeo@example.org*. Here, "*romeo*" is our local part, identifying a specific user on our XMPP server "*example.com*"(domain part).

Since XMPP supports multiple devices per account (e.g., *romeo@example.org*) there is the need to identify and address each device of an account individually. Thats where so called *full JIDs* are used. A full JID is a bare JID (e.g., *romeo@example.org*) with an appended resource part, separated by a "/". Now our user *romeo* can have multiple devices for example *romeo@example.org/home* or *romeo@example.org/phone* where "*home*" and "*phone*" represent individual devices. Figure 2.1 shows the correlation between bare and full JIDs.

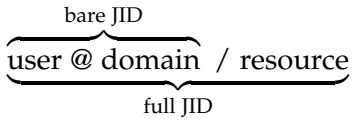

Figure 2.1: Format of a full/bare JID

2.2.2 XMPP Sessions

To build up and hold a connection/session between two entities (c2c or c2s), XMPP leverages TCP as underlying transport layer protocol.

Opening a Stream

Once a TCP connection has been established, the initiating entity sends the first XML message, the so called "*initial stream header*". The following example shows such a header initiating a new XMPP session, where *romeo@example.org* builds up a c2s connection to its XMPP server *example.org*.

```
<?xml version='1.0'?>
<stream:stream
from='romeo@example.org'
to='example.com'
version='1.0'
xml:lang='en'
xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'>
```

Note: At this point the sent message does not represent valid XML. The reason for this is that a complete session between two entities has to be seen as one single valid XML object. Therefore, if an entity likes to discontinue the session, it has to send </stream:stream>. After that, the session forms a valid XML object similar to the one visualized by Figure 2.2.

Once the receiving entity of the initial stream header responds with an initial stream header itself both entities can continue with the next step: *Stream Negotiation*

```
<stream:stream>
<presence>
<show/>
</presence>
<message from="romeo@example.org/terra"
to="juliet@example.org">
<body>
Mercy but murders, pardoning those that kill.
</body>
</message>
...
</stream:stream>
```

Figure 2.2: Valid XML at the end of an XMPP session

Stream Negotiation

Stream Negotiation is the phase where the receiving entity requires certain conditions to be met by the initiating entity like authentication, compression, or a connection upgrade to TLS. Therefore the receiving entity sends so-called "*stream features*" to the

initiator, which then tries to fulfill them. The following shows an example where an upgrade to TLS is required and the use of either *zlib* or *lzw* compression is optional. All features that are required include a <required/> tag. If there is no <required/> keyword included in the feature, this feature is optional and not required to be fulfilled before the entity processed to the next step.

```
<stream:features>
   <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
   <required/>
   </starttls>
   <compression xmlns='http://jabber.org/features/compress'>
   <method>zlib</method>
   <method>lzw</method>
   </compression>
</stream:features>
```

Once all required and optional stream features have been fulfilled both entities have to replace the current session with a new one. This happens by keeping the current TCP connection, which might now be in a new state through a potential upgrade to TLS. After that, both entities resend the initial stream headers and continue with *SASL Negotiation*.

SASL Negotiation

SASL (Simple Authentication and Security Layer protocol) negotiation describes the process of authenticating a SASL client against a SASL server. Therefore the SASL server provides an ordered list of authentication mechanisms to the client. The first item in the order is the most preferred SASL mechanism by the server. Once the client has selected a mechanism to authenticate against the server, this mechanism then gets used for authentication. The following example shows a collection of authentication mechanisms provided by the server.

```
<stream:features>
   <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
       <mechanism>EXTERNAL</mechanism>
       <mechanism>SCRAM-SHA-1-PLUS</mechanism>
       <mechanism>SCRAM-SHA-1</mechanism>
       <mechanism>PLAIN</mechanism>
   </mechanisms>
</stream:features>
```

After successful SASL Negotiation both entities have to replace the current session again with a new one, like described in section *Stream Negotiation*.

Resource Binding

The last step before an XMPP session can be seen as established is the process of *Resource Binding*. Since XMPP has proper multi-device support, in other words: it supports multiple devices associated with a single JID, we have to bind a single device or *resource* to the current stream. This is only required for c2s connections and allows the server to address each of the devices/resources associated with our JID individually. In the following example the client tries to bind the resource *balcony* to the current stream.

```
<iq id='wy2xa82b4' type='set'>
    <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
        <resource>balcony</resource>
    </bind>
</iq>
```

On success, the server responds with the *full JID* (*juliet@im.example.com/balcony*), confirming a successful *Resource Binding* like shown in the following example:

```
<iq id='wy2xa82b4' type='result'>
    <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
        <jid>juliet@im.example.com/balcony</jid>
    </bind>
</iq>
```

Once Resource Binding was successful, the session is established and both entities can start sending XML stanzas.

2.3 Wireless Technologies

There are a couple of wireless communication standards. In the following, we will have a look at a few of them and list their key criteria for the use with IoT devices.

2.3.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is the successor of Bluetooth (Classic). It operates in the same 2.4 GHz ISM (Industrial, Scientific and Medical Band) spectrum like Bluetooth

(Classic). The Bluetooth Special Interest Group (Bluetooth SIG) has developed the BLE standard. There, one device acts as a server and the other ones as clients. The server then offers a set of services to its clients. Those services (e.g., *Generic Access*, *Battery Service*, ...) [Bluc] themselves offer a set of GATT characteristics like *Age*, *Battery Level*, *Battery Level State*, ... [Blub], that carry the actual values.

Range: Bluetooth can have a range from less than a meter up to more than a kilometer [Blud]. The range heavily depends on the use case of the device, as well as the available transmission power.

Energy Usage: The higher the range for a Bluetooth signal should be, the more energy it uses. This results in the energy usage of Bluetooth ranging from 0.01 mW (-20 dBm) up to 100 mW (+20 dBm) [Blud].

Data Rates: While Bluetooth (Classic) can archive data rates of up to 3 Mb/s, BLE only archives data rates from 125 Kb/s up to 2 Mb/s [Blua]. This is caused by BLE being designed especially for very low-power operations like a peacemaker, reporting only rather small bits of data, but over a long time without having access to an external power source.

Security: BLE offers two security modes with a couple of levels each. On the one side, there is mode 1. If a connection in mode 1 is established, starting from level 2, all data that gets send and received is encrypted.

- Level 1: No security (no authentication and no encryption)
- Level 2: Unauthenticated pairing with encryption
- Level 3: Authenticated pairing with encryption
- Level 4: Authenticated LE Secure Connections pairing with encryption

The higher the level, the more security mechanisms are used to make sure attackers do not access to the data transmitted and received.

On the other hand, mode 2 does not offer any kind of encryption; it only offers data signing.

- Level 1: Unauthenticated pairing with data signing
- Level 2: Authenticated pairing with data signing

While level 1 of mode 2 only offers data signing, a connection established in mode 2, level 2, also offers authenticated pairing [Sch].

2.3.2 Wi-Fi

Wi-Fi 6 (IEEE 802.11ax) is the latest version of the Wi-Fi standard developed by the Wi-Fi Alliance [Wi-a]. It can operate in the 1 GHz to 7 GHz range with 2.4 GHz and 5 GHz as the more common frequencies used [Wi-b].

Range: Like with Bluetooth, the possible range is highly dependent on the use case of the device. It can range from a couple of meters up to multiple kilometers under optimal conditions when for example beamforming antennas are being used.

Energy Usage: Since energy usage also heavily depends on the range between devices and which features of Wi-Fi are enabled, we can not give any concrete numbers here. Due to the larger overhead and overall data rates compared to Bluetooth, the energy usage is also higher. Like for the Wi-Fi range, there are no concrete and independent numbers available at the time of writing this thesis.

Data Rates: The latest Wi-Fi standard (Wi-Fi 6) allows theoretically data rates of up to 9.6 Gbps [Wi-b]. At the time of writing, there were no independent sources available that were able to confirm those data rates.

2.3.3 ZigBee

Developed by the ZigBee Alliance, ZigBee is a low-power network protocol, designed for long-range and low data rate networks [Ziga]. This makes it especially useful for home automatization in the area of Smart Home (IoT). Like Wi-Fi and Bluetooth, ZigBee also uses the 2.4 GHz ISM band (IEEE 802.15.4-2011) for its data transmission [Zigb].

Range: ZigBee offers a transmission range of up to 300 meters (line of sight) and 75 to 100 meters indoors [Zigb].

Energy Usage: Due to the low protocol overhead and the also rather low data rates of ZigBee, the power consumed can range from 1 mW up to 100 mW, heavily depending on the range between the sender and receiver [Zigb].

Data Rates: Since ZigBee was designed to be extremely power-efficient, it only can archive data rates of up to 250 Kbits/sec [Zigb].

3 System Design

In this chapter, we will have a look at the system design for our protocol. We incorporate the distributed client-server architecture of XMPP in a way that we do not restrict the home-user to a specific XMPP server. This allows us to either host your own server or use one of the hundreds of public XMPP servers out there [404].

To archive this, our main goal is to not changing anything on the server-side. This allows a faster adoption rate since there is no *chicken or egg* problem. Server developers do not run into the problem of having to wait for client developers to implement the suggested new features first, for them to be able to test their server implementation and vice versa.

Not requiring any server-side modifications also removes, for example the possibility of having some kind of central location where a client could send something like a discovery message too, to get all available/registered IoT devices for this server or home network from.

While ensuring that all IoT devices join a central MUC (Multi-User Chat [Saic]) would solve the problem of discovery, this also would introduce a couple of new problems. For example, who is allowed to join this MUC to retrieve the list of occupants (IoT devices)? Or how do we handle/detect devices that are currently listed as members of the room while they retrieve the list of participants? What happens if an IoT device is currently offline since a MUC room has no persistent list of members. Only those who are currently online are listed as members. All in all, this solution would introduce many more problems than it actually solves.

To tackle this, we suggest making the base protocol usable without any server-side modifications. As soon as enough clients have started working on an implementation and the need for features like a central registry for IoT devices arises, extensions to our protocol could be created where server-side modifications are required.

Figure 3.1 shows a traditional home network connected to the internet using a modem. Since today's modems are not just modems anymore, but instead combine a bunch of functionality (time server, DHCP (Dynamic Host Configuration Protocol), DECT (Digital Enhanced Cordless Telecommunications), media server, ...) in one central device, this results in them being more or less small and a central *home servers* with quite respectable hardware. An example of such a device could be the *Turris Omnia*, with its *ARM Marvell Armada 385* (Dual Core) [Mar] and 2 GB of RAM [CZN],

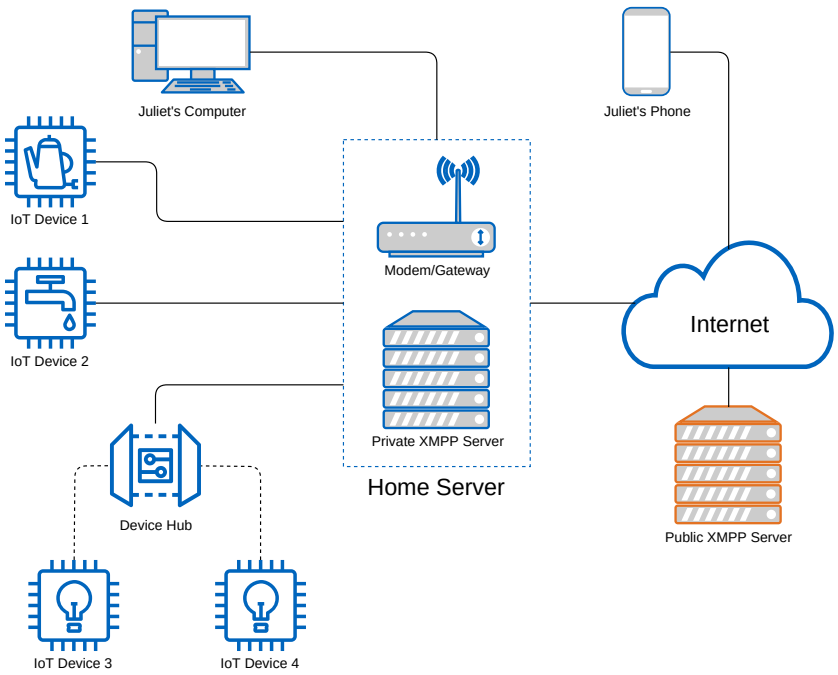

Figure 3.1: System Design

running an extended version of the Open-Source operating system *OpenWrt* [Opea] for embedded devices.

Besides devices built from ground up with the intention to work with an operating system like OpenWrt, for example modems from *TP-Link* [TP-] can also be flashed with OpenWrt and then used as a modem with an integrated XMPP server [Opec]. Nevertheless, flashing modems that are not meant to be used with a modified firmware can be quite tedious. Also, manufacturers will try anything to prevent this, since this could also be used by an attacker to manipulate the original firmware. If it is possible to flash devices with custom firmware, this also could be used to override potential restrictions in how much energy is allowed to use as transmission power for e.g., Wi-Fi antennas [Opeb].

These were only two of the hundreds of different home servers available for home-users, powerful enough to run an XMPP server besides their usual duties as a modem/router.

3.1 Application Area

Since XMPP is a protocol that builds on top of the TCP transport layer protocol and requires a complete TCP/IP and TLS stack with a global protocol state, its application area is restricted. There are multiple factors that restrict the use of XMPP to *larger* devices only.

The first factor is simply the performance overhead required for establishing and maintaining an XMPP session. If we want an XMPP session to be kept up and running while being able to collect and process sensor data in real-time, this requires at least a dual-core microcontroller or even a microprocessor, if we have only a single core, we can not react in real-time to sensor data changes while reacting to incoming requests or simply just data arriving through our XMPP connection. Especially, if the intervals in which our sensor reports data goes down, the availability of our single core for maintaining and reacting to events, happening on our XMPP connection goes down as well.

With the second core, as well as the need to send and receive data over Wi-Fi comes power usage into play. Wi-Fi initially never has been intended to be used by devices with such strict power constraints. Although this has changed with the latest version of the Wi-Fi specification (especially Wi-Fi 6), it still uses a lot more power than other standards designed from the ground up with IoT devices in their minds. Protocols like ZigBee, offering significantly less power usage, while still maintaining usable (but significantly lower) data rates if compared to Wi-Fi exist.

Besides that, with an additional core, the chips die size also increases and not all devices have enough space left for such a large chip. On the low end, devices like light bulbs have a rather strict restriction on how much space electric components can take over in their socket.

Over time, with shrinking die sizes and better, more power-efficient chip designs, those restrictions will be solved. Other than that, we are limited to only larger devices being able to host an XMPP session. This results in us currently having to split up our devices into two groups: *Hub-Based Devices* and *Standalone Devices*.

3.1.1 Hub-Based Devices

Hub-Based devices are devices that connect to a central hub using some kind of low-power and low overhead connection like ZigBee. In Figure 3.1 IoT Device 3 and 4 are connected to such a hub. The hub then acts as a proxy between the low-power ZigBee and XMPP connection.

This is especially useful if you have a lot of rather small devices like light bulbs. If every light bulb in a typical household would be connected to a single access point,

this access point would quickly get overwhelmed by the number of associated devices; it would have to maintain.

Besides that, if the amount of network-connected devices increases, the attack vector does as well. As shown by Wurm, Hoang, Arias, et al. [Wur+16] a lot of devices have numerous flaws that attackers can and are exploiting. Websites like *Shodan* list thousands of IoT devices, that are exposed to the internet readily attackable using back-dors [AA19].

3.1.2 Standalone Devices

While hub-based devices usually are only data collection (i.e., sensor) or actuator devices and expose only a handful of properties, standalone devices usually are the opposite. A device that fulfills the requirements to hold an XMPP session tends to be connected to a power outlet. Examples for such devices could be a fridge, TV or air conditioner. This is shown in Figure 3.1. Here, IoT devices 1 and 2 are directly connected to the local network and the corresponding home server.

3.2 XMPP Server

For devices to be able to communicate with each other, they require an XMPP server that handles the communication. This server also offers services like caching messages if a receiver is not available at the moment or provides a Publish-Subscribe (XEP-0060 [MSM]) interface.

Due to the decentralized nature of XMPP, it is possible for everyone to host their own server (Figure 3.1 Private XMPP Server). Since by far not every home-user is tech-savvy enough or even willing to take on the burden to host a personal XMPP server, it also has to be possible for IoT devices to be registered on a public XMPP server (Figure 3.1 Public XMPP Server).

Latency

One obvious difference between a private and public XMPP server is latency. More concrete: The latency from pressing a button on the user interface (UI) of an XMPP client to, for example turn off a light bulb until the light actually turns off and vice versa. While signals that travel from and to local devices using a local server in between won't have that much latency, for devices that are not located in the same network, the latency between action and reaction can increases drastically up to multiple seconds. It can be quite confusing and frustrating if a button on the phone to turn on a light gets pressed and it takes the light a couple hundred milliseconds (or even one or two

seconds) to actually turn on. Especially if you are standing right next to it and you are used to light switches turning lights on and off immediately.

Registration

Typically two ways exist, how an user can register a new account (JID) for his device: out-of-band and in-band registration. While out-of-band registration typically requires the user to visit some kind of external website and fill out a web form, in-band registration (XEP-0077 [Saib]) allows the registration to happen directly through the XMPP client.

3.2.1 Public XMPP Server

There exists a large number of public and well maintained XMPP servers [404]. But we have to keep in mind that every device registered on a public XMPP server is, in theory, accessible by the public. This means everybody can initiate a conversation with the device and in theory, start accessing the data and services provided by this device. To prevent other people from accessing the data, a permission system, as described in section 4.2.4 is required. Furthermore, even the best permission system does not eliminate the requirement to trust your server provider. Especially data that gets published using Publish-Subscribe (XEP-0060 [MSM]) gets stored in plain text on the server. This leads to the fact that the server provider or any other person/attacker that gets access to the server's database can read and manipulate all the stored data without the user knowing. In a case where devices publish and/or transmit sensible personal data, this is especially dangerous [Mal+18; MIT].

A public server offering services is always a primary target for attackers [Saia]. Besides that, what happens if the server provider goes down or is momentarily unreachable? Do we really want to depend on an external server for turning our lights on and off?

3.2.2 Private XMPP Server

A private XMPP server is a server that is hosted inside our direct sphere of influence (i.e., in the same network as our IoT devices). If we decide to host our own server, this solves a bunch of security-related concerns.

First of all, since the server is under our direct control, we do not have to trust any external person, organization or company to store and handle our data responsibly.

Secondly, thanks to the proper multi-device support of XMPP, it is possible for a user (e.g., Juliet in Figure 3.1) to control associated devices (e.g., IoT Device 1-4 in Figure 3.1)

at home using a computer or tablet and on the go, using a phone associated to the same JID like her device(s) at home.

All of this does not come without a cost. In our case, this cost is security and/or privacy. Since it is no easy task to set up and maintain an XMPP server, this as well can have severe security implications, if the server and its network is not configured correctly. While exposing our local server to the internet allowing users to connect from anywhere in the world to it, control and monitor their devices, this also opens the door for attackers. Those attackers could find a back door in our server setup, infiltrating or even taking over our whole server/home network. All of this just because the home-user was, for example running an outdated version of the server software.

Another way attackers could go is by starting a DoS (Denial of Service) attack against our server, resulting in a crash [MIT]. As a result, devices (IoT and home-user clients) could be unable to communicate with each other (i.e., in the best case, we just can not control our smart light bulbs any more).

As a solution for some of those problems, we suggest integrating the XMPP server into a modem/router. Like mentioned earlier, modern modems usually come equipped with enough compute power and storage capacity to host an XMPP server alongside their other various servers like mail, time, media and files. This would take the burden of having to configure and maintain the server from the home-user and shift this to the modem/router manufacturer or their operating system developers.

This does not come without the possible cost of losing direct control over the server, the freedom of choice which software for the server is being used and what configuration options are actually exposed to the home-user. Nevertheless, this gives home-users simplified and direct access to an XMPP server in their direct sphere of influence. All of this without having to go through the tedious process of setting up and maintaining their own private XMPP server on external hardware.

Anchor Points

it is possible to run an XMPP server without owning a "domain name". An IPv4 or IPv6 address can also be used as a domain name for the server (e. g. romeo@123.123.123.123 (RFC 7622 §3.2 [Sai15])). But this is only feasible if the public IP address of the server does not change on a regular basis.

If it does, we need some kind of Dynamic DNS (DynDns) provider. This provider then acts as an anchor point, providing a subdomain associated with our (regularly changing) IP address. Now we can just use this subdomain (e.g., subdomain.example.org) for our XMPP server and once our local IP address changes we just inform our DynDns provider about the change so he can update the corresponding DNS record(s) for us.

It would also be possible for the router manufacturer to offer such a DynDns service.

He could simply buy a single domain and then based on some kind of unique identifier (e.g., serial number or a hash of the serial number to not directly leak any information about the device) of the router associate a subdomain to each of his routers (e.g., <*serial_number*>*.example.org*).

If we do not want to use our XMPP server from the outside and do not want it being able to communicate with other XMPP servers (e.g., sending chat messages between different XMPP servers), we can use a VPN. A VPN (Virtual Private Network) can be set up and if we want to control our devices from remote, we just have to connect to the VPN. Nowadays, modems/routers (home servers) usually come equipped with everything required to host a VPN server, where we then can connect from the outside to.

A VPN does not solve the need for some kind of anchor point (e.g., DynDns service), which we can contact if the public IP address of our VPN server changes on a regular basis. The same home servers that come equipped with a VPN server usually also are equipped with a DynDns client. This makes it extremely easy for home-users, since everything is available and controlled in a central place (e.g., the home servers web interface).

3.2.3 Registration

Figure 3.2 shows the flow for registering a new IoT device. It starts with the client (Juliet's Phone) scanning the QR-Code printed on the IoT device (IoT Device 1). This QR-Code contains basic information about the device and its Bluetooth MAC address.

Once the scan was successful, the client will contact its XMPP server (step two in Figure 3.2) and registers a new JID for the IoT device.

In the third step of the registration procedure, the client will connect to the IoT device using Bluetooth LE and the scanned Bluetooth MAC address. Once a connection has been established successfully, the client will transmit the registered JID, JID password, Wi-Fi SSID, Wi-Fi password and its own JID to the device.

After that, in step four, the IoT device switches from BLE to Wi-Fi using the received Wi-Fi SSID and password. Once a Wi-Fi connection has been established, the IoT device also connects to the XMPP server with the received JID and JID password. If this succeeds as well, the IoT device sends a simple success message stanza to the client where this stanza then gets mirrored and gets sent back to the IoT device.

If this stanza arrives at the IoT device, the registration process is done. Now the registered device can start collecting data and distribute it or react on data received from other sources.

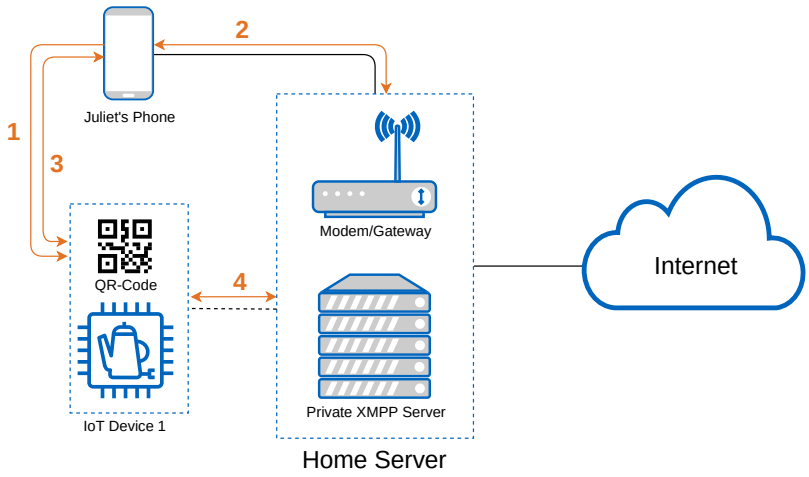

Figure 3.2: Device Registration

3.2.4 Data Access

There are two ways in which we can access the data provided by the IoT devices — active and passive.

Active Data Access

As shown in Figure 3.3 for the active data access, the client (Juliet's Phone) has to become active and send a request to the IoT device (IoT Device 4). This XMPP stanza of type request gets send over the internet to the Private XMPP Server. The server then forwards the stanza to the Device Hub, acting as an XMPP proxy for all its connected devices (IoT devices 3 and 4). Since IoT Device 4 is a hub-based device, the Device Hub will convert the XMPP stanza to some kind of low-power signal (e.g., ZigBee), which it then forwards to IoT Device 4.

Now that IoT Device 4 has received the request from Juliet's Phone, it answers it and sends it back to the Device Hub. There the signal gets converted back to an XMPP stanza of type result. This stanza then gets send to the Private XMPP Server, which forwards it to its receiver (Juliet's Phone) over the internet.

Passive Data Access

Another way of accessing data from an IoT device is by using passive data access. Here, the device (IoT Device 2) offers a data node, to which other users (e.g., Juliet) can subscribe. Once Juliet has subscribed to the node provided by IoT Device 2, she will

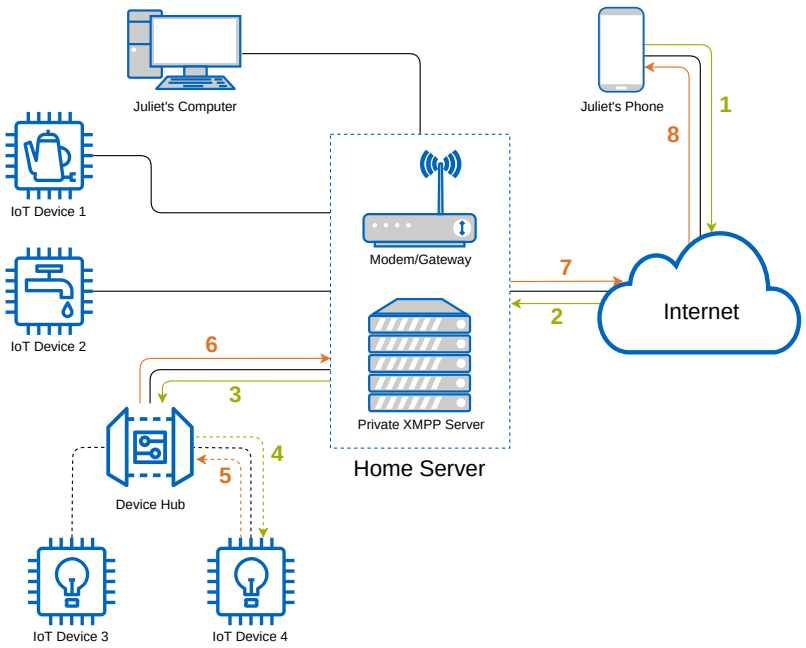

Figure 3.3: Active Data Access

receive all updates provided by IoT Device 2.

This process is illustrated in Figure 3.4. Here, IoT device 2 constantly produces new data, which it publishes to the Private XMPP Server.

There the data gets stored and mirrored to all of Juliet's devices (i.e., Juliet's Computer and Juliet's Phone) that are online. If a device is not online currently, the server forwards it to the device as soon as it comes online for the next time.

If in the meantime even newer data arrives at the server from IoT Device 2, the server will make sure to override the now outdated data This ensures that the client only receives the latest data when it comes online again.

3.3 Summary

In this chapter we introduced the concept of *Hub-Based* and *Standalone* devices. While standalone devices can host their own XMPP clients, pushing data to their XMPP server, hub-based devices can not. Those connect using some form of low-power and low overhead connection like ZigBee to a central hub, which then acts as a proxy between XMPP and for example, ZigBee.

We also do not want to restrict users to private or public XMPP servers. Both types

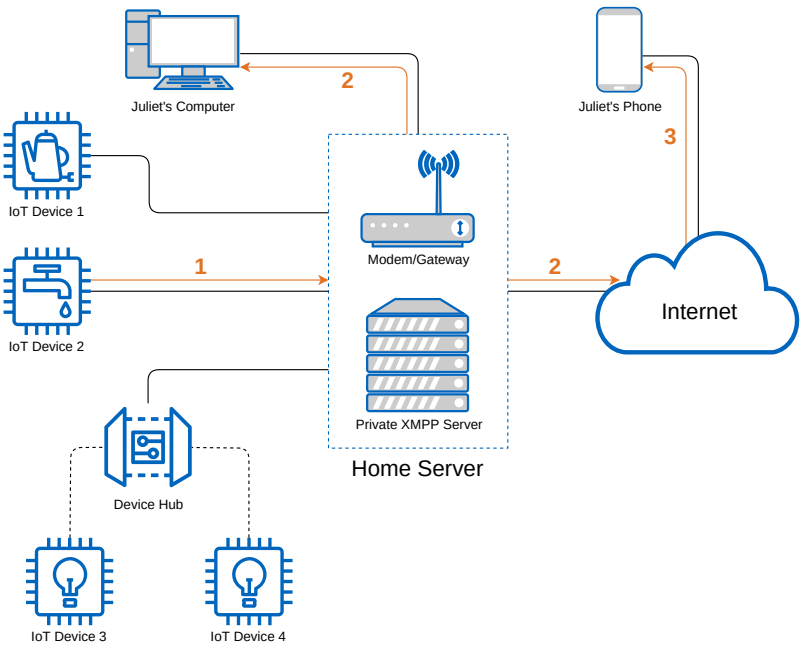

Figure 3.4: Passive Data Access

of servers have their own advantages and disadvantages. While private XMPP servers offer greater security in terms of direct control over the server and its data, public servers are always available and the home-user does not have to go through the hassle of setting up and maintaining his own private XMPP server.

Therefore we propose to integrate XMPP servers into modems/routers since they are nowadays far more than a simple modem/routers. They offer services like SMTP, DECT and act as media servers. This would shift the responsibility of maintaining an XMPP server to the router manufacturer while the home-user still has direct access to the server and its data.

As a general concept for the registration of new IoT devices, we propose the following: The procedure starts with the client scanning the IoT devices QR Code. This QR Code contains information about how to connect to the IoT device using BLE. Once scanned, the client connects to the IoT device's BLE server as a client. Both devices exchange information (e.g., Wi-Fi SSID, Wi-Fi password, JID, ...). As soon as all information has been exchanged, the IoT device shuts down its BLE server and connects to the XMPP server, using the received credentials for Wi-Fi and the XMPP server. Once connected, it confirms the successful registration by sending an XMPP message stanza to the client.

4 Protocol Design

The protocol design can be split up into two sections: *Registration* and *Data Access*. The overall rule for our design and implementations is: First of all to keep it as simple as possible. This means, on the one side it should be as easy as possible for developers to implement the new features. We archive this by using as many existing building blocks (XEPs) as possible.

On the other hand, our design aims to make it for the home-users as simple as possible to register/discover and manage their IoT devices. They should be able to use our design without any additional knowledge about the underlying architecture.

As an additional goal, we aim to allow home-users to interact with their IoT devices even if their client does not yet support all the new features. This won't always be possible (e.g., *Registration*), but basic functionality like requesting data and sending actions to devices should still be possible.

This thesis only focuses on a concrete protocol for standalone devices. Hub-Based devices are not directly targeted by the following protocol. We are going focus on those in an additional paper.

4.1 Existing Solutions

There already exists a solution developed by the *XMPP Interface Working Group* (XMPPI) [IEEb]. IEEE working groups are groups that are open to anyone (no IEEE membership is required). Those groups strive for broad representation of all interested parties and create standards published by the IEEE [IEEa].

The existing solution started as a collection of XEPs [Waha] that later got retracted and are now a part of IEEE [XMPf]. **But** there are a few problems with this existing solution, which makes it not suitable for our use case:

Complexity

The first problem is the complexity of their solution. They provide quite a large and feature-rich framework for all kinds of use cases. They primarily aim to support large scale IoT installations, while still maintaining usability for home-users and their IoT installations. Since in this thesis, we are only interested in the home-user sector, their

solution for e.g., smart contracts [XMPh] and provisioning [XMPg] is irrelevant for us and introduces unnecessary complexity.

Because the solution proposed by the XMPPI is rather large and complex (requires a client, as well as a server component/implementation to function properly), this is leading to a classical "*chicken or egg*" problem, like mentioned earlier. Server developers will (have to) wait for client developers to implement the suggested new features first, for them to be able to test their server implementation and vice versa. Not requiring any server-side modifications also removes, for example the possibility to have some central location where we could send something like a discovery message to, to get all available/registered IoT devices for this server or home network. But this is something that can be added later when the first client implementations exist and the need for such a server component increases.

There exists already a self-contained reference implementation, developed by Waher [Wahc], for all of those features. Nevertheless, this implementation is self-contained and does not build on top of other existing clients or servers, which does not solve the "*chicken or egg*" mentioned above.

Another way to boost availability could be to keep compatibility to clients that do not support the new features introduced. Here, we could, for example allow users of such apps to send messages to the IoT device and get a text-based response.

Besides that, they only mention that additional metadata required should be kept at a minimum and do not give a solution on how to minimize metadata needed [XMPe]. Also, there is no concrete way or procedure described how an IoT device should be connected to the internet if this is only possible using Wi-Fi.

This is going to result in every manufacturer coming up with its way how the device should be connected to Wi-Fi. For example, one could use a proprietary app, with the only purpose of that app being, sending a Wi-Fi SSID and password to the device by the use of Bluetooth. Another one could also use such an app, but that app does not use Bluetooth, instead it could leverage NFC for that. Even more, a third manufacturer then uses some interface, directly built into the device, which has to fulfill this particular case of entering Wi-Fi credentials.

In general manner, freedom in terms of design usually comes with the cost of interoperability and increases fragmentation. Especially if we want to create a process that is as simple as possible for the home-user to follow and is almost the same for all devices, we need a concrete standard for it. This process will not fulfill all potentially existing use cases for all IoT devices out there and there will be exceptions. Still, for the majority of devices, there should be a concrete standard available.

Target Group

The other big problem of the solution provided by the XMPPI is the target group. They aim to support not only large scale IoT installations. Instead, they also aim to support home-user installations as well. But while the basics are there, home-user support is not fully done (yet). As an example, we again use the registration process for new IoT devices. This process is existent but allows manufacturers like mentioned above, much freedom in the way their devices are setup. Like also mentioned above, this is a critical problem because now manufacture can go their way and in the worst case, every device has a different setup procedure.

To make it as easy as possible for the home-user to register and set up a new IoT device, we would like to standardize this process.

4.2 Building Blocks

Since we want to keep our processes and protocols as simple as possible for developers to implement, we heavily focused on using existing XMPP extension protocols (XEPs). In the following, we are going to introduce all XEPs required by our implementation.

4.2.1 XEP-0077: In-Band Registration

This XEP allows clients to register new accounts for example for new IoT devices directly via the XMPP protocol inside a client without having to visit an external website (i.e., the website of the server provider) [Saib].

4.2.2 XEP-0004: Data Forms

Data forms are a way for a device to offer a configuration form that then gets rendered by the client. A user then can fill out this form and send back the result to the sender where it then gets evaluated. The following shows a simple data form which offers the configuration of a simple IoT device.

```
<x xmlns='jabber:x:data'
   type='form'>
    <title>Device Configuration</title>
    <instructions>
    Fill out this form to configure your new IoT device!
    </instructions>
    <field type='fixed'>
        <value>Section 1: Device Info</value>
    </field>
    <field type='text-single'
           label='The␣name␣of␣your␣device'
           var='botName'/>
    <field type='text-multi'
           label='Helpful␣description␣of␣your␣device'
           var='description'/>
    <field type='boolean'
           label='Public␣device?'
           var='public'>
        <required/>
    </field>
</x>
```

The above example only shows the actual data form. Usually a data form comes wrapped inside an <iq/> stanza where it has a *from*, *to* and *type* (with *set*, *result*, ... as it is value) attribute. However, this was neglected here for the sake of readability [Eat+].

4.2.3 XEP-0336: Data Forms - Dynamic Forms

This extension focuses on providing more option for XEP-0004 Data Forms. It introduces properties like <xdd:readOnly/> for read only controls and <xdd:notSame/> for undefined values. This is shown in the following example where we create a data form with a read only control displaying the device id and an undefined value for the bus address control.

```
<x xmlns='jabber:x:data'
   type='form'>
    <title>Device Configuration</title>
    <instructions>
    Fill out this form to configure your new IoT device!
    </instructions>
    <field type='fixed'>
        <value>Section 1: Device Info</value>
    </field>
    <field type='text-single'
           label='Device␣ID:␣'
           var='botId'>
        <xdd:readOnly/>
        <value>583002373287</value>
    </field>
    <field type='text-single'
           label='Enter␣the␣bus␣address␣of␣the␣device.'
           var='busAddress'>
        <xdd:notSame/>
        <value>1</value>
    </field>
</x>
```

4.2.4 XEP-0060: Publish-Subscribe

The Publish-Subscribe extension allows devices to expose so-called *nodes* which other devices than can subscribe to. Those nodes support a full-fledged permission system and once a device has subscribed to a node, it will receive notifications as soon as the value of that node changed [MSM].

4.2.5 XEP-0223: Persistent Storage of Private Data via PubSub

This extension introduces the concept of a private storage associated with an users bare JID. It defines a set of best practices on how to publish nodes for private storage. While Publish-Subscribe (XEP-0060) only focuses on publishing public nodes, this extension shows how, for example we have to configure nodes so only people and devices whitelisted or added to our roster can access and interact with them.

4.3 Registration

Once a home-user has gotten access to a new IoT device, he has to register it at his local or remote XMPP server. Therefore the home-user needs a client, running on a device that has a camera and an antenna supporting Bluetooth Low Energy ($\geq$ 4.0).

Scanning the QR Code

Once all of those prerequisites are met, the registration process can start like shown in Figure 4.2. To start the process, the client (e.g., Juliet's Phone in Figure 4.2) has to scan the QR Code printed on the device itself or its packaging material.

This QR Code contains the Bluetooth MAC address of the device, as well as a public key and the name of the algorithm used to generate this key. For this, we use the Uniform Resource Identifier (URI) syntax defined by RFC 3986 [BFM05]. The following example shows the general syntax of the XMPP IoT registration URI:

```
xmpp:iot-register
?mac=<DEVICE-BLUETOOTH-MAC-ADDRESS>
&algo=<KEY-ALGORITHM>
&key=<KEY>
```

The Bluetooth MAC address is used in step 3 to connect to the BLE server. Besides the Bluetooth MAC address, the URI also contains a cryptographic public key (`key`) and the algorithm used to create this key (`algo`).

Those are used to encrypt and decrypt all messages send from and to the IoT device over Bluetooth. Since we are not able to perform authenticated pairing with encryption (Mode 1 Level 3) for the Bluetooth connection, we would have to rely on unauthenticated pairing with encryption (Mode 1 Level 2). This would allow attackers to host their own Bluetooth server, spoofing the MAC address from our IoT device and Therefore receiving XMPP credentials, as well as our Wi-Fi SSID and password, transmitted in a later step of the registration procedure.

This could be solved by showing the Bluetooth pin on both devices and then the user comparing those manually. Since it is not feasible, especially for small IoT devices like light bulbs to have some kind of small display build in just to show those Bluetooth pins, we decided to use public-key cryptography for this.

We still use encrypted Bluetooth connections, but without authentication (Mode 1, Level 2). Once a Bluetooth connection has been initiated, all data that gets send from and to the IoT device is encrypted. Since an attacker is still able to decrypt data sent from an IoT device if he was able to get access to the device and was able to obtain a copy of the public key, the IoT device should not send any sensitive data to over

Bluetooth. It should just provide necessary information like manufacturer, software revision, hardware revision and it is product name to connected devices.

This also could be solved by using a session key in combination with a hybrid cryptosystem. We decided against this because the IoT device only publishes basic data, which makes it not worth the effort.

Figure 4.1 shows a concrete example, where the IoT device with the Bluetooth MAC address `FF:FF:FF:FF:FF:FF` published a 512 bit Elliptic Curve Cryptography (ECC) public key generated with curve `ed25519`. While the option for specifying allows in theory manufacturers to specify a key algorithm of their choice, this field only exists to make the protocol future prof. This offers the possibility for further revision of this protocol to change this algorithm to something else. At the moment, only ECC keys, generated by curve `ed25519` should be used to prevent segmentation of the market and make it easier for developers to support a wide variety of devices.

We decided to use ECC keys instead of, for example, RSA keys since they provide higher security with a far shorter key. This is especially important for the QR Code. We only have a finite amount of bits available until the QR Code gets to large and Therefore to hard to read, especially for lower-end cameras with a lower resolution. The key length of choice here is at time of writing (autumn 2019) a 512 ECC key, generated using curve `ed25519` [NIS].

```
xmpp:iot-register
?mac=FF:FF:FF:FF:FF:FF
&algo=ed25519
&key=AAAAC3NzaC1lZDI1NTE5AAAAIP
 AHxEzAdvx+W6OENEQVRXUaNHQX
 hhslumDn1N5l6PRl
```

Figure 4.1: Device Registration QR Code

We decided to use QR Codes instead of, for example Near-field communication (NFC) since nearly every device has a camera nowadays. Technologies like NFC are only rather common on mobile devices and are widely used for mobile payment. Nevertheless, access to it is still rather hard since, for example an actual API to access the NFC chip for Apple's mobile operating system IOS just got added quite recently with IOS 11 [HG18].

Besides that, laptops and desktop computers usually do not come equipped with an NCF chip since there is just no need for them to have one. Cameras like mentioned above are far more common and therefore, the way to go here.

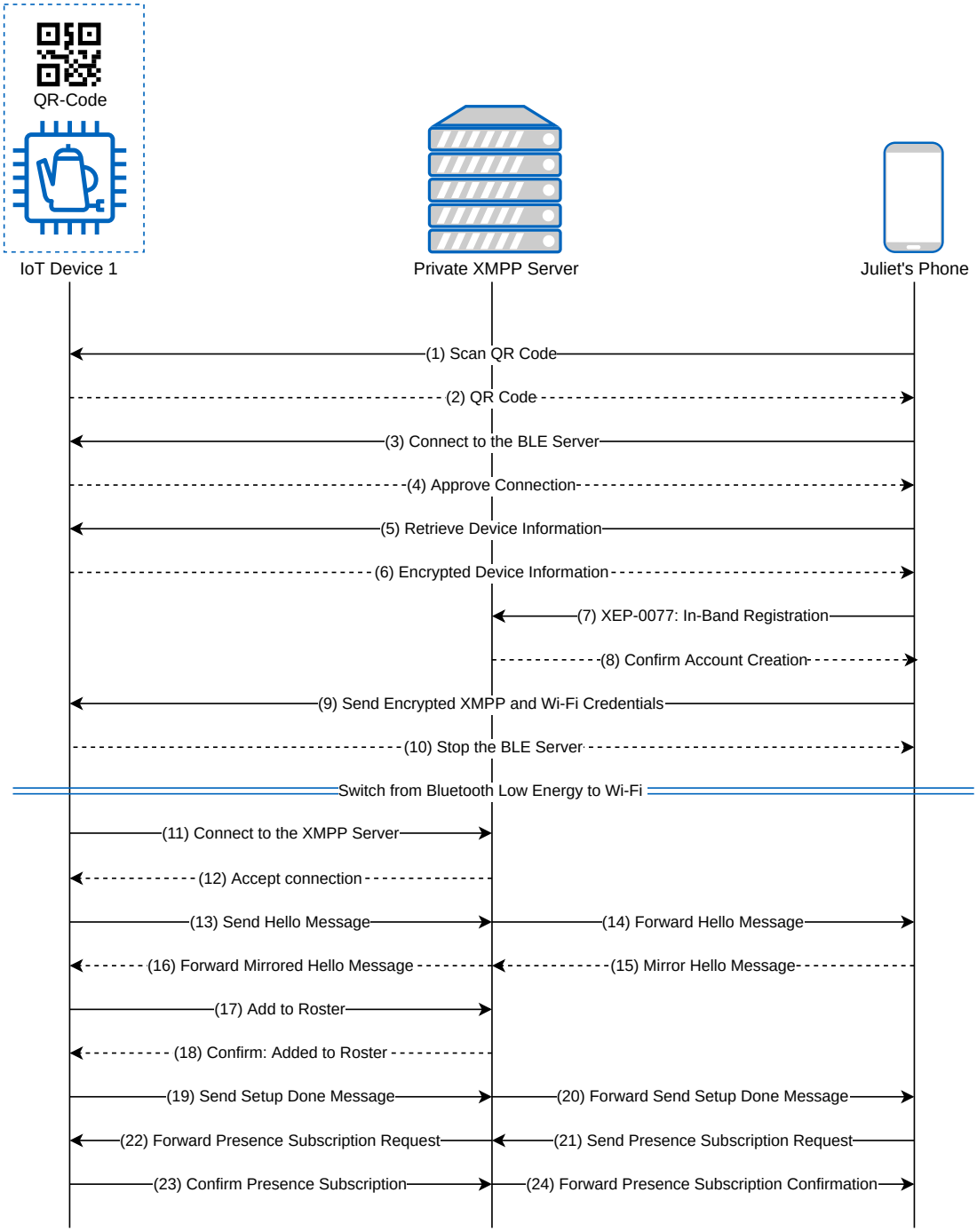

Figure 4.2: Registration process for new IoT devices

UUID	NAME	FLAGS
00002AA2-0000-1000-8000-00805F9B34FB	Language	READ
00002A27-0000-1000-8000-00805F9B34FB	Hardware Revision	READ
00002A28-0000-1000-8000-00805F9B34FB	Software Revision	READ
00002A25-0000-1000-8000-00805F9B34FB	Serial Number	READ
00002A29-0000-1000-8000-00805F9B34FB	Manufacturer Name	READ
00000001-0000-0000-0000-000000000002	Wi-Fi SSID	WRITE
00000002-0000-0000-0000-000000000002	Wi-Fi Password	WRITE
00000003-0000-0000-0000-000000000002	JID	WRITE
00000004-0000-0000-0000-000000000002	JID Password	WRITE
00000005-0000-0000-0000-000000000002	JID Sender	WRITE
00000006-0000-0000-0000-000000000002	Setup Done	WRITE

Table 4.1: Minimum Required Set of Bluetooth Characteristics

Bluetooth Data Exchange

Once the client was able to read the Bluetooth MAC address of the IoT device, he can start initiating a connection to the BLE server provided by the IoT device (steps 3 and 4 in Figure 4.2). As soon as the connection has been established, the client starts to retrieve all information (Bluetooth characteristics) from its Bluetooth server (IoT device), decrypts the contents with the scanned public key and displays them to the user.

Now the user should decide if he wants to proceed. If he decides to proceed the client performs XEP-0077 In-Band Registration to obtain a new JID for the device, with a password provided by the user (step 7 and 9 in Figure 4.2).

After that, the user can decide which Wi-Fi network should be used and eventually provide credentials for it. In steps 9 and 10, the client encrypts the Wi-Fi SSID, Wi-Fi password, JID, JID password and its JID and sends them to the representative characteristics on the server. There the IoT device decrypts them again using its private key and stores them in non-volatile memory for later use.

Table 4.1 gives an overview of all required Bluetooth characteristics and their set flags that should be provided by the IoT devices Bluetooth server. All characteristics ending with `00805F9B34FB` should be stored inside a Bluetooth service with the UUID `0000180A-0000-1000-8000-00805F9B34FB`. All others should be stored in a service with the UUID `00000001-0000-0000-0000-000000000001`.

Hello World over XMPP

Once the configuration is done, the client should send a binary one (`0b0000 0001`) to the *Setup Done* characteristics (encrypted), followed by disconnecting from the Bluetooth server. The IoT device reacts to this by shutting down its Bluetooth server and

connecting to the Wi-Fi network with the received credentials. If it was possible to establish a connection to the Wi-Fi network, the device should continue with establishing a connection to the XMPP server with the given JID and password like described in subsection 2.2.2.

Once this is done, the device has to send a message stanza to the client. The body should not be empty and contain some contend. The following message stanza shows such a message send from our IoT device *kettle@example.org* to *juliet@example.org*.

```
<message from='kettle@example.org/kitchen'
         id='hello_world_1'
         to='juliet@example.org'
         type='chat'
         xml:lang='en'>
    <body>Hi from the ESP32. Please mirror this message!</body>
</message>
```

Once the client received this stanza, he should mirror it and send it back to the IoT device. This is shown by the following example:

```
<message from='juliet@example.org/balcony'
         id='hello_world_2'
         to='kettle@example.org'
         type='chat'
         xml:lang='en'>
    <body>Hi from the ESP32. Please mirror this message!</body>
</message>
```

Now that the mirrored message arrived at the IoT device again, we know the connection works and we are able to exchange messages in both ways. To finish the setup, we add the sender JID that we received over Bluetooth earlier to our roster.

```
<iq type='get'
    from='kettle@example.org/kitchen'
    to='example.org'
    id='hello_world_3'>
    <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.org'/>
    </query>
</iq>
```

After that, we confirm to the client that the setup was successful with another message stanza with a non-empty body.

```
<message from='kettle@example.org/kitchen'
         id='hello_world_4'
         to='juliet@example.org'
         type='chat'
         xml:lang='en'>
    <body>Setup done!</body>
</message>
```

Now the client knows that the setup is done for the IoT device and the data access can begin.

4.4 Data Access

In a general manner *data* can be actual sensor data, as well as the current state of an actuator. For both cases, we need some way to access it and a way to get notified once it changes.

4.4.1 Publishing Data from an IoT Device

For publishing data to other clients we use XEP-0060 Publish-Subscribe (PubSub) in combination with XEP-0223 Persistent Storage of Private Data via PubSub. Therefore we use the node structure shown in Figure 4.3. We have three top level nodes *Sensors* (`xmpp.iot.sensors`), *Actuators* (`xmpp.iot.actuators`) and *UI* (`xmpp.iot.ui`).

The *Sensors* and *Actuators* nodes themselves host other nodes. Those nodes are actual leave nodes and consist only of the current value for the sensor/actuator. Every sensor and actuator has to have its own leave node in the correct node with its value. The *UI* node has only one child, the "`current`" leave node. This leave node contains the definition for the user interface (based on XEP-0004 Data Forms).

Every time an IoT device booted up and has initiated a connection, it first hast to check if all nodes for publishing data still exist. Therefore the device sends a query to it is PubSub server to retrieve all stored nodes.

```
<iq type='get'
    from='kettle@example.org/kitchen'
    to='pubsub.example.org'
    id='nodes1'>
        <query xmlns='http://jabber.org/protocol/disco#items'/>
</iq>
```

If no nodes exist, the server will return an empty query as a result.

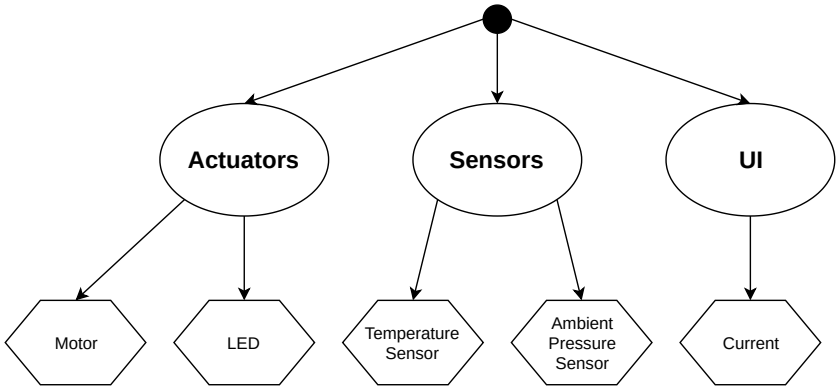

Figure 4.3: XEP-0060 Publish-Subscribe node structure

```
<iq type='result'
    from='pubsub.example.org'
    to='kettle@example.org/kitchen'
    id='nodes1'>
        <query xmlns='http://jabber.org/protocol/disco#items'/>
</iq>
```

If nodes exist, the server returns those top-level nodes. Here, the server has found all required top level nodes `xmpp.iot.sensors`, `xmpp.iot.actuators` and `xmpp.iot.ui`.

```
<iq type='result'
    from='pubsub.example.org'
    to='kettle@example.org/kitchen'
    id='nodes1'>
    <query xmlns='http://jabber.org/protocol/disco#items'>
        <item jid='pubsub.example.org'
              node='xmpp.iot.sensors'
              name='All␣IoT␣sensors'/>
        <item jid='pubsub.example.org'
              node='xmpp.iot.actuators'
              name='All␣IoT␣actuators'/>
        <item jid='pubsub.example.org'
              node='xmpp.iot.ui'
              name='UI␣data␣form␣(XEP-0004)'/>
    </query>
</iq>
```

After that, the device has to query the top level nodes `xmpp.iot.sensors` and `xmpp.iot.actuators` to make sure all sensors still exist.

```
<iq type='get'
    from='kettle@example.org/kitchen'
    to='pubsub.example.org'
    id='nodes2'>
        <query xmlns='http://jabber.org/protocol/disco#items'
               node='xmpp.iot.sensors'/>
</iq>
```

The following example shows two sensors (`xmpp.iot.sensor.temp` for temperature and `xmpp.iot.sensor.bar` for air pressure) existing.

```
<iq type='result'
    from='pubsub.example.org'
    to='kettle@example.org/kitchen'
    id='nodes2'>
        <query xmlns='http://jabber.org/protocol/disco#items'
               node='xmpp.iot.sensors'>
            <item jid='pubsub.example.org'
                  node='xmpp.iot.sensor.temp'/>
            <item jid='pubsub.example.org'
                  node='xmpp.iot.sensor.bar'/>
    </query>
</iq>
```

In case, not all leave nodes exist, the device has to create them. This is done by publishing new items with their value to the corresponding node.

To make sure only trusted people and devices get access to our nodes we set the `pubsub#access_model` option in our `<publish-options/>` node to `roster`. This prevents everyone who is not a part of our roster from accessing our node.

```
<iq type='set'
    from='kettle@example.org/kitchen'
    to='pubsub.example.org'
    id='publish1'>
        <pubsub xmlns='http://jabber.org/protocol/pubsub'>
            <publish node='xmpp.iot.sensors'>
                <item id='xmpp.iot.sensor.temp'>
                    <val xmlns="urn:xmpp:uwpx:iot"
                         unit='celsius'
                         type='float'>22.43</val>
                </item>
            </publish>
        <publish-options>
            <x xmlns="jabber:x:data"
               type="submit"/>
                <field var="FORM_TYPE"
                       type="hidden">
                    <value>http://jabber.org/protocol/pubsub#publish-
                        options</value>
                </field>
                <field var="pubsub#persist_items">
                    <value>true</value>
                </field>
                <field var="pubsub#access_model">
                    <value>roster</value>
                </field>
            </x>
        </publish-options>
    </pubsub>
</iq>
```

In case the device wants to update values stored, for example the temperature changes, it has to publish those values the same way as if it would create new leave nodes. Additionally, both messages have always to include a <publish-options/> node describing who has access to the node and how it is stored on the server.

4.4.2 Subscribing to Data from a Client

If a client likes to get access to data provided by an IoT device, it has to first has to subscribe to all of the three nodes (`xmpp.iot.sensors`, `xmpp.iot.actuators` and

xmpp.iot.ui).

The following example shows the subscription request message, send from a client to its server for the node xmpp.iot.sensors.

```
<iq type='set'
    from='juliet@example.org/balcony'
    to='pubsub.example.org'
    id='sub1'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <subscribe node='xmpp.iot.sensors'
                   jid='juliet@example.org'/>
    </pubsub>
</iq>
```

On success the server will return a confirmation with an unique subscription id.

```
<iq type='result'
    from='pubsub.example.org'
    to='juliet@example.org/balcony'
    id='sub1'>
        <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <subscription node='xmpp.iot.sensors'
                      jid='juliet@example.org'
                      subid='ba49252aaa4f5d320c24d3766f0bdcade78c78d3'
                      subscription='subscribed'/>
    </pubsub>
</iq>
```

If now the device publishes a new value for the xmpp.iot.sensor.temp node all subscribers will be notified about it.

```
<message from='pubsub.example.org'
         to='juliet@example.org/balcony'
         id='notification1'>
    <event xmlns='http://jabber.org/protocol/pubsub#event'>
        <items node='xmpp.iot.sensors'>
            <item id='xmpp.iot.sensor.temp'>
                <val xmlns="urn:xmpp:uwpx:iot"
                     unit='celsius'
                     type='float'>22.43</val>
            </item>
        </items>
    </event>
</message>
```

4.4.3 Text Based Data Access

To make IoT devices also usable from existing XMPP clients, that do not implement all of our proposed features, IoT devices should be able to fall back to text-based data access.

Here, the IoT device has to expose a set of text-based commands that can be requested by sending "`help`" to the IoT device using the default XMPP chat protocol and message stanzas. Once the "`help`" command arrives at the IoT device, it has to respond with a list of available commands. Besides the "`help`" command, the device could offer a command for retrieving the current temperature from a sensor with the "`temp`" command. Another example could be turning an LED on with the "`led on`" command.

4.4.4 The UI node

The `xmpp.iot.ui` node is different than the other nodes. It does not provide any actuator or sensor data. Instead, it contains information about the UI and how a client should display the sensors and actuators to the client.

The following example shows our IoT device *kettle@example.org/kitchen* publishing the XEP-0004 Data Forms UI definition for its sensors and actuators. As an addition we use XEP-0336: Data Forms - Dynamic Forms to define read only sensor readings like shown in the bellow example for the *Temperature* and *Air Pressure* controls. The id of the `item` element always has to be "current" to make sure old definitions get replaced.

```
<iq type='set'
    from='kettle@example.org/kitchen'
    to='pubsub.example.org'
    id='publish1'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <publish node='xmpp.iot.ui'>
            <item id='current'>
                <x xmlns='jabber:x:data' type='form'>
                <title>ESP32 XMPP</title>
                <field type='boolean'
                       label='Light␣on?'
                       var='xmpp.iot.sensor.led'/>
                <field type='boolean'
                       label='Sound␣on?'
                       var='xmpp.iot.sensor.speaker'/>
                <field type='text-single'
                       label='Temperature:␣'
                       var='xmpp.iot.sensor.temp'>
                    <xdd:readOnly/>
                </field>
                <field type='text-single'
                       label='Air␣Pressure:␣'
                       var='xmpp.iot.sensor.bar'>
                    <xdd:readOnly/>
                </field>
            </item>
        </publish>
    <publish-options>
        <x xmlns="jabber:x:data"
        type="submit"/>
            <field var="FORM_TYPE"
                type="hidden">
                <value>http://jabber.org/protocol/pubsub#publish-options</
                    value>
            </field>
            <field var="pubsub#persist_items">
                <value>true</value>
            </field>
            <field var="pubsub#access_model">
```

```
            <value>roster</value>
        </field>
    </publish-options>
</pubsub>
</iq>
```

The client itself is free in the way it presents this form to the user. It just has to make sure all fields are displayed and accessible by the user.

This data form contains all controls (fields) with respective node names. For example the *Air Pressure:* control (field) has the attribute `var='xmpp.iot.sensor.bar'`. The value of this attribute (`xmpp.iot.sensor.bar`) is the same as the pubsub item node. Therefore if the client has a concrete value for the `xmpp.iot.sensor.bar` node, it has to append it to the value of the *label* node.

4.4.5 Publishing Updated Values

The following section will describe how to proceed if, for example a home-user clicks on, for example a toggle switch to toggle on a kettle. Here, our home-user *juliet@example.org* uses her device *balcony* to toggle on her kettle (*kettle@example.org*).

After she pressed the toggle switch representing the power button in her UI, her client has to update the pubsub node. Therefore it sends, like the owner (*kettle@example.org*), a publish stanza containing the updated value for the `powerOn` leave node. This message should not contain any `<publish-options/>` since in this case the sender is not the owner.

```
<iq type='set'
    from='juliet@example.org/balcony'
    to='pubsub.example.org'
    id='publish5'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <publish node='xmpp.iot.actuators'>
            <item id='xmpp.iot.actuator.powerOn'>
                <val xmlns="urn:xmpp:uwpx:iot"
                     type='boolean'>1</val>
            </item>
        </publish>
    </pubsub>
</iq>
```

On success the server returns the node with all its items (leave nodes).

```
<iq type='result'
    from='pubsub.example.org'
    to='juliet@example.org/balcony'
    id='publish15'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <publish node='xmpp.iot.actuators'>
            <item id='xmpp.iot.actuator.powerOn'/>
        </publish>
    </pubsub>
</iq>
```

As shown earlier, the server will now inform all subscribers about the change. The following example only shows this message for the owner of this node (our kettle *kettle@example.org*).

```
<message from='pubsub.example.org'
         to='kettle@example.org/kitchen' id='notification16'>
    <event xmlns='http://jabber.org/protocol/pubsub#event'>
        <items node='xmpp.iot.actuators'>
            <item id='xmpp.iot.actuator.powerOn'>
                <val xmlns="urn:xmpp:uwpx:iot"
                     type='bool'>1</val>
            </item>
        </items>
    </event>
</message>
```

4.5 Summary

All in all, our target was to keep our protocol as simple as possible and reusing already existing building blocks like Publish-Subscribe (XEP-0060) whenever possible. While there already exists a general solution for connecting IoT devices to an XMPP server, this solution is far to complex and has a bunch of weaknesses which forced us to wor on our own design.

As building blocks we are using Publish-Subscribe (XEP-0060), In-Band Registration (XEP-0077), Data Forms (XEP-0004), Data Forms - Dynamic Forms (XEP-0336) and Persistent Storage of Private Data via PubSub (XEP-0223). Those are then used for the *Registartion* and *Data Access*.

For the registration procedure, we use a QR Code printed on the IoT device, which

allows the client to connect to the IoT device using BLE and configure it with, for example the Wi-Fi SSID, Wi-Fi password and its JID. After that, the device can connect to the XMPP server and the client can start subscribing to the PubSub nodes offered by the IoT device.

5 Implementation

The following sections will focus on the reference implementation that was used for validation and practicability testing of the overall design. The implementation for this, has been split up into multiple parts: *The IoT Device The XMPP Client*

5.1 The IoT Device

In this part of the implementation we were focusing on a reference implementation for standalone IoT devices. As a microcontroller we choose the *ESP32* from Espressif [Espa].

5.1.1 The ESP32

The ESP32 from Espressif is a low-power and low-cost system on a chip microcontroller. With its two Xtensa® 32-bit LX6 microprocessors, it is perfect for our requirements, since we would like to host a full TCP/IP stack while still being able to evaluate sensor data in real-time. Some of its other features include integrated support for Wi-Fi, Bluetooth, DAC and ADC. it is produced by TSMC (Taiwan Semiconductor Manufacturing Company) [TSM] on their 40 nm process and able to resist temperatures between $-40\,^{\circ}\mathrm{C}$ and $150\,^{\circ}\mathrm{C}$ [Espb].

5.1.2 Hardware

Figure 5.1 shows our reference device consisting of an ESP32 connected to a button, LED, buzzer and a BMP180 (pressure and temperature sensor). The LED was used as a status indicator like shown in Figure 5.2 and the button as a hardware reset, that erases all configurations once pressed.

As an actuator, we are using the buzzer and the BMP180 is used as a sensor. Both send their current values using XMPP to the subscribed clients like descried in *Protocol Design* section. The buzzer can be controlled by a connected client like also described in the *Protocol Design* section.

Besides that Figure 5.1 also shows the QR Code used for initiating the registration process like described in the *Registration* section.

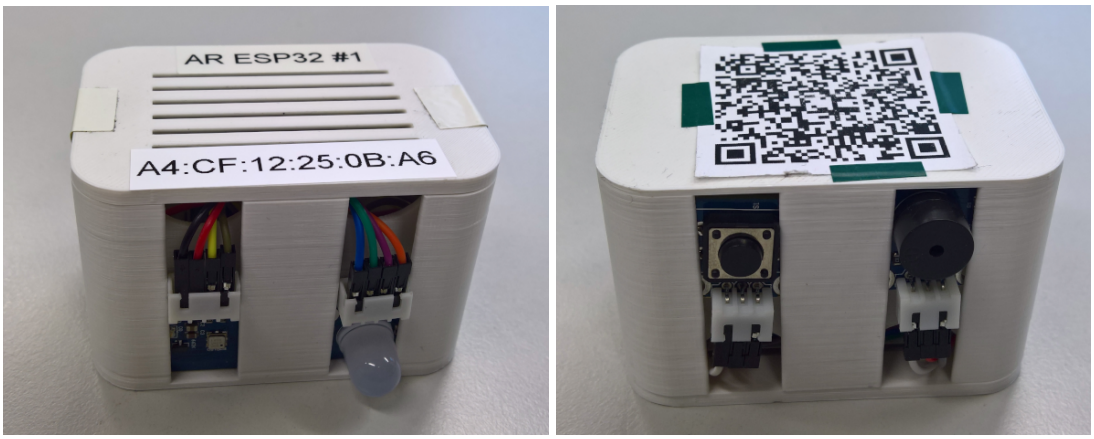

Figure 5.1: Reference IoT device hardware

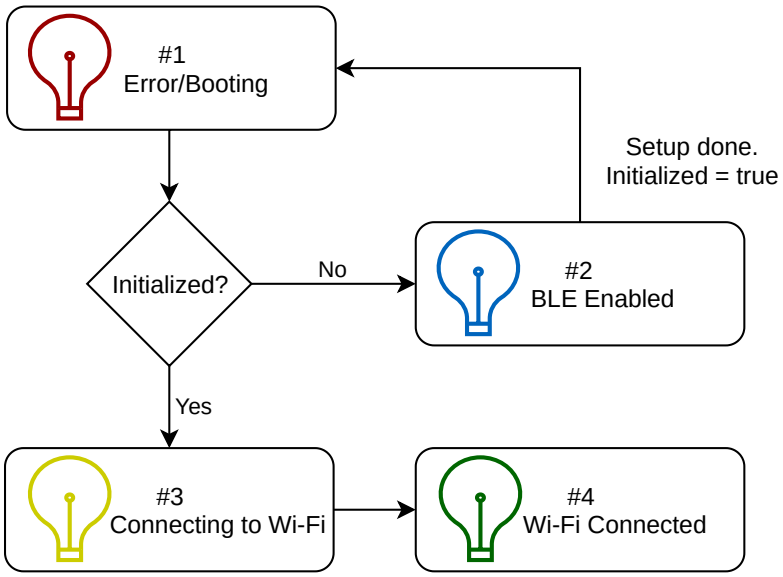

Figure 5.2: ESP32 LED status colors

5.1.3 The Development Framework

As development framework we are using the Espressif IoT Development Framework (ESP-IDF) [Espc]. It offers a rich feature set in C and is rather well documented. Since we wanted to develop the reference implementation in C++, we are using the Smooth framework [Mal]. Smooth offers C++ wrappers for a bunch of ESP-IDF and FreeRTOS APIs. Besides Smooth, we are also using a fork of Neil Kolban's *ESP32 Snippets* library [Kol]. It offers a great C++ wrapper for the Bluetooth and Bluetooth Low Energy APIs provided by the ESP-IDF.

5.1.4 XMPP and the ESP32

For the process of teaching the ESP32 how to speak XMPP, we started searching for a library that was written in C/C++. Furthermore, it had to have a minimal RAM and ROM footprint since we only have 520 KB SRAM and 448 KB ROM available [Espb]. This reduced the amount of usable libraries to only libcouplet [lou], libstrophe [Mof] and dxmpp [ste]. We decided against the use of libcouplet and libstrophe since both of them do not use CMake as their build system and they have too many dependencies, which made them too large to fit onto our ESP32.

It looked like dxmpp was our only option since it uses CMake as their build system and has only pugixml [Kap] and boost [DA] as dependencies. Therefore it was perfect and minimalist enough for our case.

Later it turned out it wasn't. Since dxmpp uses boost threads instead of p_thread, it won't be able to run on the ESP32. We no either had to replace all boost threads with p_threads or write our own minimalists XMPP API. We decided to write our own XMPP API. it is Open Source and available under https://github.com/COM8/esp32-xmpp-iot.

At the time of writing, it included only essential functionally like `PLAIN` SASL authentication and no support of TLS yet. Besides that it uses TinyXML-2 [Tho] as XML library for parsing and generating XMPP stanzas. It has support for basic stanza parsing and support the following XEPs:

- XEP-0060: Publish-Subscribe (basic)
- XEP-0223: Persistent Storage of Private Data via PubSub
- XEP-0004: Data Forms
- XEP-0336: Data Forms - Dynamic Forms

5.2 The XMPP Client

The second part of the reference implementation consists of an XMPP client implementation. While it is still possible to interact with IoT devices purely by the use of text messages, like described in subsection 4.4.3, for registering a new device, a compatible client is required. Therefore we extended the Windows 10 XMPP client *UWPX*.

5.2.1 UWPX

UWPX [Saua] is an Open Source [Saub] XMPP client app for all *UWP* [Mic] (Windows 10) devices. This includes platforms like PC, Windows Phone, Xbox and HoloLens. We choose UWPX since it is Open Source, written in C# and therefore easy enough to extend. It already supports almost all required XEPs so we only needed to add support for the following XEPs:

- XEP-0336: Data Forms - Dynamic Forms
- XEP-0077: In-Band Registration

QR Code Scanner

Besides the work that went into adding support for the XEPs mentioned above, we added a QR Code scanner control. This Control can scan any QR Code and is used by our implementation to scan the QR Codes that start the registration process for new IoT devices.

Bluetooth Low Energy

We also added support for establishing a BLE connection with the IoT device. Therefore we added a complete BLE client implementation that reads, caches and then writes to Bluetooth characteristics provided by the Bluetooth server, running on the IoT device.

5.2.2 Registering IoT Devices

In the following, we are having a look at the registration procedure required for new IoT devices. Like shown in Figure 5.3 the registration procedure starts with the home-user having to select the device type (*Standalone* or *Hub-Based*). At the time of writing only *Standalone* devices are supported.

Once the home-user has selected an option, he has to scan the QR Code printed on the device, like shown in Figure 5.4.

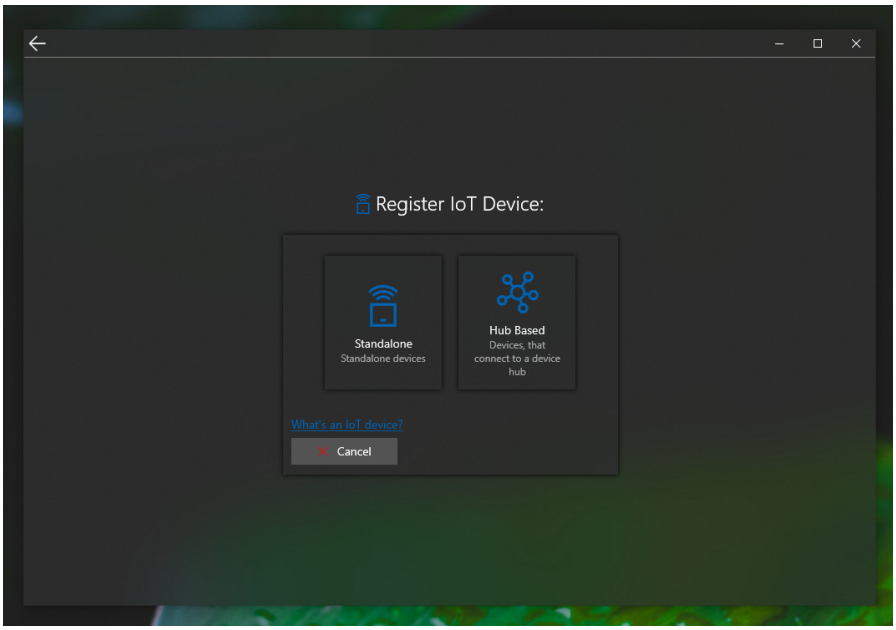

Figure 5.3: Selecting either a *Standalone* or *Hub-Based* device.

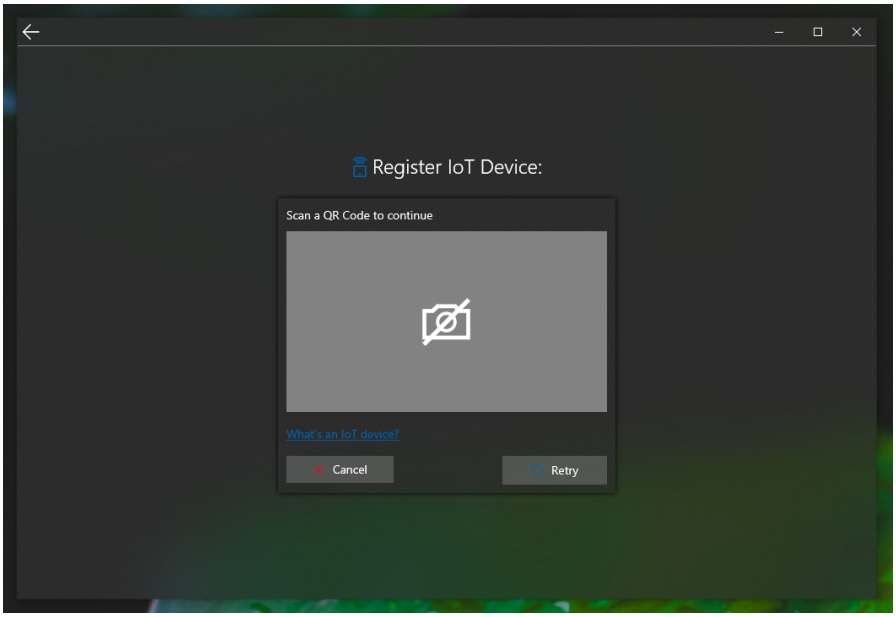

Figure 5.4: Scanning the device QR Code.

After that, UWPX starts connecting to the IoT devices BLE Server as a BLE client, as shown in Figure 5.5.

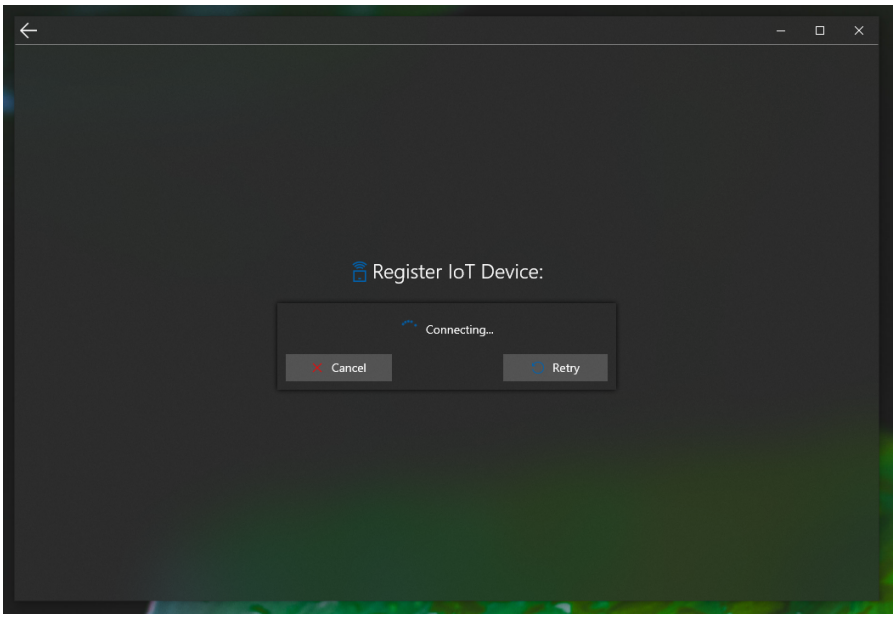

Figure 5.5: UWPX connects to the IoT device using BLE.

Once a connection has been established, UWPX retrieves basic information like device name, manufacturer, serial number and hardware revision from the IoT device and shows them to the home-user. This is shown in Figure 5.6. Now the home-user can decide if this is the correct IoT device he connected to. If so, he can enter the Wi-Fi SSID and password for the IoT device to connect to. UWPX typically already fills out the Wi-Fi SSID filled with the currently connected Wi-Fi network the device is connected to, where UWPX is running on. Due to API limitations, it is not possible to retrieve the Wi-Fi password as well.

If the home-user now clicks on the "`Send`" button, UWPX sends the information using the BLE connection to the IoT device. Once this is done, the IoT device restarts and connects to the configured Wi-Fi network and the XMPP server. This is shown in Figure 5.7.

Once the IoT device has sent the initial hello message using XMPP, UWPX mirrored it and the IoT device responded with another XMPP message stanza the setup is done. This is shown to the home-user in Figure 5.8.

Once the setup is done, the home-user can select the IoT device in the chat view. When the home-user selects the IoT device, UWPX automatically shows the latest

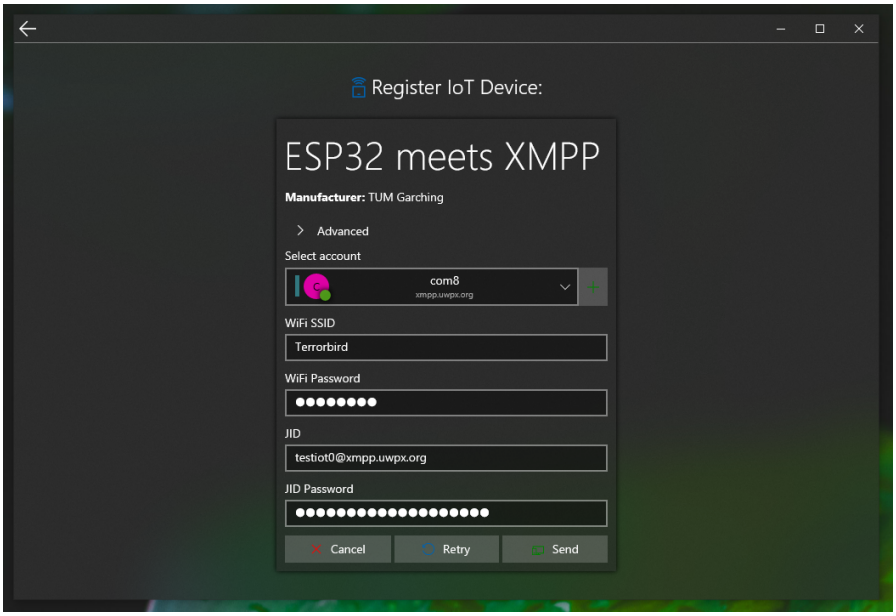

Figure 5.6: UWPX showing device information and configuration options.

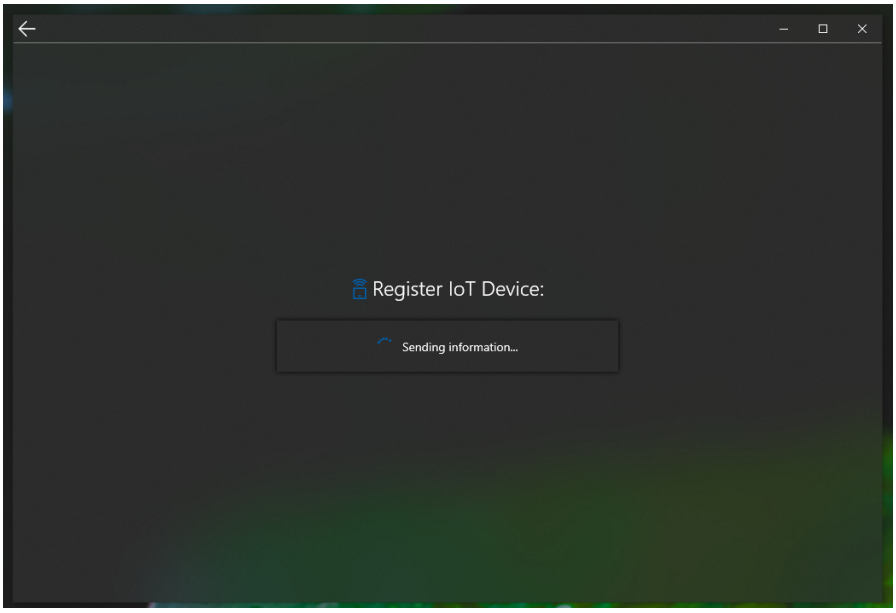

Figure 5.7: UWPX sends all the configuration options to the IoT device.

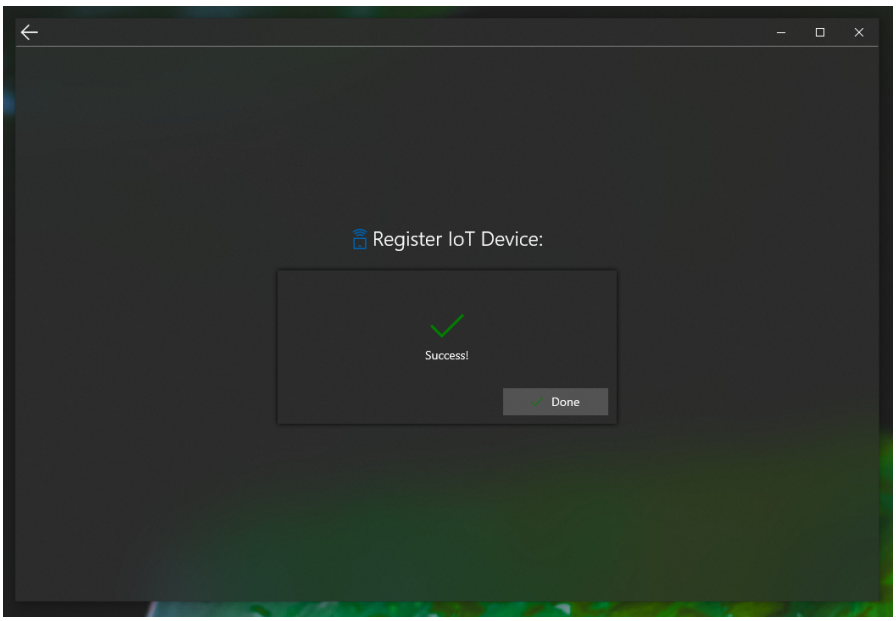

Figure 5.8: Setup was successful.

version of the UI provided by the IoT device. In Figure 5.9 UWPX shows the UI received from an IoT device with two actuators (LED and Speaker) as well as two sensors (temperature and pressure). Once the IoT device publishes new values for its sensors, those will be shown in their representative boxes.

5.3 Summary

To sum up, we split the implementation into two parts: The IoT device and the client. For the IoT device, we are using an ESP32 from Espressif. We are programming it using C++ with the ESP-IDF framework, where we developed a custom bare-bones XMPP client library for it. This is required since all the existing libraries are either too large and therefore, won't fit on our ESP32 or use dependencies that do not work in combination with the ESP-IDF framework we are using.

The second part is a reference XMPP client implementation. Therefore we are extending UWPX, an Open Source XMPP client written in C#. It already provides most of the required components; we are using for our IoT protocol like Publish-Subscribe (XEP-0060).

Because of that, we only had to do minimal work on the XMPP API. We still had to implement the UI, Bluetooth Low Energy and a QR Code scanner for our protocol.

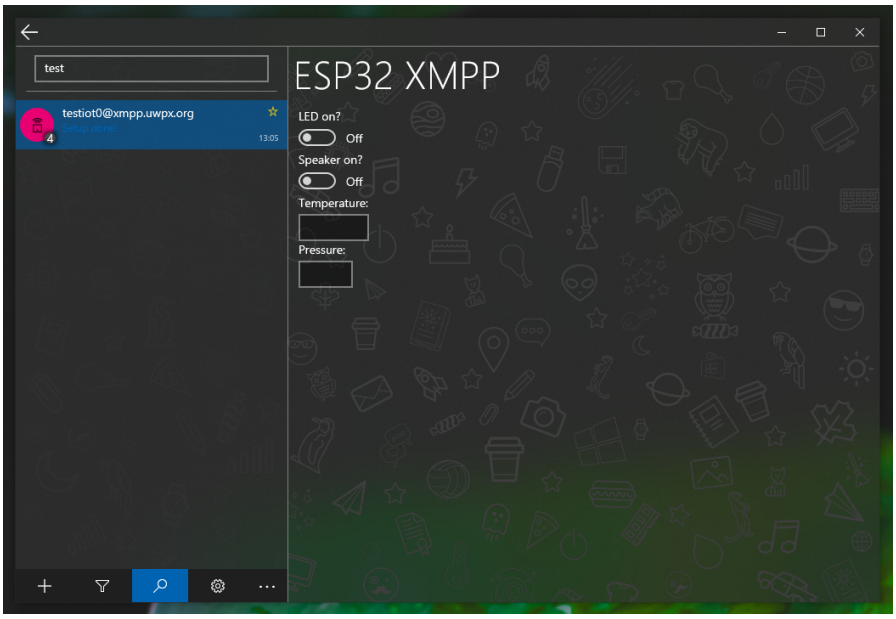

Figure 5.9: Sensor and actuator view with empty values.

6 Evaluation

Our evaluation is split up into two parts: In part on (*Use Case*) we will have a look at the home-user use case and evaluate if we were able to fulfill all of our goals mentioned in the *System Design* and *Protocol Design* chapters.

In the second part (*Performance*) we will go through and benchmark our implementation. We measure for example, how long it takes a measurement that has been sent from an IoT device to arrive at the home-users client.

6.1 Use Case

As a general use case for our solution we are targeting the home automation market. We see our self as a direct competitor to all the existing solutions like *Smart Home* from AVM [AVM] or *Smart Home* from Bosch [Bos].

Therefore we have to ask our self the following questions:

- How easy is the setup process for new IoT devices?
  - What are the requirements?
  - Do we require any server-side components?
  - Will there be a "*chicken or egg*" problem?
- Do home-users require a client that supports all proposed features?
- How easy is it for developers to implement our protocol?
- Did we reinvent the wheel with our solution?

6.1.1 The Setup Process

We want to start with the question: "*How easy is the setup process for new IoT devices?*". To answer this, we conducted a short experiment with 12 test persons. All of them were either bachelor or master computer science students. Eight out of them never used XMPP as a communication protocol before. None of them ever used UWPX as a chat client before.

Their task was to register a new IoT device (like shown in Figure 5.1) to their account using UWPX. After a basic introduction to the chat UI, we started a timer and measured how long it would take them to figure out how to register a new device. We stopped the timer as soon as they archived to get to the success screen (as shown in Figure 5.8).

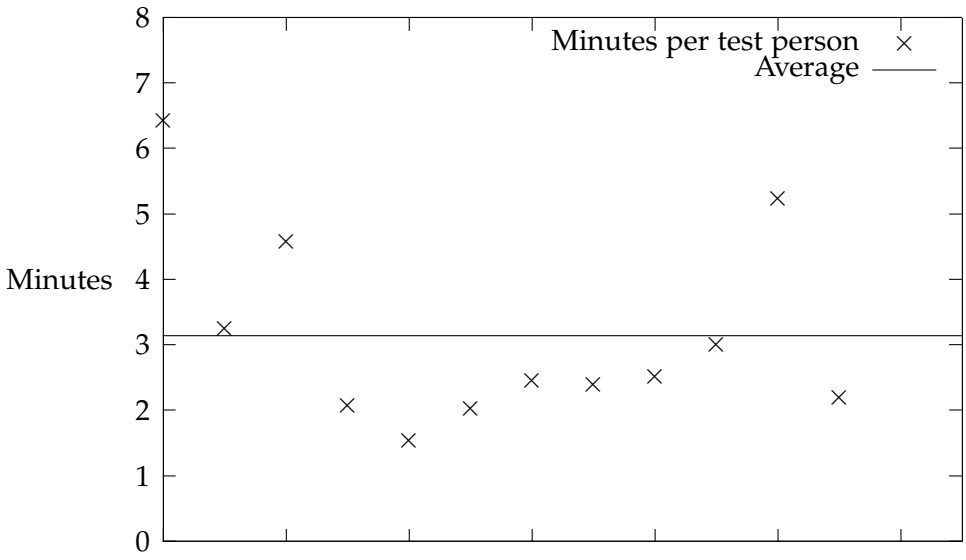

Figure 6.1: Time in minutes for registration per test persons

The results are shown in Figure 6.1, where we see the time in minutes it took them to register a new device. The mean is 2 minutes and 14 seconds and the standard deviation 1 minute and 26 seconds.

So did we succeed in making it as easy as possible for home-users to register a new IoT device?

Yes, since once they found out how to initiate the process, all of them were able to go through it without any additional guidance. Besides that, all of them responded positively to the whole process. The most time was spend either waiting for the BLE data transfer or by typing in the Wi-Fi SSID and password the IoT device should connect to.

Server Side Components: We can realize all of that without requiring any server-side modifications for our protocol. Not requiring any server-side components also limits our ability to discover already registered IoT devices since there is no central registry where new IoT devices register themselves on the server.

Even if we went the route of IoT devices registering themselves in a central group

chat (MUC), this would introduce other problems like: How do we ensure only IoT devices register/enter those group chats? Or for being able to discover all devices in that one central group chat, the home-users client would have to enter the group chat as well, discover all participants and leave it again. During the time he discovers all other participants, he is a participant as well, which again brings us back to the point, how do we distinguish participants in those group chats from each other?

In the long term, we will require a server component that does provide precisely those kinds of features, but for a start until adoption of our protocol arises, a server component is not required necessarily. This also prevents the so-called "*chicken or egg*" problem where server developers would wait for client developers to start implementing our protocol on their side for them being able to test their implementation and vice versa.

Requirements: Let's go back one step and have a look at the requirements for our protocol. First of all, the home-user requires an account on either a private or public XMPP server. Our protocol does not limit the use to private or public XMPP servers only. We want to keep it open for the home-user to decide what server to choose. If he decides to host his own private XMPP server, while this greatly improves the control he as over his data, this does not come without a cost. Here, the cost comes in the form of additional labor he has to do to set up and maintain his private XMPP server.

Therefore we suggested to include XMPP servers in modems/routers since nowadays those have enough computational power to host an XMPP server besides their usual duties as a home server, e.g., media or files. This would shift the burden of setting up and maintaining such a server implementation to the modem/router manufacturer. The manufacturer then would have to make sure the home-user has easy access to it is XMPP server through, for example a dedicated web interface.

Besides the requirement of an account on an XMPP server an XMPP client that supports our protocols is also required — at least for the IoT device registration. Once the IoT device has been registered, the home-user also can interact with its IoT device using plain text message stanzas if he prefers (like described in section *Text Based Data Access*).

6.1.2 Reuse of Existing Protocol Mechanisms

Our second goal was not to reinvent the wheel and make it as easy to implement on top of an existing XMPP client implementation as possible. We think, we also succeeded here since we only use already existing building blocks like Publish-Subscribe (XEP-0060), In-Band Registration (XEP-0077), ...

The only component we are introducing is how sensor values are stored. Therefore we are using a custom `<val/>` XML node with our own namespace (`urn:xmpp:uwpx:iot`) like shown in the section about *Data Access*. In a later iteration of our protocol, we plan to replace this by using a simplified version of XEP-0323: Internet of Things - Sensor Data [Wahb] for this task.

6.2 Performance

To measure the responsiveness and Therefore performance of our solution, we measured the delay between publishing a new sensor reading and the home-users client receiving the updated value.

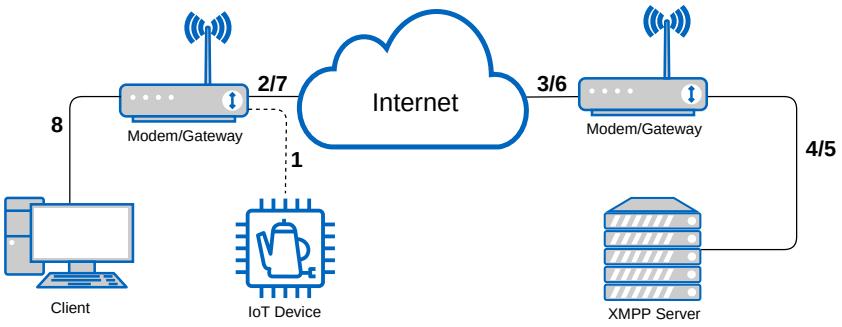

Figure 6.2: Round trip time (RTT) test setup with a public XMPP server

Figure 6.2 shows the test setup for our first test with a public XMPP server. Our IoT device and the client were both connected to the same network. Both them used an external server (`xmpp.uwpx.org` with ejabberd 19.09.1 installed on it) as their XMPP server of choice. While the XMPP client was connected over a wired connection to the modem/router, the IoT device was not. It was connected to a wireless access point in the same network.

Since synchronizing time is rather hard between two devices, especially if we want to measure time in the range of a couple of milliseconds, we decided to measure the round trip time (RTT) instead. Therefore as soon as the client receives an event for a changed value, it has to publish a new value as well. Now, as soon as the IoT device receives this event, it stops the measurement. This process is shown in Figure 6.3.

The results of those measurements can be found in Figure 6.4. With a minimum of 49.69, a mean of 60.88, a variance of 90.48 and a standard deviation of 9.51 milliseconds delay, this is acceptable. Especially for latency-sensitive actions like turning a light bulb on or off, were users expect an instant reaction, this is a good sign.

```
int64_t begin = esp_timer_get_time();
```
IoT device publishes a new value

Client receives the new value

Client publishes a new value as well

IoT device receives the new value

```
double rtt_ms = double(esp_timer_get_time()- begin)/ 1000;
```

Figure 6.3: Measuring the RTT

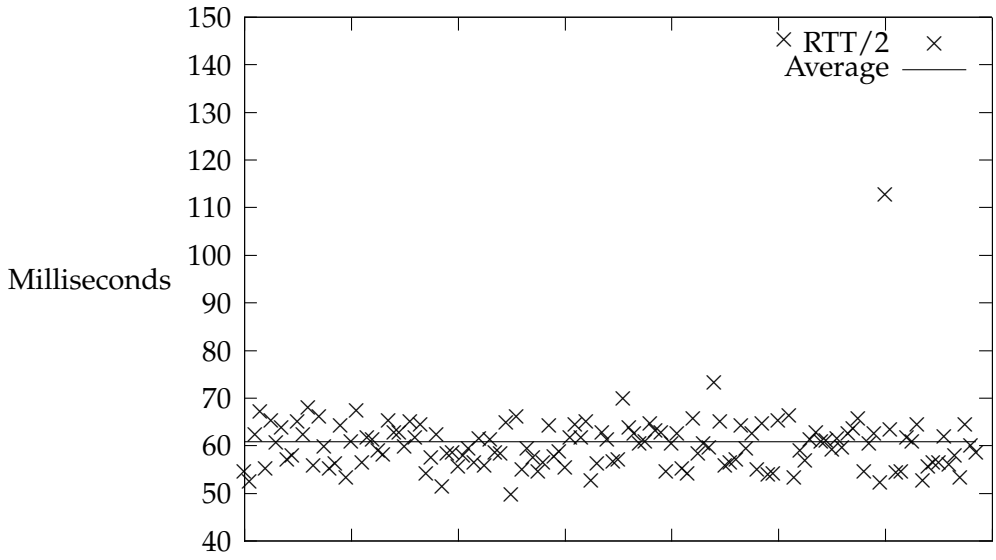

Figure 6.4: *RTT*/2 of a value published to a public XMPP server

We conducted the same test for a network where all three devices were connected to the same network. Figure 6.5 shows the test setup where all devices except the IoT devices had a wired connection to the local modem. The IoT device was connected wirelessly to the modem using an access point. The results of this experiment can be found in Figure 6.6.

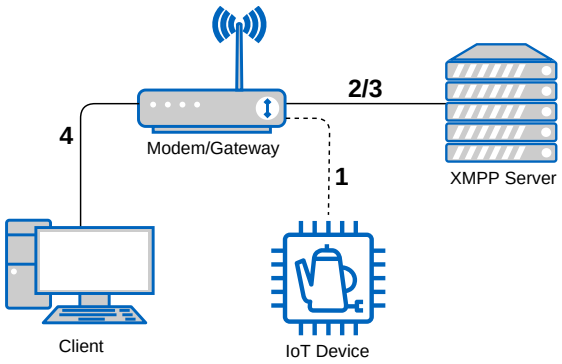

Figure 6.5: Round trip time (RTT) test setup with a private XMPP server

Here, we definitely see a lower mean travel time with a minimum of 0.926 milliseconds and a mean of 13.12, variance of 15.3 and a standard deviation of 3.91 milliseconds delay. Although the mean results are better, we see spikes up to 21.298 milliseconds. Those spikes are probably caused by the unpredictable nature of the wireless connection used for connecting the IoT device to the access point.

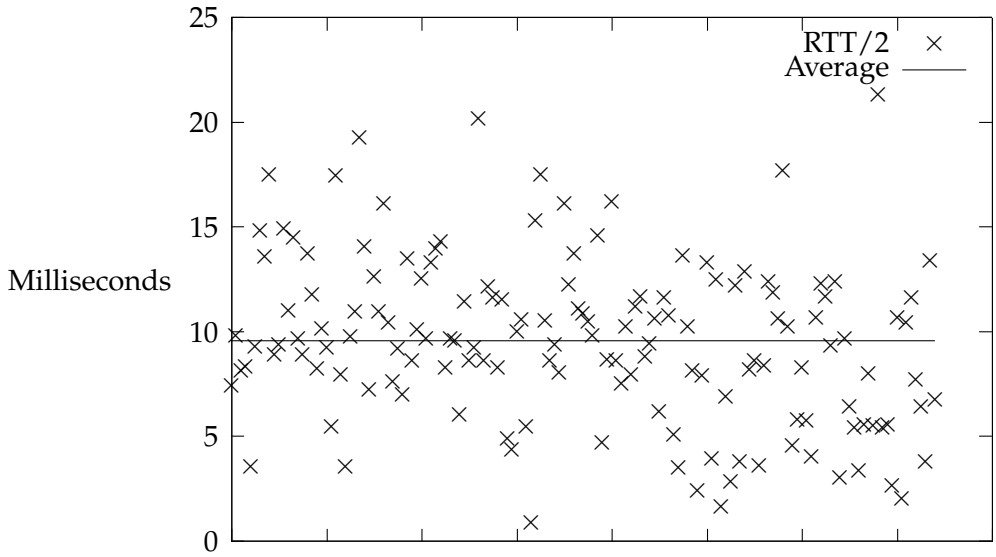

Figure 6.6: $RTT/2$ of a value published to a private XMPP server

7 Conclusion

To sum up we can say, with this thesis we started the first step into the direction of an open standard for connecting and controlling IoT devices using XMPP based clients.

While we have a functioning prototype with a working reference implementation, this is by no means the end of this project. Currently, our protocol only supports features like registering new IoT devices in combination with subscribing and publishing new values for sensors and actuators. It still misses features like access control or transfer of ownership, which would allow for example, other family members to access the same IoT device using their XMPP accounts (JIDs) at the same time. To go even further integration with other services like IFTTT, Alexa and Google Assistant are also possible future ways in how this protocol can be expanded.

Nevertheless, with our approach of reusing existing extensions like Publish-Subscribe (XEP-0060) for publishing and subscribing new values, features like access control need a little bit more work to be usable. All in all, we can say we created a sturdy base for later extensions and changes to our protocol.

List of Figures

List of Tables

Bibliography

[404] 404.city. *Open list of public XMPP servers*. URL: https://xmpp-servers.404.city/ (visited on 10/01/2019).

[AA19] A. Albataineh and I. Alsmadi. *IoT and the Risk of Internet Exposure: Risk Assessment Using Shodan Queries*. IEEE. Aug. 2019.

[AVM] AVM. *Smart Home*. URL: https://avm.de/ratgeber/smart-home/ (visited on 12/07/2019).

[BFM05] T. Berners-Lee, R. T. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, Jan. 2005.

[Blua] Bluetooth Special Interest Group (Bluetooth SIG). *Radio Versions*. URL: https://www.bluetooth.com/bluetooth-technology/radio-versions/ (visited on 10/29/2019).

[Blub] Bluetooth Special Interest Group (Bluetooth SIG). *Specification GATT Characteristics*. URL: https://www.bluetooth.com/specifications/gatt/characteristics/ (visited on 10/28/2019).

[Bluc] Bluetooth Special Interest Group (Bluetooth SIG). *Specification GATT Services*. URL: https://www.bluetooth.com/specifications/gatt/services/ (visited on 10/28/2019).

[Blud] Bluetooth Special Interest Group (Bluetooth SIG). *Understanding Bluetooth Range*. URL: https://www.bluetooth.com/bluetooth-technology/bluetooth-range/ (visited on 10/29/2019).

[BMS] R. Blackman, P. Millard, and P. Saint-Andre. *Bookmarks*. XEP 0048. Version: 1.1 (2007-11-07).

[Bos] Bosch. *Bosch Smart Home*. URL: https://www.bosch-smarthome.com/ (visited on 12/07/2019).

[CZN] CZ.NIC. *More than just a router. The open source center of your home.* URL: https://www.turris.cz/en/omnia/ (visited on 10/15/2019).

[DA] B. Dawes and D. Abrahams. *boost C++ libraries*. URL: https://www.boost.org/ (visited on 12/06/2019).

[Eat+] R. Eatmon, J. Hildebrand, J. Miller, T. Muldowney, and P. Saint-Andre. *Data Forms*. XEP 0004. Version: 2.9 (2007-08-13).

[Espa] Espressif Systems. *ESP32 Overview*. URL: https://www.espressif.com/en/products/hardware/esp32/overview (visited on 09/17/2019).

[Espb] Espressif Systems. *ESP32 Series Datasheet Version 3.1*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (visited on 09/17/2019).

[Espc] Espressif Systems. *Espressif IoT Development Framework (ESP-IDF)*. URL: https://github.com/espressif/esp-idf (visited on 12/06/2019).

[HG18] C. Huynh and A. Galishnikov. *Method and apparatus for virtually writing to a nfc chip*. Patent US20190122010A1. 2018.

[IEEa] IEEE. *What is a Working Group*. URL: https://standards.ieee.org/develop/mobilizing-working-group/wg.html (visited on 12/06/2019).

[IEEb] IEEE. *XMPP Interface Working Group*. URL: https://standards.ieee.org/develop/wg/XMPPI.html (visited on 10/22/2019).

[Kap] A. Kapoulkine. *pugixml*. URL: https://github.com/zeux/pugixml (visited on 12/06/2019).

[Kol] N. Kolban. *ESP32 Snippets*. URL: https://github.com/COM8/esp32-snippets (visited on 12/06/2019).

[lou] louiz'(louiz). *libcouplet*. URL: https://github.com/louiz/libcouplet (visited on 12/06/2019).

[Mal] P. Malmberg. *Smooth*. URL: https://github.com/PerMalmberg/Smooth (visited on 12/06/2019).

[Mal+18] M. I. Malik, I. N. McAteer, P. Hannay, S. N. Firdous, and Z. Baig. "XMPP architecture and security challenges in an IoT ecosystem." In: *Proceedings of the 16th Australian Information Security Management Conference*. 2018, p. 62.

[Mar] Marvell. *Highly scalable, multi-core SoC*. URL: https://www.marvell.com/embedded-processors/armada/armada-38x/ (visited on 10/15/2019).

[Mic] Microsoft Corporation. *What's a Universal Windows Platform (UWP) app?* URL: https://docs.microsoft.com/de-de/windows/uwp/get-started/universal-application-platform-guide (visited on 09/23/2019).

[MIT] MITRE Corporation. *Process-one : Ejabberd : Security Vulnerabilities*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-4455/product_id-7709/Process-one-Ejabberd.html (visited on 10/24/2019).

[Mof] J. Moffitt. *libstrophe*. URL: https://github.com/strophe/libstrophe (visited on 12/06/2019).

[MSM] P. Millard, P. Saint-Andre, and R. Meijer. *Publish-Subscribe*. XEP 0060. Version: 1.16.0 (2019-09-11).

[NIS] NIST National Instiute of Standards and Technology. *Recommendation for Key Management*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (visited on 11/23/2019).

[Opea] OpenWrt. URL: https://openwrt.org/ (visited on 10/15/2019).

[Opeb] OpenWrt. *Exceeding transmit power limits*. URL: https://openwrt.org/docs/guide-user/network/wifi/transmit.power.limits (visited on 11/05/2019).

[Opec] OpenWrt. *TP-Link*. URL: https://www.tp-link.com/en/home-networking/dsl-modem-router/ (visited on 12/11/2019).

[Ric+06] V. Ricquebourg, D. Menga, D. Durand, B. Marhic, L. Delahoche, and C. Logé. *The Smart Home Concept : our immediate future*. IEEE. Dec. 2006.

[Saia] P. Saint-Andre. *Best Practices to Discourage Denial of Service Attacks*. XEP 0205. Version: 1.0.1 (2018-11-21).

[Saib] P. Saint-Andre. *In-Band Registration*. XEP 0077. Version: 2.4 (2012-01-25).

[Saic] P. Saint-Andre. *Multi-User Chat*. XEP 0045. Version: 1.32.0 (2019-05-15).

[Sai11a] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 6120. http://www.rfc-editor.org/rfc/rfc6120.txt. RFC Editor, Mar. 2011.

[Sai11b] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. RFC 6121. http://www.rfc-editor.org/rfc/rfc6121.txt. RFC Editor, Mar. 2011.

[Sai15] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Address Format*. RFC 7622. RFC Editor, Sept. 2015.

[Saua] F. Sauter. URL: https://uwpx.org/ (visited on 09/23/2019).

[Saub] F. Sauter. URL: https://github.com/UWPX/UWPX-Client (visited on 09/23/2019).

[Sch] T. Schlüter. *Sicherheitsmechanismen von Bluetooth Low Energy*. URL: https://return-false.de/archive/1098 (visited on 10/29/2019).

[SCM] P. Saint-Andre, D. Cridland, and R. Meijer. *XMPP Extension Protocols*. XEP 0001. Version: 1.23.0 (2019-01-17).

[Spe] Specifications - XMPP. URL: https://xmpp.org/extensions/ (visited on 09/17/2019).

[ste] stefan (stefandxm). *DXMPP - Deus ex Machinae XMPP framework*. URL: https://github.com/stefandxm/dxmpp (visited on 12/06/2019).

[Tho] L. Thomason. *TinyXML-2*. URL: https://github.com/leethomason/tinyxml2 (visited on 12/06/2019).

[TP-] TP-Link. *DSL Modems & Routers*. URL: https://openwrt.org/toh/hwdata/tp-link/start (visited on 12/11/2019).

[TSM] TSMC (Taiwan Semiconductor Manufacturing Company). URL: https://www.tsmc.com (visited on 09/17/2019).

[Waha] P. Waher. *Internet of Things - Control*. XEP 0325. Version: 0.5 (2017-05-20).

[Wahb] P. Waher. *Internet of Things - Sensor Data*. XEP 0323. Version: 0.6 (2017-05-20).

[Wahc] P. Waher. *IoTGateway*. URL: https://github.com/PeterWaher/IoTGateway (visited on 10/24/2019).

[Wi-a] Wi-Fi Alliance. *Overview*. URL: https://www.wi-fi.org/ (visited on 10/29/2019).

[Wi-b] Wi-Fi Alliance. *Wi-Fi CERTIFIED 6™Highlights*. URL: https://www.wi-fi.org/file/wi-fi-certified-6-highlights (visited on 10/29/2019).

[Wur+16] J. Wurm, K. Hoang, O. Arias, A.-R. Sadeghi, and Y. Jin. *Security analysis on consumer and industrial IoT devices*. IEEE. Mar. 2016.

[XMPa] XMPP (Extensible Messaging and Presence Protocol). URL: https://xmpp.org/ (visited on 09/17/2019).

[XMPb] XMPP (Extensible Messaging and Presence Protocol). *Instant Messaging*. URL: https://xmpp.org/uses/instant-messaging.html (visited on 10/01/2019).

[XMPc] XMPP (Extensible Messaging and Presence Protocol). *Online Gaming*. URL: https://xmpp.org/uses/gaming.html (visited on 10/01/2019).

[XMPd] XMPP (Extensible Messaging and Presence Protocol). *Social*. URL: https://xmpp.org/uses/social.html (visited on 10/01/2019).

[XMPe] XMPP Interface Working Group. *Discovery*. URL: https://gitlab.com/IEEE-SA/XMPPI/IoT/blob/master/Discovery.md#installation (visited on 10/22/2019).

[XMPf] XMPP Interface Working Group. *IoT*. URL: https://gitlab.com/IEEE-SA/XMPPI/IoT (visited on 10/22/2019).

[XMPg] XMPP Interface Working Group. *Provisioning*. URL: https://gitlab.com/IEEE-SA/XMPPI/IoT/blob/master/Provisioning.md (visited on 10/22/2019).

[XMPh] XMPP Interface Working Group. *Smart Contracts*. URL: https://gitlab.com/IEEE-SA/XMPPI/IoT/blob/master/SmartContracts.md (visited on 10/22/2019).

[Ziga] ZigBee Alliance. *Overview*. URL: https://zigbee.org/ (visited on 10/29/2019).

[Zigb] ZigBee Alliance. *ZigBee For Developers*. URL: https://zigbee.org/zigbee-for-developers/zigbee-3-0/ (visited on 10/29/2019).