Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Discovery of IoT devices and it’s integration into robotics

Timothy Ernst

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Prof. Dr. Marco Aiello
Commenced:	June 28, 2021
Completed:	December, 28, 2021

Abstract

In the past decade home-assisting devices have become more and more popular. From smart-home appliances to home assistant-robots software has entered every part of our lives. As a result fulfilling specific needs for a person can become quite complicated. In this work we would like to reduce some of this complexity by introducing new ways of communication between before not commonly linked devices. In particular home-assistant robots and smart-home devices. These robots have become very efficient in completing the task they were designed for but have the potential to do so much more in a home environment. This we would like to achieve by presenting a flexible architecture allowing robots to communicate with other smart-home devices and hereby laying the ground work to allow robots to assist people in new and better ways. Furthermore we will present an implementation of this architecture using the Robot Operating System (ROS) and transport protocol MQTT.

Contents

List of Figures

List of Listings

1 Introduction

In the past decade house-assisting appliances have been becoming increasingly popular. Partly due to these becoming more affordable and partly because of improvements in their AI and addition of feature. Many of these house-assisting appliances can be categorized under the umbrella of smart-home appliances. These can be for instance be lights, heating, blinds to name a few, that can be controlled wirelessly via a separate controller (e.g., a phone app). Not only can these appliances be manually controlled remotely but many such systems already contain other features such as the options to set timers for certain appliances to be switched on and off. These only suffice for predefined events and are therefore inflexible and are unable to react properly to unexpected events [GA16]. Therefore these might not fit every demand an owner of these devices might have. Another kind of house-assisting appliances are autonomous robots such as the Roomba, which is a vacuuming robot. These robots are usually limited to a specific task, which is in the case of the Roomba is vacuuming the floor. This presents these robots with a lot of unexplored potential. Questions arise such as: How could these robots further support the owner? What if these robots could fulfill more than just one simple task? What if these robots could also interact with objects that are not physically attached to the robot? A possible answer could be having these robots, in addition to their core functionality, interact with other smart-home appliances. This could be expressed by having the robot check for light sources within the building at a certain time and turning them off if they are still on. Or in a different scenario have the robot turn lights on, when entering a hallway or a room that is getting too dark and opening doors by itself while maneuvering through the building. The core idea here being to make house-assisting robots more self-sufficient by giving them access to Objects in a house that are not physically part of the robot, while also opening new ways of assisting of human owners. For that a few key problems must be solved: Firstly, the robot must be able to discover new services and be able to connect to them after discovering them. Secondly, it must be able to identify what kind of service it is. For example, if it is a light-service or a window-service. Thirdly, it must be able to interact with given service. Fourthly, it should be able to identify the distance of the object, providing the service, to itself. This can be useful to check if it connected to a targeted device through for instance visual feedback. Fifthly, depending on its set of tasks it might need extra context information to execute its tasks in the most efficient manner, while also making sure to make desired decisions. By desired decisions we mean decisions that the owner of the robot would view as desired as in not turning off the lights while the owner is still in the room, to give an example. This extra context can manifest in behavioural patterns of the owner or the most efficient order of service calls to fulfil a goal, to give another.

Our contributions for this are the following:

- devising an architecture that allows robots to discover devices in a network,
- extracting service information from discovered devices,
- and allows the robot to interact with other devices via messages

We will firstly take a look at what has already been done in this field concerning service discovery and Internet-of-Things (IoT). For this we will look at some already existing technology, understand how it works and then use it build a solution for the 3 Problems we previously mentioned. These will be found in the **Chapter 6**. We will also be looking at different approaches for Service discovery. Sources will be other scientific papers. All this information will be found in **Chapter 3**. We will then design a general architecture to offer a solution for the three problems in **Chapter 4** and specify it further in **Chapter 5**. We will also show an actual code implementation in **Chapter 7** and test our prototype in **Chapter 8**.

2 Background Information

But for us to understand contents of this paper we first need to understand a few key concepts and terms: IoT (short for Internet-of-Things) is the idea of creating a network for physical devices or objects (embedded with sensors, software and other technologies) which these can use to communicate and exchange information. These devices can range from simple household devices, such as a heater, up to heavy industrial machinery[1].

Transport protocols using a publish/subscribe mechanism are often used to communicate with these devices. Reason being that messages will be mainly be sent and received asynchronously therefore it making sense to use this kind of patter to structure and organise messaging. A prime example of the protocols is the standardised protocol MQTT, which we are going to talk about in a later chapter. The general idea of the publish/subscribe pattern is the following: In a network where the publish/subscribe pattern is used there are two distinct groups of entities. On group being message-producing and therefore publishing messages to the network. The other group being message-consuming and therefore subscribing to messages. There is also a third party that can act as an intermediary and is often referred as a “broker”. Sent messages between publishers and subscribers are often structured in hierarchical manner, referred to as topics. Topics can be thought of as a name-space, grouping and describing a set of messages.

As we have already described in our vision in **Chapter 1**, the goal is to make robots further support us in our daily lives. This idea of assisting and supporting of humans through machines is at the core of what we call ubiquitous computing. I. Georgievski and M. Aiello describe ubiquitous computing in their work “Automated Planning for Ubiquitous Computing” as following: “Ubiquitous computing tends towards revolutionising the way we live in terms of comfort, assistance, and safety by cooperatively utilising diverse technologies and various forms of computation to monitor and assist us”[GA16].

Lastly the final concept we will need to know of is the design pattern of Service Oriented Programming. In this a software application is divided into smaller modular units that can be run independently from each other. Each of these modular units is described as a service. Each service offers an interface that allows it to receives messages from other services or software. More details to this concept and its inception can be found in the Paper “Introduction to Service-Oriented Programming (Rev 2.1)” by Guy Bieber and Jeff Carpenter [BC01]. This design pattern will become interesting for us, as one technology we will be using implements this concept to some extent.

[1]https://www.oracle.com/internet-of-things/what-is-iot/

3 State of the Art

M. Aiello and S. Dustdar make use of Web-services in their publication "Are our homes ready for services? A domotic infrastructure based on the Web service stack" [AD08]. In this they propose a Web-service based architecture, combining standard XML based Web-service protocols with the publish/subscribe pattern. Although this making the architecture very scale-able, according to the authors it comes at the price of a "using verbose and computationally intensive XML-based messages"[AD08] and therefore unsuitable for devices with low computational ability. They offer a solution to this problem by suggesting "moderate to very powerful" devices to implement the web service stack while less powerful devices connect to an interface device that implements the web service stack. Their architecture involves one server acting as a Broker for all devices. These devices can discover the server and request a list of all available topics as also publish and subscribe to events via this server. Service discovery is done via publishing of own service information to the network/Broker and then dynamically discovered and invoked by other entities. Device discovery is an important part of each IoT network and therefore to be considered in our architecture. In the Paper "A Categorization of Discovery Technologies for the Internet of Things" by A. Boering et al. [Arn16] evaluate different established technologies used for the discovery of devices. Their focus here being on P2P (Peer-to-Peer) connections. Criteria for evaluation of each technology were Bootstrapping, Range, Basic Search, Rich Queries and Ranking of Results. Boering et al. divide all evaluated technologies into four distinct categories:

1. "search things around you"
2. "search things on my network"
3. "search in directories"
4. "accessing metadata"

The first category deals with devices close to the sender. Therefore, these have a limited discovery range. The discovering service sends a discovery message to a device. The device answers with an advertisement of itself. The second grouping deals with technologies that discover endpoints of devices over a network. Both the discovery service as also a device will be constantly listening to the network. The discovery service will send a multicast message over the network. All devices in the network will then return their own advertisement message to the sender. Alternatively, a central directory can be used for the discovery of devices and their resources. In that case communication happens between directories and the discovery service. The major difference compared to the previous mentioned groups is that the discovery service can send search queries to the directory. These technologies are part of the third grouping. Lastly after discovering a device, the sender would possibly like to access the devices meta data. Technologies dealing with these kinds of interactions are described in the final grouping. The discovering service will request the Metadata via a request message. The corresponding device will reply by sending their Meta data [Arn16].

Next to different technologies the type of network has to be taken into consideration. In their work “Device discovery strategies for the IoT” by P. C. Ccori et al. [CDZS16] analyse the impact different network topologies have on the discovery of devices in a network. To evaluate the differences between topologies 5 criteria are adopted: (1) required storing during discovery, (2) discovery time, (3) network traffic, (4) success rate and (5) reliability. Three different topologies are discussed in this paper, being: centralized, decentralized, and hierarchical networks. Centralized networks usually consist of to types of devices. The first type of devices are devices offering a service in the network. The second type of devices are devices acting as a directory. Every device of the first grouping will connect and send their information (including service advertisements) to this directory. Therefore, all information regarding different service advertisements can be found in this directory. The second network type, decentralized networks, on the other hand only consist of devices offering a service. A global directory does not exist in these networks. Therefore, devices usually only know of their direct neighbours (without discovery). In the last network type, hierarchical networks, each regular node (device) is connected to a super node [CDZS16].

In their work “Intelligent Device Discovery in the Internet of Things – Enabling the Robot Society” [SNY18] authors J. Sunthonlap, P. Nguyen, Z. Ye offer a different way of discovery in distributed IoT networks. In this they use a weighted function to decide which neighboring device receives a message other than flooding the network through sending messages to all neighbors, as is common for regular IoT discovery schemes. Although it is of interest for us to find the most efficient discovery schemes to implement in our architecture, the scheme presented by J. Sunthonlap, P. Nguyen, Z. Ye performs worse than current IoT discovery schemes in matters of success rate and will therefore not be included.

S. K. Datta and C. Bonnet present in their work “Search Engine Based Resource Discovery Framework for Internet of Things” an web-based architecture to discover devices in a network. This they follow up with another Paper “Resource Discovery in Internet of Things: Current Trends and Future Standardization Aspects” [DDB15] providing a case study and future standardisation suggestions. For this Web-services and a RESTful API is used. In this architecture they consider 5 layers of different functionality. The first layer is the “Perception Layer” in which different devices with their different communication protocols are located. These can talk with proxies in the “Proxy Layer”, which implement these different protocols. This is the second Layer listed. The third layer is called the “Discovery Layer”. In this information of different devices is stored and indexed. Also a search engine is present to filter through the collected data efficiently. The fourth layer is called “Service Enablement Layer”. In this the core functionalities of their architecture are exposed and offered to a final Layer called “Application Layer”. Although this architecture includes many features our robot would need to discover devices, this also would make the robot dependant on having a server running this web application. Nonetheless, as we will see in **Chapter 4** our architecture will be heavily inspired by their proposed architecture, due to the flexibility it brings.

Finally the last area we would like to cover is the area of Ubiquitous Computing. As already mentioned in **Chapter 2** one of the primary goals for the robot is to make things easier and comfortable for iFinally the last area we would like to cover is the area of Ubiquitous Computing. As already mentioned in **Chapter 2** one of the primary goals for the robot is to make things easier and comfortable for its user. In “Automated Planning for Ubiquitous Computing” [GA16] I. Georgievski and M. Aiello describe a framework to allow for planning for ubiquitou ts user. In “Automated Planning for Ubiquitous Computing” [GA16] I. Georgievski and M. Aiello describe a framework to allow for planning for ubiquitous computing from qualitative information. As services provided by home-assistant robots by nature fall into the category of ubiquitous computing, this framework allows the creation of better suited services for our goal of better assistance of the device owners.

I. Georgievski et al. even go a step further and compose a solution in their work “Planning meets activity recognition: Service coordination for intelligent buildings” [GNN+17] for more intelligent service design to meet needs within a building management. In this they propose the use of an Artificial Intelligence (short A.I.) to compose services based on different environmental variables and activity recognition. This means a A.I. observes a buildings environment with the help of sensors and tries to correctly guess the current activity with the help of a list of set activities. On this basis the A.I. composes and adjusts services on the activities observed. This could be helpful in guiding a robot on which actions to take. Using this A.I. as means to create taylor-made services matching the exact needs of the occupant(s) is also a future worth considering.

4 Design of Solution

For a robot to interact with other devices, it first needs to know about said devices. For this some sort of discovery mechanism is required. In the following we are going to look at the proposed architecture (as shown in **Figure 4.1**) for said requirement. This architecture will allow a robot to discover and interact with other devices. Note that this architecture model is heavily inspired by proposed architecture in S. K. Datta's and C. Bonnet's work "Search Engine Based Resource Discovery Framework for Internet of Things". We simplify their approach to only three contrary to their five layers. Namely a layer dedicated to handle different communication protocols, the "Proxy Layer". A Layer dedicated to collect all device information received, the "Discovery Layer". And finally the "Application Layer" in which services can be created to interact with other devices and have meaningful message exchanges.

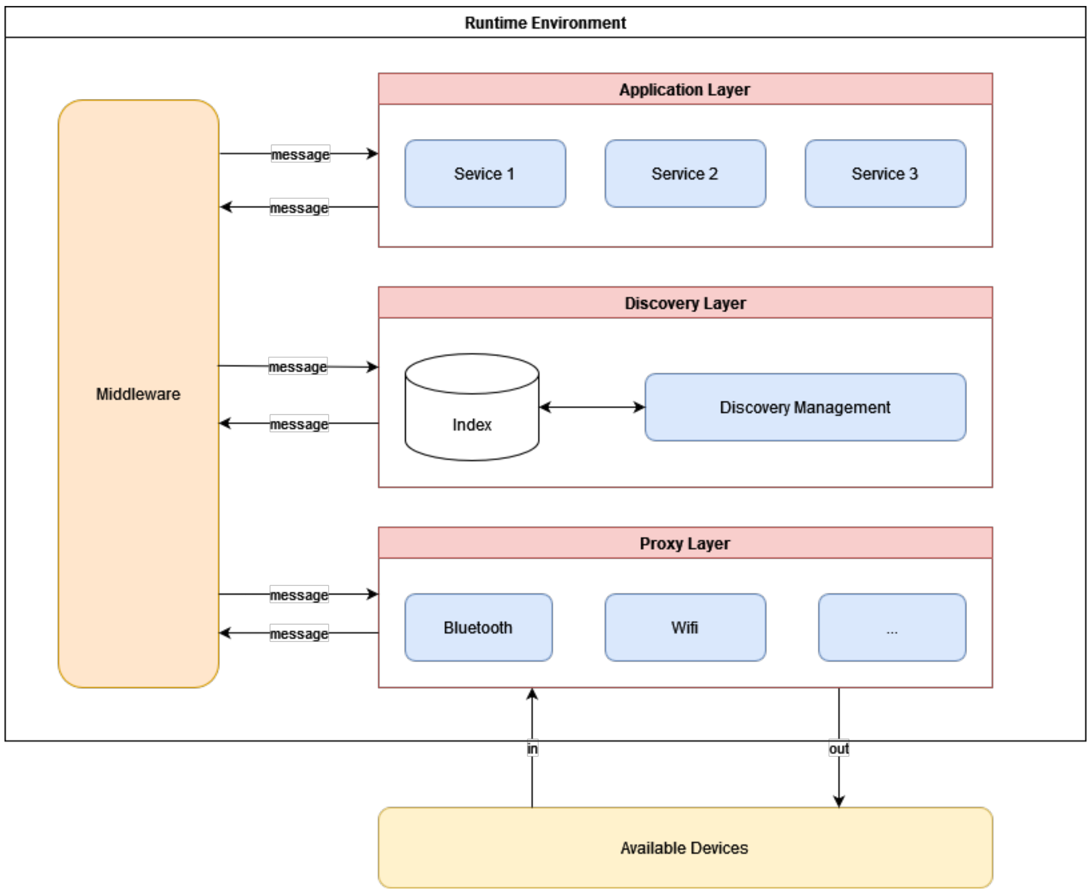

Figure 4.1: Proposed Architecture

For devices to talk with each other, both devices need to be able to talk the same language. In other terms implement the same protocol. There are many different protocols used for communicating between machines with their own set of use-cases. Examples being NFC (Near Field Communication) and BLE (Bluetooth Low Energy) for Close-range communication and mDNS (multicast Domain Name System) for network communication. These mentioned protocols also implemnt discovery features. All device discovery and communication can be found in the “Proxy Layer” of our architecture. Here for every protocol a dedicated gateway service is created. Each gateway acts as interface for all communications for their protocol. Also, all discovery for each protocol is handled by their dedicated service. By structuring communication in that way, deploying new services and removing old services becomes a lot easier. This makes the architecture highly flexible and adaptable to changes in communication technology. The remaining layers of the proposed architecture is designed in a similar way. In this, two more layers are proposed: The “Discovery Layer” and the “Application Layer”. All three Layers are held together and communicate over a middleware. This Middleware will ideally be the operating System of the robot. By doing this integration and deployment of services becomes significantly easier. Also, valuable resources can be preserved, as no additional Environments must be created. The primary function of the “Discovery Layer” is to collect and manage all device information gathered by the interfaces of the “Proxy Layer”. All collected information will be saved to the “Index”. The saving of this information can be either persistent or non-persistent, depending on the set of requirements. The saved information can only be accessed directly over data-managing services described in our architecture as “Discovery Management”. The services in the “Proxy Layer” will inform the “Discovery Management” over any changes that occur to device information. In the final Layer of our architecture, the “Application Layer”, all services can be found that are interested in the collection of device information, stored in the “Discovery Layer”. Services in the “Application Layer” can request this information from the “Discovery Management”. With help of the received device information from the “Discovery Management”, Services from the “Application Layer” can send messages to available devices. This is done with help of the corresponding interface in the “Proxy Layer” over which the device has been discovered. By creating this architecture in style of a service oriented architecture we allow this architecture to stay highly flexible and adaptable to change. Also grouping different functionalities into different services allows for more effective power usage as time dependant services can be shut down without affecting the whole system/application. New Services and can easily be deployed this way as the environment around the robot changes. This will very useful in cases of regular service deployments and adjustments.

5 Realisation of Architecture

To realise this architecture we will need a few things to build it. As out middle-ware for the architecture we will be using the meta operating system ROS. ROS is not only used to develop programs for robots but is according to Vinicius Hax et al. [HDBM13] very suitable as a middle-ware for for IoT purposes. ROS does, like popular IoT messaging protocols, use the publish/subscribe pattern to communicate internally. But we're not only using ROS for its suitability as a middleware for IoT purposes but also because of its inherit support of a service oriented programming by offering the creation of different ROS nodes that can communicate and be discovered over the ROS network. This way we can keep the flexibility described in the previous chapter without creating more overhead. More on how exactly ROS functions will be described in Section 6.1. In the following chapters we will also cover a prototype implementing this architecture.

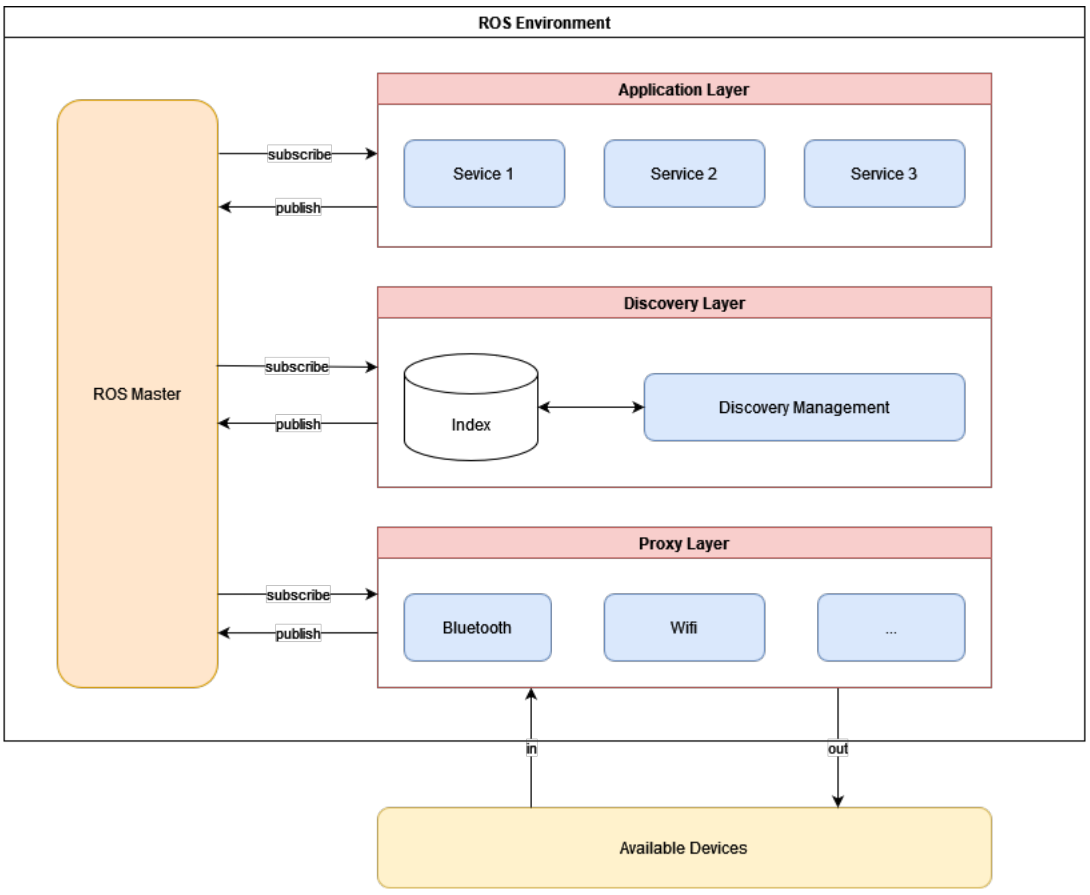

Figure 5.1: Proposed Refined Architecture

As one of the main ways of communicating between ROS nodes/objects is done via the publish/subscribe pattern, we can see the addition of publish and subscribe messages added to our graphic in Figure 5.1. The ROS Master acts here as a Broker (as discussed in Chapter 2) for all ROS Nodes providing them with the contact information of other nodes, topic list a Node can publish or subscribe to.

To develop the services (of our prototype) in each of the layers we will be using the programming language Python. Python is one of two programming languages that are officially supported by the ROS developers for ROS development and therefor maintaining their libraries. There are more libraries created and maintained by the community for numerous other programming languages but we thought it best to go with officially maintained libraries. We will test our prototype by connecting it via a Gateway in the Proxy Layer to a MQTT server. This will provide our prototype with all the necessary inputs.

6 Technological Background

Now that we have we have mentioned all technologies used to implement a solution we will cover the two most important pieces of software we will be using. Namely ROS and MQTT.

6.1 Robot Operating System (ROS)

ROS, short for Robot Operating System, is despite its name not an operating system.
Rather ROS is an open-source, meta-operating system [1] and better viewed as a Software Development Kit (short SDK) for robots. [2]
ROS contains many services a operating system would have, such as hardware abstraction, low-level device control message-passing between processes, and package management.
Note that even though ROS is used for robots, ROS is not a real time framework. Even so it is possible to integrate ROS with real time code.
ROS applications are organised into packages and metapackages. Metapackages on the other hand are used to structure and group packages together.
Packages are the most atomic build and release items of ROS applications. These can contain ROS runtime processes, (ROS-dependant) libraries, configuration files and datasets.
These packages are not limited to the previous mentioned contents but can also include other useful files (e.g., message files, scripts).
ROS includes client libraries which allow ROS nodes written in different programming languages to communicate. [3]

6.1.1 ROS Computational Graph

The ROS Computational Graph is the peer-to-peer network of ROS processes. A peer-to-peer networks is a distributed application.
In these applications all participants (here ROS nodes) are equally privileged.
According to the official ROS wiki, “The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.” [4]
In this paper we are only going to cover ROS nodes, the ROS Master, topics, services, and messages, as these are core to understand the information flow of the ROS Computational Graph.

[1] http://wiki.ros.org/ROS/Introduction
[2] https://www.ros.org/blog/why-ros/
[3] http://docs.ros.org/en/rolling/Concepts.html
[4] http://wiki.ros.org/ROS/Concepts

ROS nodes are the process entities in a ROS network.
ROS nodes can be in the same process, a different process, and on different machine. These perform their own computations and can communicate with other nodes on the network. Communication between ROS nodes is done through messages.
Messages are simple data structures, comprised of typed fields.

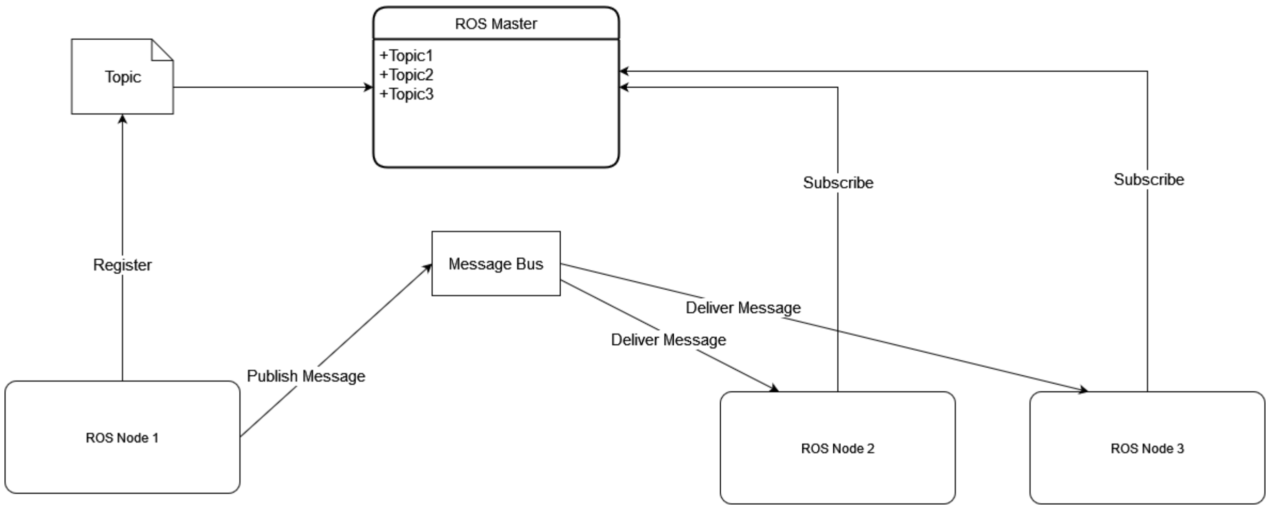

Figure 6.1: ROS Topic example

Messages are transported via special message buses with publish/subscribe semantics. This means a ROS node needs to publish their messages to these message busses and subscribe to receive the message. An exception are services and we will talk about them shortly. The identifiers of these message busses are called topics. topic names follow the pattern of strings separated by /". An example topic in ROS could be: "ros/home/device/".
Topics are also used to identify the contents of a message.[5]

All ROS nodes interested in receiving certain types of data will subscribe to the corresponding topics. The delivering of messages is done anonymously, meaning the subscriber of a topic does not know who published the message. The publisher of a message also doesn't know who is receiving their messages.
Concerning the relationship between publisher and subscriber to a topic, it is a many-to-many relationship.
This means there can be multiple publishers and multiple subscribers to the same topic. An example for described interactions is shown in **Figure 6.1**

[5]http://wiki.ros.org/Topics

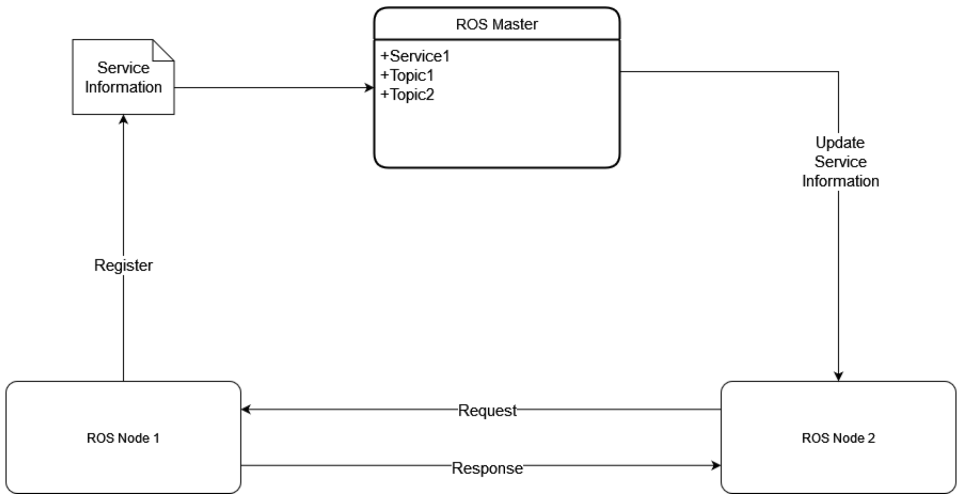

Figure 6.2: ROS Service example

Next to publishing and subscribing to topics, nodes can also offer and use services shown in **Figure 6.2**.
For this, nodes using a service connect directly to a node offering the service. The requesting node will send a request to the node offering the service and will wait for a reply.
In programming terms, the requesting node makes a remote procedure call (short RPC). Like a topic, services are published under a name.
The ROS Master acts as a look-up directory for all entities in the Computational Graph.
Here all topic and service registration information are stored. This includes information required to connect to registered nodes.
The ROS Master informs nodes of any changes that occur to this collection of information.
This allows node to dynamically create new connections as new nodes are registered.

6.2 MQTT

MQTT is a light weight client server publish/subscribe messaging transport protocol.
The following we directly quote out of the latest version (MQTT 5) of the MQTT Standard[6] :
“The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. Its features include:

- Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.
- A messaging transport that is agnostic to the content of the payload.
- Three qualities of service for message delivery:

[6] https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

  - "At most once", where messages are delivered according to the best efforts of the operating environment. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  - " At least once", where messages are assured to arrive but duplicates can occur.
  - " Exactly once", where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
- A small transport overhead and protocol exchanges minimized to reduce network traffic.
- A mechanism to notify interested parties when an abnormal disconnection occurs." [19]

The MQTT protocol is standardized by the OASIS MQTT Technical Committee. As we can see from this snippet out of the MQTT specification, MQTT also implements the publish subscribe pattern. Topics in MQTT can be separated into different topic levels indicated by a foward slash ("/" U+002F). An example being " home/lights/device/in". MQTT offers a service to search for topics registered to the MQTT Broker. These search queries offer next to an exact topic search, more broader search queries using wildcards as place holders. There are two type of wildcards that can be used in these search queries: "#" is a multi-level wildcard. Multi-level wildcards can be used to find multiple topics that match the rest of the topic specified in the search query. An example would be: "devices/lights/#". This search query would return all topics containing "devices/lights/" such as "devices/lights/philips"but not topics like "home/devices/lights". if a topic was named "devices/lights" it would also be returned by the above search query. Multi-level wildcards are only permitted to be used at the end of a search query after a foward slash ("/" U+002F). This means, while "devices/lights/#" is valid, "devices/lights#" is not. The only exception to the rule is the search query "#". This query returns all available topics. Next to the multi-level wildcards there are also single-level wildcards specified by a plus sign ("+" U+002B). Unlike the multi-level wildcard this wildcard can be anywhere in the search query, even the first level, as long as it occupies the entire topic layer of the search query. This means queries like "devices/lights+" would be invalid. An example using single-level wildcard would be: "devices/+/devicename". Single-level wildcards can also be used in conjunction with multi-level wildcards forming queries like "devices/+/devicename/#" or "+/lights/+/inputs/#" [7]. MQTT also offers the possibility of having messages posted to a topic for only certain amount of time by defining a "message expiry interval" and a "retain flag". The MQTT broker will also retain the message for all subscribers, currently not connected to the broker for a set amount of time, which again is defined by the "message expiry interval". Just like in ROS, subscriptions in MQTT are handled via subscription requests. If a MQTT client would like to (un)subscribe to a topic, it sends a request to the MQTT broker with the topic it wishes to (un)subscribe to. The MQTT broker will then respond with either a positive acknowledge message (ACK message) or if there were any problems a negative ACK message. In any case the MQTT Broker will supply a "Reason Code" in the payload of the ACK message. These codes contain in a postive-ACK-case the QoS code. "0" being "at most once", "1" being "at least once" and "2" being "exactly once". In the case of a negative ACK the "Reason Codes" can be quite diverse. These range from a "Unspecified Error", to "Topic Filter invalid" for invalid topic filter (synonymous with search query we talked about above), to "Not Authorized" for topics that are not available to to all

[7]`https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html` 4.7 Topic Names and Topic Filters

clients, to unsupported features, such as for shared subscriptions with SShared Subscriptions not supported". Shared subscriptions are another feature of newer versions of MQTT (MQTT version 5) that allow certain subscriptions to function differently than regular subscriptions. In regular subscriptions every time a message is published to a topic each subscriber receives a copy of the published message. In shared subscriptions every time a message is published to the topic, only one subscriber receives the published message. This is done in an alternating fashion so every subscriber of the topic will eventually receive a message from the broker. This can be seen as a load balancing mechanism for clients, allowing applications to scale horizontally. So if there is a lot of message through-put on a topic clients won't be as easily overloaded depending on the amount of clients subscribed to the topic. Shared subscriptions follow a predefined topic structure ("$share/GROUPID/TOPIC") and is divided into 3 parts: the first part "$share" informs the MQTT broker that this is a shared topic. The second part "GROUPID" specifies which group the topic belongs to. A group in this sense is a collection of clients interested in a certain group of topics. The last part is the actual topic. This topic follows the same rules a specified above. MQTT also implements an End-to-end Acknowledgement. Usually the sender and receiver are completely decoupled from each other. This can be problematic as in some usecase the receiver would like to know if the recipient of a message has received the message. As a result with newer versions of MQTT (version 5) response topics are introduced. These response topics are a optional field in a publish or connect packets, that if filled with information will be identified as a request by the recipient. Lastly MQTT also offers a feature for ungraceful disconnects. This feature is called "Last Will and Testament" (LWT). MQTT clients can specify a last will message to the broker, which the broker will send to all subscribers if the client ungracefully disconnects from the broker. A last will message is like a regular MQTT message containing a topic, payload, QoS and retained message flag. A client can specify a last will message for each topic it publishes to. [19]

7 Code Implementation

The implemented software is divided into 4 different packages: ***proxy_layer, discovery_layer, application_layer*** and ***more_ interfaces***.
These are standard ROS packages containing buildin scripts, ROS node definitions, message and service definitions and program scripts. The *more_interfaces* package is distinctly different from the other three packages as it contains nothing but additional *msg*- and *srv*-files. These are used by our created ROS nodes at run time to communicate with each other. *msg* files describe custom message contents for message busses. And *srv* files describe custom request and response contents for services. We will look at these different files as we encounter them in our other packages. Compilation of our software will yield us 3 ROS Nodes: a Node called Lights Service, the discovery manager and the mqtt Gateway. Each represent a layer in out architecture in the order Application Layer(Lights Service), Discovery Layer (discovery manager) and Proxy Layer(MQTT Gateway). The MQTT Gateway implements an interface for MQTT which allows the software to communicate a MQTT Brokers.
For the following source Code we assume all received MQTT topics follow the structure *"HOME-/SERVICE_TYPE/DEVICE_ID/CHANNEL"*. ***"HOME"*** represents the ROOT topic of all devices to be available to this software. ***"SERVICE_TYPE"*** is the type of service a device is offering, e.g. lights, microphone, window/door-opener. ***"DEVICE_ID"*** is a unique identifier for each device. Lastly ***"CHANNEL"*** represents one of two options: firstly an output channel, in which the device itself publishes all its data, and secondly an input channel in which the device can receive in input from other devices e.g. commands.

7.1 ROS Node Interactions

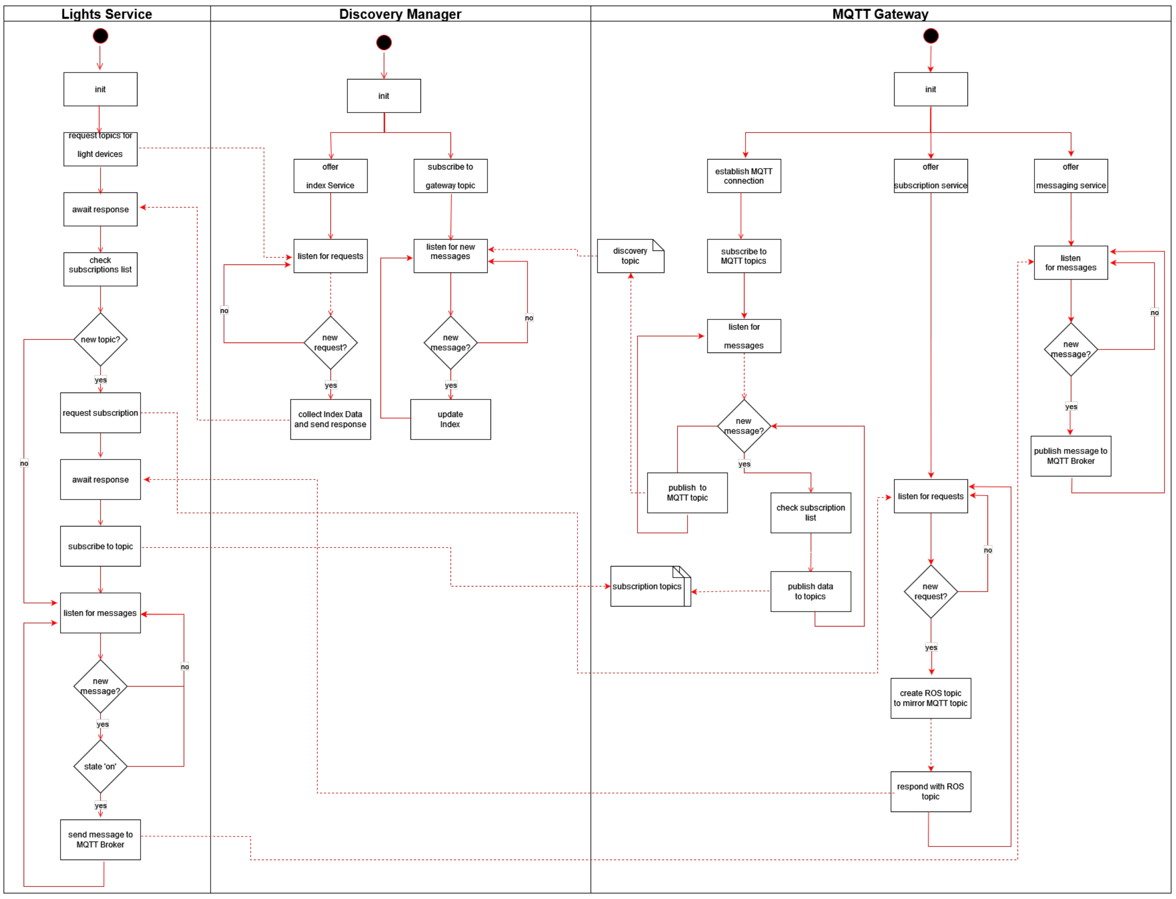

Figure 7.1: ROS Node Relationship

In **Figure 7.1** we can see a flow chart showing the core functions of each service and their interactions with each other. The discovery manager starts with to parallel running processes. In the first it subscribes to a ROS topic dedicated to device discovery. Each time it receives a new message from the topic it will update its index, the data structure containing all information pertaining ot discovered devices. In a second process the discovery offers a service, in which other ROS nodes can request specified parts of the index.
The MQTT Gateway, representing a Gateway in the Proxy Layer, handles 3 separate processes. of two being services. In the first process the MQTT Gateway handles all functions connected to connecting, communicating and receiving messages from a MQTT Broker. This process will also forward any messages to Nodes interested in certain topics. This is done by creating ROS topics mirroring the original MQTT topic. Interested ROS nodes can request a subscription of these topics. This handled by the offered subscription serive in its second process. In the final process of the MQTT Gateway other ROS nodes can use this node to send messages directly to the MQTT Broker. This is implemented via a ROS topic subscription.
Our final ROS node, the lights service, implements an exemplary service handling light devices. This service will request information concerning all light-devices from the discovery manager.

After receiving this information it will check if it knows the device already. If not it will request a subscription from the MQTT Gateway. After going through all the received information received from the discovery manager, it will continue to listening to all subscribed topics. If it receives a message from a device stating, that its light is currently switched on, it will a message over the MQTT Gateway to switch it back off.
Now that we have a general understanding on how these ROS nodes function, we will now have a closer look in how each Node works in more detail by looking at their source code.

7.2 Proxy layer

Our proxy layer consists of one file called "mqtt_publisher.py". This python file implements a gateway for our ROS network to communicate with an MQTT server. We can see how to set up the client in Listing 7.1. Note we use the Paho Python - MQTT Client Library for this.

Listing 7.1 mqtt_publisher: MQTT client setup

```
1    # setting up MQTT client
2    self.client = mqtt.Client()
3    # defining callback methods
4    self.client.on_connect = MQTTClient.on_connect
5    self.client.on_message = MQTTClient.on_message

6    # establishing async connection to MQTT Broker
7    self.client.connect_async(MQTT_BROKER_HOST, MQTT_BROKER_PORT, MQTT_KEEP_ALIVE)

8    # call that processes network traffic, dispatches callbacks and
9    # handles reconnecting.
10    self.client.loop_start()
```

In line 2 we initialise the mqtt client class and create a client object. For this client object we override and define two new custom methods for *on_connect* and *on_message* in lines 4 and 5. These are callback methods. Callback methods are called when certain requirements are met. *on_connect* is a method that is called when this client object connects with a MQTT Broker (server) and receives a response.
on_message is called when this client object receives any other message from the MQTT Broker. In line 7 we use the client object to establish an asynchronous connection with the MQQT Broker. This connection has to be called asynchronous so our program can continue working while the connection is established. Calling this method normally will stop the program of doing anything else until the connection is severed. The connection is established via the TCP protocol [1], which requires the host name of the Broker and the port it can be accessed by. The parameter *MQTT_KEEP_ALIVE* contains the amount of second before this client checks if the connection with the MQQT Broker is still alive.

[1]https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

In line 10 we start a loop that automatically handles any callbacks (e.g. receiving incomming messages) and re-connections.

Now let's have a look at the defined callback methods in lines 4 and 5 in **Listing 7.1**:

Listing 7.2 mqtt_publisher: MQTT client callbacks

```
    1   #overriding methods of the MQTT library
    2   class MQTTClient:

    3       # The callback for when the client receives a CONNACK response from the MQTT
server.
    4       def on_connect(client, userdata, flags, rc):
    5          print("MQTT client Connected with result code " + str(rc))

    6           # Subscribing in on_connect() means that if we lose the connection and
    7           # reconnect then subscriptions will be renewed.
    8           client.subscribe(ROOT_TOPIC_MQTT + "#")

    9       # The callback for when a message is received from the MQTT server.
    10       def on_message(client, userdata, msg):
    11          messageQueue.append(msg)
```

In lines 4-8 we define our *on_connect* callback method. In line 5 we only print out the result code of the connection. If everything went well the Code will be "0". In line 8 we then send a subscription request for all topics under the root we define with *ROOT_TOPIC_MQTT* and the multilevel wildcard "#".
In lines 10-11 we define our *on_message* callback method. This method contains a single line where we add the received MQQT message to a global message queue.

Now that we have looked at how our MQTT client works we will look at our second major object in this file: The ROS Node. this Node is defined by the class MQTTGateway shown in **Listing 7.3**, **Listing 7.4** and **Listing 7.5**:

Listing 7.3 mqtt_publisher: MQTTGateway

```
class MQTTGateway(Node):

1  def __init__(self):
2        #ROS publisher
3        super().__init__('mqtt_publisher')

4        # init topic publisher
5        self.publisher = self.create_publisher(MqttGatewayIn, ROOT_TOPIC_ROS + '
received_topics', 10)

6        # init subscription for messages to be sent to MQTT broker
7        self.subscription = self.create_subscription(MqttGatewayOut, ROOT_TOPIC_ROS + 'out',
self.send_message, 10)

8        # init subscription service
9        self.service = self.create_service(SubscribeTopic, 'subscribe_to_mqtt_topic', self.
subscribe_to_mqtt_topic_callback )

10        # checks for messages continuously and publishes them to ROS network
11        timer_period = 0.5
12        self.timer = self.create_timer(timer_period, self.publish_message)
```

In **Listing 7.3** we can see the constructor method of our ROS Node. Constructors are executed for each Object created. In line 3 we declare its name in the ROS network. In the following lines 5-9 we declare a publisher, a subscription and a service our ROS Node will be offering. Each of these created objects contains a message template described by either a msg-file or a srv-file which we menttioned in **Section 6.1**. Finally in lines 11-12 we declare a timer, that executes the method *publish_message* every half-second.

Listing 7.4 mqtt_publisher: MQTTGateway

```
class MQTTGateway(Node):
1    # publishes received MQTT messages to designated ROS topic
2    def publish_subscribed_messages(self, message):

3        if message.topic in subscription_mapper.keys():
4            # find correct publisher object for topic
5            self.ros_publisher = subscription_mapper.get(message.topic)

6            # create new ROS msg Object
7            sub_data = SubscriptionData()
8            sub_data.topic = message.topic
9            sub_data.data = str(message.payload)

10            # publish message
11            self.ros_publisher.publish(sub_data)



12    # publishes topics of received messages to ROS network
13    def publish_message(self):

14        if messageQueue:

15            message = messageQueue.pop(0)


16            # create new ROS msg object
17            ros_message = MqttGatewayIn()
18            ros_message.gateway = GATEWAY_ID
19            ros_message.data = message.topic
20            # publishes topic for discovery
21            self.publisher.publish(ros_message)

22            self.get_logger().info('Publishing: "%s"' % ros_message)

23            # publishes message contents to designated ROS topics (if available)
24            self.publish_subscribed_messages(message)
```

In **Listing 7.4** we can see an implementation of the method *publish_message* called in lines 11-12 in **Listing 7.3**. In line 14 of **Listing 7.4** the ROS Node checks for received messages in the message queue. If there is a message the ROS Node creates a new message and publishes the message to the ROS Network. Contents of the message are the gateway this message is coming from, as also the MQTT topic of the received MQTT message. Then after publishing this message this method calls the *publish_subscribed_messages* method. This method (described in lines 2-11) publishes contents of these MQTT messages to ROS topics. If a message gets published to a ROS topic depends on if other ROS nodes have requested the contents of these MQTT topic. This information is saved in the dictionary *subscription_mapper* shown in line 3. A dictionary is a data structure in the python programming language where 2 sets of data can be linked. In this dictionary the MQTT topic and the ROS publisher object, used for publishing the data of this topic, are linked. Now if there is an entry for the MQTT topic, the ROS node creates a message and publishes this message via the correct publisher object. The message contains a field for the MQTT topic and a field for its data.

Listing 7.5 mqtt_publisher: MQTTGateway

```
class MQTTGateway(Node):

1    # publishes data to MQTT Broker
2    def send_message(self, msg):
3        self.get_logger().info('Publishing Message to MQTT topic: "%s"' % msg.topic)
4        self.client.publish(msg.topic, msg.data)

5    # topic subscription service callback method offered by this Node
6    def subscribe_to_mqtt_topic_callback(self, request, response):

7        # MQTT topics have syntax: HOME/SERVICE_TYPE/DEVICE_ID
8        topic_elements = request.topic.split('/')
9        deviceID = topic_elements[2]

10        ros_topic = ROOT_TOPIC_ROS + 'received_data/' + str(deviceID)

11        # create and publish new ROS topic
12        ros_publisher_= self.create_publisher(SubscriptionData, ros_topic, 10)
13        # add publisher obj to the mapper
14        subscription_mapper[request.topic] = ros_publisher_

15        # return ROS topic for MQTT topic to requester
16        response.data = ros_topic
17        return response
```

Now in **Listing 7.5** in the lines 6-17 callback method for the subscription service of our ROS node is described. This service offers other ROS nodes the opportunity to subscribe to MQTT topics over this ROS node and receive their messages. This method received both the request message object as also the response message object. The request object contains a single field containing the MQTT topic, the requester would like to subscribe to. The service takes this topic and extracts the device identifier (lines 8-9) and constructs a ROS topic with this information(line 10). The the service creates a publisher object for this ROS topic (line 12) and adds it to the *subscription_mapper* with the MQTT topic as a key. Finally the service adds the created ROS topic to the response message. The requesting ROS node can then subscribe to this topic to receive any messages from the request MQTT topic.

7.3 Discovery Layer

The discovery layer, the same as the proxy layer, consists of one filed “discovery_manager”. This file creates to the time of initialisation one ROS node. This Node collects all the information published by the Gateway Node in the proxy layer. Also it offers a service to provide parts of this collected data to requesting ROS nodes.
To collected information about other devices, the discovery manager Node subscribes to the topic, to which the MQTT Gateway Node publishes all received topics from the MQTT Broker. This is shown in line 4 of **Listing 7.6**. A more elegant solution would be having the MQTT Gateway request a list of all topics from the MQTT Broker. Unlike ROS, MQTT currently does not offer this kind of feature, which is why this kind of discovery may not discover all devices. Devices that only publish a message under specific conditions might be missed by this discovery approach. Therefore we make the assumption in this Code that all devices publish messages periodically.

Listing 7.6 discovery_manager: Initialisation

```
class DataManager(Node):

1   def __init__(self):
2        super().__init__('discovery_manager')

3        # init subscription of all received MQTT topics over the MQTT Gateway
4        self.subscription = self.create_subscription(MqttGatewayIn, MQTT_GATEWAY + '
received_topics', self.update_index, 10)

5        # init service offering a collection of the collected data for a specified
SERVICE_TYPE
6        self.service = self.create_service(DataRequest, 'getIndexData' , self.
send_index_data_callback)
7        # init index
8        self.index = {}
```

As callback to any new message received by the MQTT Gateway the method *update_index* is called (see **Listing 7.7**). In line 6 the data manager node initalises the service offered, to send parts of all collected data to the requester. the callback method for this service is called *send_index_data_callback* and is shown in **Listing 7.8**.

Listing 7.7 discovery_manager: Update Index

```
class DataManager(Node):

1    # subscription method keeps index up-to-date
2    def update_index(self, msg):
3        self.get_logger().info('I heard: "%s" from Gateway' % msg.data)

4        gateway = msg.gateway

5        # message received from MQTT Gateway
6        if gateway == MQTT_ID:

7            topic = msg.data
8            topic_elements = topic.split('/')

9            # ROS topics follow pattern: HOME/SERVICE_TYPE/DEVICE_ID
10            service_type = topic_elements[1]

11            if service_type in self.index.keys():

12                if not (gateway,topic) in self.index.get(service_type):

13                    # retrieve current topic list for SERVICE_TYPE and adds new information
to the end of the list
14                    newlist = self.index.get(service_type).append((gateway,topic))
15                    self.index[service_type] = newlist

16            else:
17                # create a new SERVICE_TYPE and add first entry
18                newlist = [(gateway, topic)]
19                self.index[service_type] = newlist
```

Every time the discovery manager node receives a message from the proxy layer it checks from which gateway the message was sent. As we only have one gateway implemented, namely the MQTT Gateway, this method only knows how to handle MQTT topics. Cases for other Gateways would have to be added here in this method. If the message comes from the MQTT Gateway this method will first separate the received topic into its different parts. The method assumes the received mqtt follows a specific pattern: “home/SERVICE_TYPE/DEVICE_ID/out”. “HOME” is the root of our MQTT topics. “SERVICE_TYPE” specifies what kind of service the device offers that publishes to this topic. And “DEVICE_ID” being the identifier of the device. If the service type exists (line 12) in the index (being the collection of all collected information), the discovery manager checks if it has received the topic from the gateway before. If it doesn’t, the topic gets added to the index (lines 11-15). If the service type doesn’t exist in the index at all, the service type is added to the index together with topic as its first item, linked to the service type (lines 16-19).

Listing 7.8 discovery_manager: Data Request Service

```
class DataManager(Node):

1    # request service method. returns all entries of a specified SERVICE_TYPE.
2    def send_index_data_callback(self, request, response):
3        self.get_logger().info('I got a Request for: "%s" ' % request.servicetype)
4        servicetype = request.servicetype

5        responselist = []

6        # retrieve all current entries for given SERVICE_TYPE
7        servicetypelist = self.index.get(servicetype)

8        if servicetypelist:
9            # to send a list of lists over the ROS network we need a new ROS msg file to
represent a list and another msg file representing a list of this list
10           # StringList will be this intermediary list of strings
11           for element in servicetypelist:

12               stringlist = StringList()
13               # note each entry in the tuple in "element" is added as one separate element
 in the list via the "extend" function
14               stringlist.elements.extend(element)

15               responselist.append(stringlist)

16        # respone object represents a list of stringlist objects
17        response.servicelist = responselist
18        return response
```

As mentioned above, next to collecting data from the proxy layer, the discovery manager also offers other Nodes the opportunity to request parts of this collected data. This is done by offering a service shown in **Listing 7.8**. The service starts by reading the request it received (line 4) and checks if the service type exists in the index (lines 7-8). If the service type exists, this service will create a list out of every entry in the index (line 11-14). This list includes the gateway the data was received from and the contact data. In currently it only will be the topic as no further gateways are currently implemented. Each created list will then be added to the response list (line 15), making the response object a list of lists. To being able to send a list of lists we create an intermediate *msg*-file called *StringList*. This *msg*-file has a single field containing a list of string objects.

7.4 Application Layer

In our last layer, the Application Layer, we have one example service implemented in file “light_service”. This service at its core checks for light devices in the network. Depending on the state of a device, this service will try and interact with it. To be more precise this service will check if a light device is currently switched on. If it is turned on, it will send a message to the light device to switch it back off. Now let us take a closer look on how this service exactly. For this we will now look at the main-method shown in **Listing 7.9**:

Listing 7.9 light_serive: Main Method

```
class LightService(Node):

    def main(args=None):
        rclpy.init(args=args)
        lights_service = LightService()

        # awaits service avaiability
        while not lights_service.client.wait_for_service(timeout_sec=1.0):
           lights_service.get_logger().info('service not available, waiting again...')

        # requests a list of light devices from the discovery manager
        # future is the response for the service
        future = lights_service.send_request()

        # wait until service responds
        rclpy.spin_until_future_complete(lights_service, future)

        lights_service.get_logger().info('I got a response from the discovery manager')
        result = future.result()

        #servicelist_ is of type StringList
        #elements of StringList are StringList.elements and contain elements [gateway, topic]
        for stringlist_ in result.servicelist:

            topic = stringlist_.elements[1]
            subTuple = lights_service.check_subscription(topic)

            # if device ID is not in the subscription list
            if subTuple[0]:

                # request a subscription from the MQTT gateway
                response = lights_service.request_mqtt_subscription(topic)
                rclpy.spin_until_future_complete(lights_service, response)

                # gateway responds with a ROS topic this service can subscribe to to access requested topic's content
                subscription_response = response.result()
                # create the subscription to the ROS topic
                subscriber = lights_service.create_subscription(SubscriptionData,
                              subscription_response.data,
                              lights_service.mqtt_subscription_callback,
                              10)
                # add Device ID to the subscription list
                lights_service.mqtt_subscriptions.append(subTuple[1])
        rclpy.spin(lights_service)
```

In the first two lines of the main method initialise the ROS environment and the ROS node for this service. In line 5 the ROS node then waits until the service offered by the discovery_manager in **Listing 7.8** is available in the ROS network. In lines 8-10 the ROS Node then sends a request for all discovered light devices to the discovery_manager and then waits for the discovery_manager to reply. In line 12 we access the response received by the discovery_manager. the respons should contain a list of lists. These sub-lists are entries in the index that represent a light device. As we have only implemented the MQTT Gateway, all entries will consist of the gateway idenfier and the MQTT topic. For each entry this ROS Node receives it will check if its currently already subscribed to the topic (line 17-19). If this Node is currently not subscribed to the topic, it will request a subscription from the MQTT Gateway (line 21-22). After receiving a response from the MQTT Gateway, the ROS Node will subscribe to the ROS topic received from the MQTT Gateway. On this topic, the Gateway will publish all received message from the MQTT Broker for the requested MQTT topic. This Node will also add a reference in a list called mqtt_subscriptions to avoid duplicate subscription requests. Finally after handling all subscriptions this Node will then only be listening and responding to any message it receives from its subscriptions (line 29). The callback method for these can be seen in **Listing 7.10**:

Listing 7.10 light_serive: Subscription Callback

```
class LightService(Node):

1    def mqtt_subscription_callback(self, message):

2        self.get_logger().info('Got Subscription Message: ' + message.data + ' from topic: '
+ message.topic)

3        # checks if message matches 'on'
4        if str(message.data).lower() == STATE_IS_ON:

5            # send message to turn device off
6            # init msg object
7            mqtt_msg = MqttGatewayOut()
8            # switch ROS topic for outputs to ROS topic for inputs of a device
9            mqtt_msg.topic = self.change_topic_channel(message.topic, TOPIC_ENDING_IN,
TOPIC_ENDING_OUT)
10           mqtt_msg.data = 'off'
11           # publish message
12           self.publisher.publish(mqtt_msg)
13           self.get_logger().info("Turing device with topic %s off" + message.topic)

14        # checks if message matches 'off'
15        if str(message.data).lower() == STATE_IS_OFF:
16            self.get_logger().info("Light of device on topic %s is off" + message.topic)
```

For every message the Node receives it checks the contents of the message (line 4). If the message contents matches the saved state for “Off” the Node will log this. If it matches the state for “On” instead, the Node will create a message which tells the device to turn off (lines 7 -10). This is message is sent to the MQTT Gateway. This is done via publishing to a ROS topic (line 12) to which the MQTT Gateway subscribes to for sending messages to the MQTT Broker. The contents of the sent message is the input topic for the device (created in line 9) and the command to turn off the light of the target device.

8 Testing of the Prototype

To test the performance of our prototype we will run this software against an existing MQTT server. To provide the software with the necessary input data, we will simulate 3 different devices. Two devices imitating light services and one device offering a service for a window-opening-mechanism. The corresponding device topics are:

- for input channels:
  - ***home/lights/roomlights405/in***
  - ***home/lights/philips165/in***
  - ***home/window/firstlevel1/in***
- for output channels:
  - ***home/lights/roomlights405/out***
  - ***home/lights/philips165/out***
  - ***home/window/firstlevel1/out***

As we are simulation all three devices, we are going to make use of a MQTT Webclient to publish to the output-channels and subscribe to the input-channels of these devices. We will publish 10 messages per topic, making a total of 30 messages sent bei the Webclient. 5 of each will be published after the Nodes from the Proxy Layer and Discovery Layer have been start. 5 Messages will be sent after the light service node in the application Layer has been started. In our test case we will assume that one light is switched on to begin of the test, while the other light is switched off. This light will turn its state to off if receiving a message from the robot to turn off.

8.1 Test Results

For reasons of clarity and visibility we include only those parts of the logs where changes occurs. Logs containing duplicate messages were sent are therefore being cut.

Listing 8.1 mqtt_publisher: Logs

```
[INFO] [1639760965.475796600] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/philips165/out')"
[INFO] [1639760983.376557000] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/out')"
[INFO] [1639761019.374509100] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/window/firstlevel1/out')"
---- light service is started here ----
[INFO] [1639761117.871529900] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/philips165/out')"
[INFO] [1639761156.381938300] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/out')"
[INFO] [1639761156.391886200] [mqtt_publisher]: Publishing Message to MQTT topic: "home425/
lights/roomlights405/in"
[INFO] [1639761156.879453100] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/in')"
[INFO] [1639761209.377847600] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/out')"
[INFO] [1639761209.387764000] [mqtt_publisher]: Publishing Message to MQTT topic: "home425/
lights/roomlights405/in"
[INFO] [1639761209.881923100] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/in')"
[INFO] [1639761210.368774200] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/out')"
[INFO] [1639761210.377785400] [mqtt_publisher]: Publishing Message to MQTT topic: "home425/
lights/roomlights405/in"
[INFO] [1639761210.880438900] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/in')"
[INFO] [1639761211.372656000] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/out')"
[INFO] [1639761211.381855200] [mqtt_publisher]: Publishing Message to MQTT topic: "home425/
lights/roomlights405/in"
[INFO] [1639761211.874713600] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/in')"
[INFO] [1639761213.882544600] [mqtt_publisher]: Publishing: "more_interfaces.msg.MqttGatewayIn
(gateway='mqtt', data='home425/lights/roomlights405/out')"
```

The Logs for the MQTT Gateway (shown in **Listing 8.1**) show that messages from three different topics are received from the MQTT Broker (15 out of 15 messages were received). After the light service is started we can see the MQTT Gateway receiving messages and publishing them to the MQTT Network via the MQTT Broker. The Gateway also receives the same message from the MQTT Broker after sending.

Listing 8.2 discovery_manager: Logs

```
[INFO] [1639760965.456265300] [discovery_manager]: I heard: "home425/lights/philips165/out"
from Gateway
{'lights': [('mqtt', 'home425/lights/philips165/out')]}

[INFO] [1639760983.376408600] [discovery_manager]: I heard: "home425/lights/roomlights405/out"
 from Gateway
{'lights': [('mqtt', 'home425/lights/philips165/out'), ('mqtt', 'home425/lights/roomlights405/
out')]}

[INFO] [1639761019.374755400] [discovery_manager]: I heard: "home425/window/firstlevel1/out"
from Gateway
{'lights': [('mqtt', 'home425/lights/philips165/out'), ('mqtt', 'home425/lights/roomlights405/
out')], 'window': [('mqtt', 'home425/window/firstlevel1/out')]}

---- light service is started here ----
[INFO] [1639761043.042854300] [discovery_manager]: I got a Request for: "lights"
[INFO] [1639761072.378545900] [discovery_manager]: I heard: "home425/window/firstlevel1/out"
from Gateway
---- no new data is added to the index -----
```

In the Logs of the discovery manager (shown in **Listing 8.2**) we can see the discovery manager is receiving data from the MQTT Gateway. After encountering a new device for the first time, we can see it being added to the index. For clarity the content of the index is logged too. It's content is inside the { and } brackets. Here we can see that each device has been added. After the lights service started we can see from the logs, that the light service sent a request to the discovery manager. Although not visible here in these logs, the discovery manager also received all messages from the MQTT Gateway (30 out of 30 messages including the messages sent to the MQTT Broker by MQTT Gateway, as the Gateway also receives a copy of their message from the MQTT Broker).

Listing 8.3 light_serive: Logs

```
[INFO] [1639761043.039954700] [light_service]: sending request to discovery manager
[INFO] [1639761043.046713100] [light_service]: I got a response from the discovery manager
[INFO] [1639761043.048788200] [light_service]: sending subscription request for topic: home425
/lights/philips165/out
[INFO] [1639761043.065250200] [light_service]: sending subscription request for topic: home425
/lights/roomlights405/out
[INFO] [1639761117.877741600] [light_service]: Got Subscription Message: b'off' from topic:
home425/lights/philips165/out
[INFO] [1639761117.882294400] [light_service]: Light of device on topic home425/lights/
philips165/out are off
[INFO] [1639761130.376319700] [light_service]: Got Subscription Message: b'off' from topic:
home425/lights/philips165/out
[INFO] [1639761130.379857800] [light_service]: Light of device on topic home425/lights/
philips165/out are off
[INFO] [1639761132.378608100] [light_service]: Got Subscription Message: b'off' from topic:
home425/lights/philips165/out
[INFO] [1639761132.383205400] [light_service]: Light of device on topic offhome425/lights/
philips165/out are off
[INFO] [1639761133.876367500] [light_service]: Got Subscription Message: b'off' from topic:
home425/lights/philips165/out
[INFO] [1639761133.879915800] [light_service]: Light of device on topic home425/lights/
philips165/out are off
[INFO] [1639761145.873732700] [light_service]: Got Subscription Message: b'off' from topic:
home425/lights/philips165/out
[INFO] [1639761145.877669000] [light_service]: Light of device on topic home425/lights/
philips165/out are off
[INFO] [1639761156.387136800] [light_service]: Got Subscription Message: b'on' from topic:
home425/lights/roomlights405/out
[INFO] [1639761156.391659600] [light_service]: Turing device with topic home425/lights/
roomlights405/out off
[INFO] [1639761209.382834700] [light_service]: Got Subscription Message: b'on' from topic:
home425/lights/roomlights405/out
[INFO] [1639761209.387513100] [light_service]: Turing device with topic home425/lights/
roomlights405/out off
[INFO] [1639761210.373989100] [light_service]: Got Subscription Message: b'on' from topic:
home425/lights/roomlights405/out
[INFO] [1639761210.377566500] [light_service]: Turing device with topic home425/lights/
roomlights405/out off
[INFO] [1639761211.377487400] [light_service]: Got Subscription Message: b'on' from topic:
home425/lights/roomlights405/out
[INFO] [1639761211.381709200] [light_service]: Turing device with topic home425/lights/
roomlights405/out off
[INFO] [1639761213.887358900] [light_service]: Got Subscription Message: b'off' from topic:
home425/lights/roomlights405/out
[INFO] [1639761213.890784200] [light_service]: Light of device on topic home425/lights/
roomlights405/out are off
```

From the logs of the lights service (shown in **Listing 8.3**) we can see it sending a request to the discovery manager and receiving a response. We can see it then requesting two subscriptions from the MQTT Gateway. Afterwards we can see it receiving 10 out of the 15 Messages we published to the MQTT Network, responding to 4 out of these 10 Messages.

Listing 8.4 ROS Topic List

```
/gateway/mqtt/out
/gateway/mqtt/received_data/philips165
/gateway/mqtt/received_data/roomlights405
/gateway/mqtt/received_topics
/parameter_events
/rosout
```

In **Listing 8.4** we can see a list of all ROS topics to run time. The first topic being the topic for messages to be sent to the MQTT Network. The second and third topic being device topics created by the MQTT Gateway to mirror MQTT topics. The fourth topic is the topic the Gateway publishes to for the discovery of devices. The discovery manager is subscribed to this topic. The last two topics are default ROS topics and are therefore uninteresting for us.

8.2 Interpretation of Test Results

From the logs in **Listing 8.1** we can see the Gateway properly communicates with the MQTT Broker. It receives messages from the Broker and also successfully sends messages to the Broker. One side effect of how the Gateway subscribes to the MQTT topics is that the Gateway receives its own messages from the Broker. As the Gateway subscribes to the highest level topic pf our devices in the MQTT Network, it subscribes to both the input- as also output-channels of each device. To avoid this the Gateway could manually unsubscribe from the input-channels. This is something worth considering as processing ucessary messages have an effect on performance and message throughput. All messages get published at a minimum to the discovery topic. Meaning the discovery layer will try to update their index every time a message is received, even if its a faulty one. Lastly from the topic list in **Listing 8.4** and logs in **Listing 8.3** we can tell that mirroring of MQTT topics via ROS topics is also working as intended.

From the logs in **Listing 8.2** we can tell that the discovery manager properly adds new devices to its index after receiving messages form the MQTT Gateway. Also it responds respond to service requests for index data. This is shown by its own logs but also the logs shown in **Listing 8.3** by the lights service. The lights service receives data for two devices from the index and then requests two subscriptions from the MQTT Gateway for these two devices. As the lights service receives messages and is able to respond to messages and the topics for two devices are present (**Listing 8.4**, we can say the interaction between lights service and the MQTT Network works as intended.

9 Conclusion and Outlook

In Conclusion we can say that we have created and implemented an architecture that enables a robot to find and communicate with surrounding devices. It can send and receive messages over different Gateway Nodes in the Proxy Layer, which each implement a different communication protocol. It can discover new devices over these gateways and collect the information required to contact these devices in an Index (Discovery Layer). Then finally distribute this information within the Robots Runtime Environment and make it available to other Nodes withing Robots Runtime Environment. With this information the Robot can then contact and interact with discovered nearby devices. This implementation does show some limitations though which have to be considered: Although it is a strength of this architecture to realise all different components as services/ROS nodes, allowing to clear definitions and separation of concerns as also allowing high flexibility with the architecture, implementing Gateways in the Proxy Layer in this way brings some disadvantages. ROS, unlike other frameworks, is not designed to scale as classical service oriented frameworks are designed to do. As a result it can occur, that a gateway node receives more messages than it can work off. In other words the gateway node is overloaded and as a direct result messages are lost. ROS has no load balancing feature for topic subscription as MQTT has in its newer versions. Another issue that arises is how we implement the passing on of messages received from the gateways. To make contents of messages received by the Gateway available for other ROS nodes a topic is created for each distinct device a message is received from. This can lead to a ROS node being subscribed to a multitude of topics. Also there seems to be no checking if a device is still active/ available in the network if there is no notification of removal of a device. As a result a topic might end up obsolete. To get around the multitude of topics created we can change how topic creation is done in this architecture. Instead of creating a new topic when subscribing to a devices messages, the gateway adds the name of the interested service to a listener list. Then when a new message is received the Gateway publishes the message to the default topic of each interested service. The downside for this approach is that a service as to make itself known to the gateway first, before being able to receive messages from the gateway. On the other hand the topic overhead can be decreased significantly.

Outlook

With this robot-assistants are able to assist and interact with its environment in new ways. Although only in limited way. These robots still wont be able to know where a connected device is physically located and its distance to it. Therefore if a robot is trying to connect with a particular device in its environment, it has no means of verifying if it indeed connected to the desired device. Future work could build on this and provide a means to assist the robot in finding out if it connected to the correct device through for example visual feedback. As we have seen in our conclusion some parts of our implementation can be improved upon, while the proposed architecture has proven successful in our testing. Also adding a way to dynamically deploy new services to the robot to better serve the needs of occupants of a household or building as exemplified in [GNN+17] will prove invaluable for the future of home assisting robots. The primary goal of home assisting robots is to fulfill the needs of the house owner as best possible. To fulfill those needs a static set of rules and routines will not be enough to satisfy every owners demands. A more flexible customize-able set of qualities, rules and routines is therefore required to fit and fulfill the needs of the owner. By keeping this architecture service orientated, rules, qualities, services and features of the robot can remain very flexible, changeable and therefore also adaptable. These changes can either be done by hands of developers or as we have seen in [GNN+17], use an artificial intelligence (A.I.) to observe the behavioural patterns of the owner and have it create and deploy services that are made to fit the needs of the owner perfectly. The communication between the robot and the A.I. could also be done over the proposed Gateway system. Finally by introducing the ability for a robot to communicate with other devices, we create new vulnerabilities in the security of the robots software. As we do not cover any security features in this work, future work might also offer a solution malicious messages by correctly identifying and handling them, for example through purifying message contents. The usage of a Gateway system allows our application to in theory create an interface for every kind of transporting protocol and messaging system. The problem here being that not all transporting protocols were designed with security in mind. As a result security would have to be handled for each gateway separately.

Bibliography

[19] *MQTT Version 5.0.* OASIS, Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html (cit. on pp. 26, 27).

[AD08] M. Aiello, S. Dustdar. "Are our homes ready for services? A domotic infrastructure based on the Web service stack". English. In: *Pervasive and Mobile Computing* 4.4 (2008). Relation: http://www.rug.nl/informatica/organisatie/overorganisatie/iwi Rights: University of Groningen. Research Institute for Mathematics and Computing Science (IWI), pp. 506–525. ISSN: 1574-1192 (cit. on p. 15).

[Arn16] C. B. Arne Broering Soumya Kanti Datta. "A Categorization of Discovery Technologies for the Internet of Things". In: (2016) (cit. on p. 15).

[BC01] G. Bieber, J. Carpenter. "Introduction to service-oriented programming (rev 2.1)". In: *OpenWings Whitepaper, April* (2001) (cit. on p. 13).

[CDZS16] P. C. Ccori, L. C. C. De Biase, M. K. Zuffo, F. S. C. da Silva. "Device discovery strategies for the IoT". In: *2016 IEEE International Symposium on Consumer Electronics (ISCE)*. 2016, pp. 97–98. DOI: 10.1109/ISCE.2016.7797388 (cit. on p. 16).

[DDB15] S. K. Datta, R. P. F. Da Costa, C. Bonnet. "Resource discovery in Internet of Things: Current trends and future standardization aspects". In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 2015, pp. 542–547. DOI: 10.1109/WF-IoT.2015.7389112 (cit. on p. 16).

[GA16] I. Georgievski, M. Aiello. "Automated Planning for Ubiquitous Computing". In: *ACM Comput. Surv.* 49.4 (Dec. 2016). ISSN: 0360-0300. DOI: 10.1145/3004294. URL: https://doi.org/10.1145/3004294 (cit. on pp. 11, 13, 16, 17).

[GNN+17] I. Georgievski, T. Nguyen, F. Nizamic, B. Setz, A. Lazovik, M. Aiello. "Planning meets activity recognition: Service coordination for intelligent buildings". English. In: *Pervasive and Mobile Computing* 38.Part 1 (July 2017), pp. 110–139. ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2017.02.008 (cit. on pp. 17, 50).

[HDBM13] V. Hax, N. Duarte Filho, S. Botelho, O. Mendizabal. "ROS as a middleware to Internet of Things". In: *Journal of Applied Computing Research* 2 (July 2013), pp. 91–97. DOI: 10.4013/jacr.2012.22.05 (cit. on p. 21).

[SNY18] J. Sunthonlap, P. Nguyen, Z. Ye. *Intelligent Device Discovery in the Internet of Things - Enabling the Robot Society*. 2018. arXiv: 1712.08296 [cs.AI] (cit. on p. 16).

All links were last followed on December 23, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature