Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Automated discovery and binding of IoT devices

Alexandros Fouskas

Course of Study:	Informatik
Examiner:	PD Dr. Holger Schwarz
Supervisor:	Ana Cristina Franco da Silva, M.Sc., Dipl.-Inf. Pascal Hirmer
Commenced:	May 3, 2017
Completed:	November 3, 2017
CR-Classification:	C.2.2, C.2.3

Abstract

The emerging Internet of Things (IoT) leads to new challenges in registering and binding billions of devices. Performed manually, the necessary tasks are time-consuming and error prone, which brings the need for automation. The Resource Management Platform (RMP) provides automated services from the binding of devices to the provisioning of their data. However, the devices need to be registered manually. Thus, this thesis extends the RMP with an automated discovery service, which allows the RMP to discover and register devices using different communication technologies, like WiFi or Bluetooth. Moreover, the work includes a monitoring mechanism, which enables the RMP to automatically deregister unavailable devices. Furthermore, modern smartphones have extended sensing capabilities reaching from positioning to environmental parameters. However, mobile operating systems do not allow access to the internal sensors from outside the phone. Thus, this work describes an Android application, which can be used to integrate smartphones into the RMP.

Kurzfassung

Das Registrieren und Anbinden von Geräten sind zentrale Aufgaben im aufkommenden Internet of Things (IoT). Dies liegt vor allem an der großen Anzahl der zu vernetzenden Geräte. Müssen Milliarden Geräte manuell konfiguriert werden, benötigt das sehr viel Zeit und birgt eine hohe Fehlerwahrscheinlichkeit. Daher bedarf es automatisierter Lösungen. Die Resource Management Platform (RMP) bietet automatisierte Funktionen zum Anbinden von Geräten und zur Bereitstellung der zugehörigen Daten. Jedoch müssen Geräte noch immer manuell registriert werden. Aus diesem Grund wird die RMP in dieser Arbeit um einen automatisierten Discovery Service erweitert. Dieser ermöglicht es Geräte aufzufinden und zu registrieren. Der Service unterstützt hierzu mehrere Netzwerktechnologien, beispielsweise WiFi oder Bluetooth. Darüber hinaus beinhaltet die Arbeit einen Monitoring Mechanismus. Dieser erlaubt es der RMP, nicht mehr verfügbare Geräte automatisch zu erkennen und zu deregistrieren. Des Weiteren bieten moderne Smartphones eine Vielzahl von Sensoren, über Positionsbestimmung bis zur Erfassung von Umgebungsbedingungen. Allerdings erlauben mobile Betriebssysteme keinen Zugriff auf die internen Sensoren durch Applikationen auf anderen Geräten. Daher beinhaltet diese Arbeit auch eine Android Applikation, mit deren Hilfe Smartphones in die RMP integriert und die Daten ihrer Sensoren verfügbar gemacht werden können.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

Bluetooth SDP Bluetooth Service Discovery Protocol. 7, 17, 20

CEP complex event processing. 15

CoAP Constrained Application Protocol. 26

DNS Domain Name System. 18

DNS-SD DNS-Based Service Discovery. 7, 17, 18

IETF Internet Engineering Task Force. 20

IoT Internet of Things. 3, 15

IPP Internet Printing Protocol. 19

mDNS Multicast DNS. 17

P2P Peer-to-Peer. 29

PAN Personal Area Network. 20

RMP Resource Management Platform. 3, 15, 17

SIG Special Interest Group. 20

UUID Universally Unique Identifier. 21

WSN wireless sensor network. 15

1 Introduction

Technological progress and growing digitization lead to new emerging concepts, like the Internet of Things (IoT) [VP13] and Industry 4.0 [Jaz14]. These concepts are based on interconnected devices being able to generate data, communicate, and take appropriate actions without human intervention. With the number of devices growing fast, concepts, like advanced manufacturing [TCZN17], smart homes [GBMP13], or smart cities [VP13], become more and more viable. In this context, *"the integration of sensors and actuators becomes more and more important"* [HBF+17]. However, connecting sensors and actuators to so called IoT environments imposes a great effort in configuring the devices. To use these devices effectively, raw data must be acquired, information must be extracted and combined, and the results must be delivered to applications, that act upon them.

An example for an IoT environment are smart factories [RBBM14]. In such a factory there could, for instance, exist different ambient sensors, like temperature, pressure, or humidity sensors. These sensor readings could be collected by a monitoring system and be provisioned to other applications, e.g., a complex event processing (CEP) system [Luc05]. The CEP system could, for example, recognize critical ambient conditions, such as high temperature and humidity. As a reaction, the system could stop production machines until the critical conditions are gone.

As the landscape of IoT devices and concepts is highly heterogeneous, the tasks, necessary to build a monitoring system for IoT environments, require an advanced knowledge of the technical backgrounds. There are many approaches for such systems, yet, in most of them, devices must be *registered* and *bound* manually [HBF+17]. Manual configuration is time-consuming and error-prone, especially as IoT environments are highly dynamic, with devices entering or exiting the environment frequently. Several approaches to reduce the amount of manual configuration have been introduced, one of which is the Resource Management Platform (RMP) by Hirmer et al. [HBF+17]. The RMP aims on automating the *binding* of devices, sensors, or actuators and the provisioning of their data. However, the devices must be *registered* manually. Thus, the RMP is not able to adapt to changes in the network, such as, devices entering or leaving the environment. As this is a vital capability for the use in the IoT, it is the target of this thesis to extend the RMP with discovery and monitoring abilities. For this purpose, this work examines existing discovery mechanisms, especially those optimized for IoT environments. Moreover, a discovery and monitoring service is designed, which extends the RMP and enables it to register and bind devices joining the network, as well as to deregister devices leaving the network.

To ease their installation, most IoT devices use wireless communication, which lead to the concept of wireless sensor networks (WSNs) [RSZ06]. These networks can build on differ-

ent radio technologies, for example, IEEE 802.11 (WLAN) [Ins16] or Bluetooth [Blu14]. Thus, it is essential that a system like the RMP does not specialize on one radio technology, but supports network monitoring for multiple technologies. As a consequence, the service introduced in this work incorporates a gateway approach to allow the monitoring of multiple networks based on different technologies

In addition to the general problems of using IoT devices, smartphones, the most common IoT *thing*, are hard to include in IoT environments. In 2016, 77% of American adults[1] owned a smartphone carrying it with them every day. Due to technological progress, smartphones contain numerous sensors to track their users or their surroundings [SWS+10]. For example, almost every smartphone contains an accelerometer to track its movements, and a GPS module to determine its position. These sensors can provide data, which might be interesting to IoT applications. However, current mobile operating systems do not allow access to the internal sensors from outside the phone. Thus, there is no standardized way for systems monitoring the environments, such as the RMP, to provision the phone's data. To make this possible, this work includes an application, which has access to the phone's sensors and is able to register the phone at the RMP whenever its network connection changes. Since, according to Gartner [Gar16], the Android OS[2] is by far the most spread operating system for smartphones, this work focuses on an Android application and leaves other operating systems for future work.

Outline

The remainder of this thesis is structured as follows:

Chapter 2 – Fundamentals This chapter explains the fundamentals used in this thesis. This contains a description of the RMP and explanations of several concepts and protocols used for the implementation.

Chapter 3 – Related Work: There already exist approaches on the topic of device discovery in IoT environments. This chapter discusses these solutions and describes the differences to the work presented in this thesis.

Chapter 4 – Automated discovery and binding of IoT devices: This chapter contains the concepts and specifications for the discovery of devices. It introduces a dedicated discovery and monitoring service extending the RMP and describes an Android client collecting and forwarding sensor data.

Chapter 5 – Implementation: As a proof of concept, this chapter describes the implementation of a prototype for the introduced disocvery service.

Chapter 6 – Conclusion: This chapter concludes the findings of this work and names possible approaches for future work.

[1] http://www.pewinternet.org/fact-sheet/mobile/ (visited on October 27, 2017)
[2] https://android.com

2 Fundamentals

This chapter contains fundamentals, necessary for the understanding of the concepts described in Chapter 4. These fundamentals include a description of the RMP, summaries of the used protocols DNS-Based Service Discovery (DNS-SD) (Section 2.2), Multicast DNS (mDNS), and Bluetooth Service Discovery Protocol (Bluetooth SDP), and, finally, a description of sensor access on Android devices.

2.1 Resource Management Platform

IoT environments usually contain a huge number of devices, like sensors and actuators. To allow an application to access these devices three steps are necessary: registration, binding, and data provisioning. Carried out manually, these tasks may be time-consuming and error-prone. To ease these tasks, Hirmer et al. [HBF+17] introduce the Resource Management Platform (RMP).

The idea behind the RMP, as depicted by the simplified architecture in Figure 2.1, is to bind physical *things* and their associated sensors using device adapters and provision the received data to other applications. To achieve this, the devices can be registered at the RMP using the *device registry*. For this purpose, the device must have an unique identifier which then can be used to get further information about the device and its associated sensors, like frequency or accuracy. This information is stored using ontologies, to enable efficient storage and retrieval. Using the device's identifier, information can be extracted by traversing the ontology. Based on the extracted information, the device registry can select an appropriate *device adapter*, used to gather the device's data. These adapters may be parameterized to specify dynamic information, like the device's MAC-address, pinset, or the url of the RMP. The selected adapter is then deployed either to the device itself, provided that the device has sufficient computing capabilities, or onto an external resource. While the adapter is running, data from the device and its sensors is gathered and sent to the RMP, which takes care of provisioning these data streams to other applications. If the device goes offline, or its data is not needed anymore, the device can be deregistered. This means, the adapter is stopped and all resources allocated for the device are freed. Nonetheless, the device may be reregistered again in the same manner as explained above.

Following the steps described above, the RMP provides a fully automated process from binding devices to provisioning their data. Such a level of automation lowers the costs for experts, managing the devices manually, by far. However, the unique identifier and some other necessary information, like the IP-address of a device, must still be provided manually.

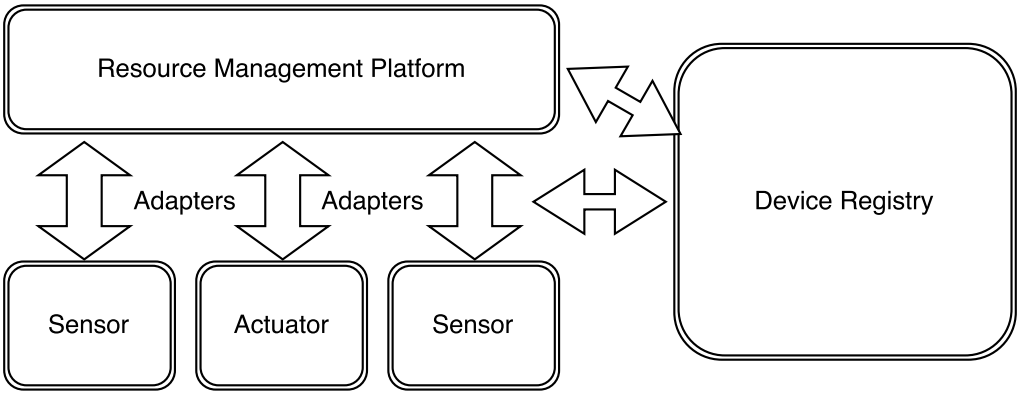

Figure 2.1: Simplified architecture for the RMP

Thus, the target of this work is to provide the RMP with an automated mechanism for discovering and registering devices.

Moreover, the platform currently does not support a dedicated monitoring of registered devices. The state of a device may be derived by the retrieved values. However this is not sufficient for all contexts, as for example actuators can not be monitored this way. Consequently, such a automated monitoring service is also part of this work.

2.2 DNS-SD

In the field of service discovery many protocols and solutions have been published so far, as stated by Edwards [Edw06]. One of the most popular ones is the DNS-Based Service Discovery (DNS-SD) [CK13a], which builds upon the well-known Domain Name System (DNS) [Moc87]. According to Edwards [Edw06], naming and discovery systems have many similarities. Moreover, the DNS is a widely used and implemented standard, thus, providing a good base to be expanded for service discovery purposes.

The DNS provides name resolution using records. Each record holds specific data and has a specific name. These records are stored in a central directory, the DNS server. An application interested in this data may query the central directory for records matching a given name. The most common used record is the A (IPv4) or AAAA (IPv6) record. The name of this record is a specific domain, for example

ipvs.uni-stuttgart.de

and holds the corresponding IP address under which this domain can be reached. However, DNS provides several other record types. The PTR record, for example is used for reverse resolution of the A record. A client, having an IP address, can query the respective PTR record, which contains the host name for the given address. The TXT record is used to store arbitrary – mostly user readable – data. SRV records can be used to announce an available service. For example, the SRV record "_http._tcp.example.com" advertises a running web server for the domain "example.com". There may be multiple instances with the SRV record. For instance the webpage under "example.com" might be served by multiple servers for

load balancing. In this case it is assumed that all instances offer the exact same service. As a consequence, the instances can be used interchangeably.

One of the main goals of DNS-SD is to avoid any fundamental changes in the underlying DNS protocol. Hence, DNS-SD only uses the existing SRV, TXT, and PTR records. Consequently, while allowing effective service discovery, DNS-SD does not require any new records or messages, rather than giving existing parts a new meaning.

As mentioned above, traditional SRV records are used to denote interchangeable instances of a given service. However, there may be services which do not fulfill this assumption. Because of that, DNS-SD extends the service name with a human-readable instance name. For example, a printer, using the Internet Printing Protocol (IPP), may advertise itself as:

IPVS-AS printer._ipp._tcp.ipvs.uni-stuttgart.de

Following this, a client looking for IPP printers can display the list of instances to the user, who selects the desired printer, depending on the instance name.

As the SRV record now contains a fully qualified instance name, one must know the instance name to be able to retrieve the associated address information. Thus, DNS-SD specifies the PTR record to be used for service instance enumeration (browsing). For every SRV record there is also a PTR record. This record has the service name as its name and serves the fully qualified instance name as its data. Consequently, querying PTR records for a specific service in a specific domain, returns a list of service instance names, as described above. With this list, the SRV records for the specific instances can then be queried to retrieve address information and connect to the service.

There are services needing more information than the plain address and port of the host, to allow a successful connection, or to help the client select an appropriate instance. Hence, DNS-SD requires an additional TXT record with the same service instance name as the SRV record. In the TXT record, arbitrary data may be stored using key/value pairs and this data may be retrieved by a client in addition to the SRV record.

As an example, a client searches for all available printers by querying PTR records with:

_ipp._tcp.ipvs.uni-stuttgart.de

resulting in the following list:

AS printer pool._ipp._tcp.ipvs.uni-stuttgart.de
VS printer._ipp._tcp.ipvs.uni-stuttgart.de

The client may then select the desired instance and query the SRV and TXT records with:

AS printer pool._ipp._tcp.ipvs.uni-stuttgart.de

to retrieve address and port, and additional information, such as supported paper formats or duplex printing.

2.3 mDNS

DNS-SD, described in Section 2.2, provides a powerful service discovery mechanism. Nonetheless, it requires a running DNS-server in the network, when used with conventional DNS. As this would impose a great effort on configuring the services and the devices to use this server, a zeroconf approach, called mDNS [CK13b], was introduced by the Internet Engineering Task Force (IETF).

The idea is to spare the central DNS server as it imposes a configuration effort and a single point of failure. Instead, DNS messages are transmitted using IP multicast. Rather than querying the central DNS server, clients may send their queries to the multicast address 224.0.0.251:5353. Every device offering any services can register for this multicast address. For each service the device manages one or more records. If an incoming query matches one of these records, the device generates a DNS response. As records may be cached by other devices for performance reasons, multiple devices have a knowledge of the available records. However, only the device which initially generated the record responds to the query. As to reduce network traffic, devices issuing a query may explicity specify that they desire an unicast response. In this case the response may be returned via unicast, otherwise all responses are submitted over multicast.

Using mDNS it is differentiated between one-shot and continuous queries. One-shot queries aim at resolving a single record name, for example, conventional host name resolution. Continuous queries on the other hand try to monitor the current network state to maintain an updated list, for example, a list of currently active printers. Therefore, the query is sent repeatedly.

As multicast communication is an expensive operation, mDNS incorporates several methods for traffic reduction, like known-answer suppression, duplicate question, and duplicate answer suppression. As their names indicate, all these approaches try to suppress DNS messages, which would not provide new information. For example, a continuous query might contain all services the querying device knows about. Devices receiving this query can suppress their answer, if the query already contains their offered services. Moreover, mDNS defines caching mechanisms, similar to conventional DNS, to further reduce traffic.

Since link-local multicast reaches only devices on the local link, the specification also introduces a new top-level domain ".local.". The usage of this domain enforces a query to be resolved using mDNS and enables administrators to address their own hosts without access to a part of the global DNS name space. As a consequence, domain names and, thus, service names, may not be globally unique. However, name clashes are very unlikely on the local link.

The main purpose of mDNS is to provide a zeroconf DNS solution to use with DNS-SD. This way, devices may offer their services on the local network without – or at least with minimal – configuration effort. However, mDNS may also be used to resolve global host names and, thus, provides a fallback mechanism in case the conventional DNS service is not available.

2.4 Bluetooth SDP

The Bluetooth communication protocol is designed as a Personal Area Network (PAN). As such, it only has a very limited transmission range. Consequently, the number of available devices and with that the number of available services changes frequently. To cope with this rapid fluctuation, the Bluetooth Special Interest Group (SIG) specifies the Bluetooth Service Discovery Protocol (Bluetooth SDP) [Blu14].

Architecture, as depicted in Figure 2.2, and structure of the Bluetooth SDP are similar to DNS-SD described in Section 2.2. Any device offering services to other devices runs a Bluetooth SDP server. Devices can run a Bluetooth SDP client to discover such services. On the client side, several applications may use the Bluetooth SDP client to look for services, as well as there may be multiple applications on the server side, advertising their services with the corresponding Bluetooth SDP server.

The Bluetooth SDP server holds a record for each service registered with it. The record contains identifying, as well as additional feature information. For that purpose, every service implements a service class. Every service class is identified by a globally unique Universally Unique Identifier (UUID) [SLM05]. For common services, such as audio streaming, there are predefined service classes with an assigned UUID. Nonetheless, a service provider may specify a custom service class and assign a new UUID to it. A service may implement more than one classes, which usually means that the implemented classes are subclasses of each other.

A service class lists several service attributes, describing the service. Service attributes may be seen as key/value pairs, consisting of an attribute id and an attribute value. Some service attributes are common to all services, like the *ServiceClassIDList* attribute holding a list of UUIDs, assigned to the service classes implemented by the service. More common attributes are the *ServiceID*, the *ServiceName*, or the *IconURL* attributes.

A Bluetooth SDP client can search for services by connecting to a Bluetooth SDP server and issuing a Bluetooth SDP request. The request contains a list of UUIDs the client is searching for. If the requested UUIDs are a subset of the UUIDs listed inside a record, the Bluetooth SDP response contains a handler to the respective record. Using this handler, the client can retrieve the service attributes for the respective service. The client application can subsequently select the best applicable service, according to the information provided by the service attributes.

In summary, Bluetooth SDP only provides a plain information exchange, similar to DNS-SD. The actual connection to the service is not part of the protocol and must be handled by the client application. In addition, Bluetooth SDP does not provide any notification service in case services become available or inaccessible. Thus, the only way of recognizing a new service or a newly gone service, is by repeated Bluetooth SDP requests.

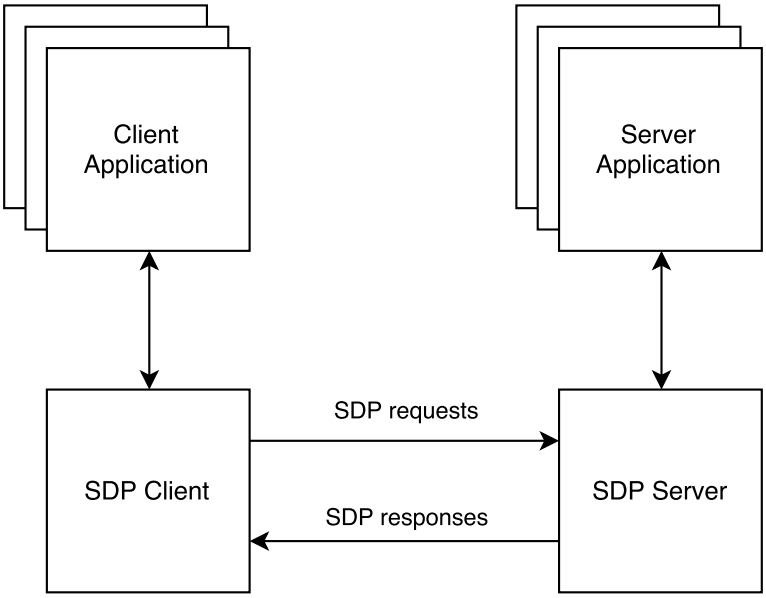

Figure 2.2: Simplified architecture of the Bluetooth SDP (from [Blu14]).

2.5 Android Sensor Access

The Android OS currently does not provide an integrated solution for provisioning data, acquired by internal sensors, to applications not running directly on the phone itself [SWS+10]. Consequently, the only way to access the phone's sensors from outside the device is, to implement an application which forwards the data to external resources. For this purpose, the Android platform offers the sensor framework, allowing applications to access sensors and their data [And17].

As there is a huge variety of phone models running android and since every manufacturer uses different sensors, the number of sensor models is too high to address them directly. To cope with this problem, the sensor framework uses an abstract view on the sensors. It defines a number of types, shown in Table 2.1, which can be used by an application. Each type in a way defines an interface which manufacturers must implement using their specific sensor models. The application may then use the interface to retrieve sensor data.

To access the internal sensors, an application can use the *SensorManager* provided by the sensor framework. The *SensorManager* provides an API to retrieve available sensors and to register listeners with them. The interaction with sensors is event based, as the sensor cannot be queried directly, but only using listeners. Whenever an application registers a listener, the system starts the sensor to generate data. The frequency at which values are generated may vary between different sensor types or models, so the system asynchronously informs the application about a new value using the *SensorEvent* class. Since sensors may acquire data at a higher rate than the application can handle, developers can define a delay, specifying the minimum time frame between two values.

Sensor	Type	Common Uses
TYPE_ACCELEROMETER	Hardware	Motion detection (shake, tilt, etc.).
TYPE_AMBIENT_TEMPERATURE	Hardware	Monitoring air temperatures.
TYPE_GRAVITY	Software or Hardware	Motion detection (shake, tilt, etc.).
TYPE_GYROSCOPE	Hardware	Rotation detection (spin, turn, etc.).
TYPE_LIGHT	Hardware	Controlling screen brightness.
TYPE_LINEAR_ACCELERATION	Software or Hardware	Monitoring acceleration along a single axis.
TYPE_MAGNETIC_FIELD	Hardware	Creating a compass.
TYPE_ORIENTATION	Software	Determining device position.
TYPE_PRESSURE	Hardware	Monitoring air pressure changes.
TYPE_PROXIMITY	Hardware	Phone position during a call.
TYPE_RELATIVE_HUMIDITY	Hardware	Monitoring dewpoint, absolute, and relative humidity.
TYPE_ROTATION_VECTOR	Software or Hardware	Motion detection and rotation detection.
TYPE_TEMPERATURE	Hardware	Monitoring temperatures.

Table 2.1: List of available Android sensors (from [And17]).

Because of the asynchronous behavior, it is necessary to specify a background service, handling the sensor data. The so retrieved data may then be used for calculations or simply be forwarded using networking services. Moreover, the use of sensors has a significant impact on battery power. As the users attach great importance to low battery consumption, it is recommended to keep the usage of sensors at a necessary minimum.

3 Related Work

According to Zaslavsky and Jayaraman [ZJ15], the IoT, and all technologies connected with it, will lead to an *"explosion of connected devices"*. Thus, the discovery of such devices must be automated to cope with the huge number of devices. Already, there have been different approaches to the topic of automated device discovery in IoT networks. They all have in common that they treat the discovery of devices as equivalent to the discovery of services in such environments, as stated by Ccori et al. [CDZS16]. Moreover, they try to reach a *zeroconf* solution, which means that no manual configuration is needed, as this is error prone and not scalable.

According to Ccori et al. [CDZS16], these approaches can be classified into three categories: *centralized*, *decentralized* and *hierarchical*. *Centralized* approaches use one central server, all other devices are connected to. This server handles the detection of available devices. While such an approach has a very poor storage efficiency and has a single point of failure, the advantages are an efficient communication and high success rate. To overcome the disadvantages of *centralized* systems, the discovery of devices may be spread over all participating devices, advertising themselves to each other. Such a *decentralized* approach does not have a single point of failure and does not need a large central storage, but it also has the poorest time efficiency and the lowest success rate. *Hierarchical* systems try to combine these two concepts using the advantages of one to overcome the weaknesses of the other. In these system a specific domain is maintained by a central server, while several of these servers may be joined on another central server in a higher hierarchy level, to form a bigger domain.

Klauck and Kirsche [KK12] introduce a *decentralized* concept building upon DNS-SD. Devices inside the network use Bonjour and IP-Multicast to exchange service records. If a service is needed, a query for such records can be spread through the network using IP multicast. Any devices fitting the description of the requested service can then react accordingly. This approach poses as an efficient method to discover services based on a specific query, but it does not contain any binding systematic. As for this work, it is vital to not only discover devices and their services, but to bind them to an existing platform, the concept of Klauck and Kirsche [KK12] is not fully applicable to the task. Nonetheless, it may be used as a tool in the process of device discovery.

Leguay et al. [LLJC08] address the task of Service Discovery in a similar fashion. In comparison to Klauck and Kirsche [KK12] they base their work on existing web service discovery techniques. To tackle the problem of constrained nodes, they assign each node to one of the categories *full-capacity*, *limited-capacity* and *low-capacity*. Each of these categories has a special protocol stack for service description and discovery assigned to

it. As this approach aims at a fully interconnected network, it is far too complex and not applicable for the tasks of this thesis, since the goal is to only enable the RMP to discover single devices.

Jara et al. [JLF+13] propose with “digcovery” a *centralized* solution. Their aim is to discover devices and services over different communication technologies, like IP, Bluetooth, or RFID. All of these “digrectories” are combined into the “digcovery core”, a central server which serves the data to other clients over a RESTful API. This system has some similarities to the idea of the RMP. However, in some parts it still relies on manual configuration, like scanning QR codes or barcodes. In addition, Jara, Martinez-Julia, and Skarmeta propose the use of a light-weight version of DNS-SD, called *lmDNS-SD* [JMS12], beeing optimized for constrained devices in IoT environments. Although the “digcovery” system has obvious similarities to the RMP, it serves a different purpose. The main focus of “digcovery” is to provide a searchable collection of nearby devices as on the other hand the RMP tries to access the devices data and provision it to other applications. Thus, the binding of devices for the RMP needs a more sophisticated solution than for the “digcovery” system.

Cirani et al. [CDF+14] propose a system, similar to Klauck and Kirsche [KK12], but make use of a so called IoT Gateway. This IoT Gateway is a central server, residing in the same network with many IoT devices. A newly joined device may announce itself to the network using a DNS-SD approach as described by Klauck and Kirsche [KK12]. The IoT Gateway receives this announcement and then queries the offered services using the Constrained Application Protocol (CoAP) [SHB13]. CoAP is similar to HTTP but was optimized for constrained applications and devices. The IoT Gateway may then offer the retrieved services to other clients in or outside the network. The concept of IoT Gateways, collecting information on available services and distributing them to other applications, can be fit into the existing RMP environment. Especially, since this work enables the RMP to support different communication technologies, such gateways are a reasonable extensions. Nonetheless, the system proposed by Cirani et al. does not provide the data but only a handler to the service itself. Thus, the system can only be used as a tool do discover the services, but still must be expanded to allow a sophisticated binding mechanism.

With their platform SENSE-SATION, Shirazi et al. [SWS+10] tackle the problem of integrating the sensors of mobile phones into IoT environments. Current mobile phones contain numerous hardware based or informational sensors, giving the phone an extended sensing capability. Hardware sensors are built in devices, that can measure environmental parameters, like temperature or pressure. Informational sensors are software modules, which can accumulate different hardware sensors to compute own readings, such as the phones position or orientation. As most users carry their phone with them every day, it is desirable to use these capabilities in a different context, from other devices or applications, as well. However, modern mobile operating systems do not allow access to these sensing capabilities from other applications, such as web services. The idea of SENSE-SATION is to create a smartphone client offering the sensor data via a RESTful API and syncing it with a backend service. To cope with the variety of included sensors, the system uses virtual sensors which may be a unified wrapper for different hardware sensors or a combination of multiple sensor values. Since the creation and distribution of these virtual sensors, as well

as the connection between client and server must be done manually, the platform lacks a sophisticated discovery mechanism.

To sum up, most of the listed approaches aim at a *decentralized* system, as their advantages, such as robustness and storage efficiency, fit the dynamic character of IoT environments. However, the RMP already is a *centralized* system, which neutralizes these advantages. Moreover, all approaches rely on some broadcast method to advertise new devices in a network [KK12][CDF+14][JLF+13][LLJC08]. Following that, there might be some bidirectional communication to retrieve more details about the advertised services and to achieve some sort of binding to a central instance [CDF+14][JLF+13]. The work of this thesis adapts this pattern to enhance the RMP with automated device discovery.

4 Automated discovery and binding of IoT devices

The concept, this work introduces, extends the RMP with an automated discovery and monitoring service. The purpose of this service is to monitor an existing environment and discover new devices within it. Discovered devices should be registered at the RMP, to allow automated binding and deployment of adapters as described in Section 2.1. To monitor networks using different communication methods, this concept involves the gateway approach by Cirani et al. [CDF+14]. For each monitored network, the discovery service has a dedicated gateway managing the actual discovery and the following communication with the centralized RMP.

The remainder of this chapter describes the concepts in detail. Section 4.1 discusses different approaches on discovering devices and Section 4.2 describes the selected method. In Section 4.3, the actual protocol used to register and bind devices is explained. Section 4.4 details the monitoring service, used to determine, if a device is still active, and, finally, Section 4.5 lays out, how the new service is integrated into the existing RMP architecture.

4.1 Overview

In IoT environments, every device usually offers one or multiple services. Sensors are offering their values, for example, a temperature sensor providing the current room temperature. Actuators offer to complete certain tasks, for example, a speaker playing a sound. As a result, there is no real difference in discovering devices or discovering their services [CDZS16]. Consequently, established service discovery mechanisms may be used to accomplish the discovery.

In a straightforward approach, each device could use any service discovery solution to advertise itself as a service. Possible protocols for this task would be JINI [ASW+99], SSDP [ALG+99], or the previously described DNS-SD. Most of these protocols are based on a Peer-to-Peer (P2P) strategy, especially when adapted to constrained IoT networks, and do not make use of a centralized directory. This strategy is reasonable since centralized directories, for example, DNS servers, must be manually configured and their addresses must somehow be announced to the devices. As this contradicts the desired *zeroconf* solution, such a method would not be applicable for the RMP.

Using the P2P strategy, a client, searching for a particular sevice, may issue a query into the network, describing the type of service it is looking for. All devices in the network

receiving the query can check the description of the desired service with the description of the services provided by the device. If one of these services matches the query, the device sends a response to the issuing client which usually contains all information for making a successful connection to the service. Note, that the address and port of the service are usually sufficient for a successful network connection. To effectively use a service, there might be more configuration necessary, for example, the name of a printing queue. However, this information can be exchanged using the established communication channel, thus, being out of scope for conventional discovery systems.

The problem resulting from this method is, that a client needs to reissue the query every time it needs a particular service. For many services and networks that remain static for longer time periods, e.g., printers on the local network, this may be avoided using caches. However, IoT environments are highly dynamic, especially when built on wireless communication technologies. As a result, the list of available devices may change rapidly, making such caches superfluous.

As a consequence, any system, such as the RMP, trying to monitor the environment, must issue repeated queries into the network. These queries may impose a significant load to the underlying networks, though, there exist some approaches to reduce this load [CK13b; JMS12; KK12]. The traffic penalty especially increases for the RMP, since it does not only monitor one specific type of service, rather than *all* available services.

Besides the induced traffic, there is another difficulty when working with repeated queries. To reduce the traffic caused by discovery requests, it is desirable to keep the frequency of these requests as low as possible. Especially as high frequencies might lead to many requests not reporting any changes. In this case, the induced load would have been to no purpose. On the other hand, a very low frequency may delay the recognition of new devices or even cause them to enter and exit the environment unnoticed. While this might seem acceptable for many purposes, for example, new thermometers or printers, it might have fatal consequences, for example, in traffic control systems for autonomous cars. An unnoticed car might lead to tragic and fatal accidents.

To cope with the described problems, this concept proposes to turn around the discovery process. Instead of devices advertising themselves and beeing discovered, the RMP advertises a special discovery service. Devices entering the network may use this discovery service to register themselves at the RMP. This approach has two major advantages. First, new devices entering the network are registered immediately. Second, as long as no devices enter the environment, no resources, neither of the RMP, nor, the network, are used. However, as devices are not responding to discovery requests, it is not possible to monitor registered devices using the discovery service. As a consequence, an additional monitoring mechanism must be implemented.

4.2 Discovery Service

For devices entering the environment, the discovery service differentiates between previously known and unknown devices. There are use cases, such as smart homes or presence detection systems, for which this difference is important.

Before the discovery service is described further, it is necessary to define what a device actually is. The RMP is designed in the context of *things*, as described in Section 2.1. These things may have a variable number of sensors and actuators attached to them. An example for such a *thing* is a RaspberryPi[1] with a temperature sensor and a speaker attached to it. This RaspberryPi[1] might be used to monitor the room temperature and to give audio feedback. It can be registered at the RMP, and the device registry consequently extracts the attached sensors and actuators from the respective ontology, using the provided ID. A trivial discovery approach would be to discover the RaspberryPi, retrieve its ID and then register it at the RMP. However, in this case, the sensors and actuators would have to be manually preconfigured in the ontology, since a RaspberryPi may have an arbitrary sensor/actuator setup. Thus, it is not sufficient to discover the single raspberry, but it is necessary to acquire the used sensors and actuators at discovery time. For this purpose, the discovery service, in contrast to the RMP, treats every sensor and every actuator as a separate device. Moreover, each device may be hosted on another device. This allows the previously described RaspberryPi, to be modeled as a device hosting a temperature sensor device and a speaker device.

The discovery service internally identifies devices using a generated GLOBAL_ID. This GLOBAL_ID is specified as an integer value greater than zero. Each device is assigned a GLOBAL_ID upon registration. The discovery service introduces this ID, although the RMP assumes hardware addresses, like the MAC address, to already be globally unique. This is justified with the discovery service being designed to support different communication technologies, some of which might not use unique addresses.

For registration at the RMP, the discovery service has three functions:

- register a device — this function registers an unknown device.
- reregister a device — this function reregisters a previously known device.
- read adapter configuration — this function reads information, necessary for adapter configuration

The function to register an unknown device requires information, identifying the device and specifiying how to communicate with it. This information includes:

- the device's type
- the LOCAL_ID

[1] https://www.raspberrypi.org/

- the GLOBAL_ID of the device's host
- the hardware address – MAC or Bluetooth address
- the network layer address – usually the IP address

The type is identified by a string representing its name and describes how the device should be accessed. Usually every sensor or actuator model has its own type and every type can be mapped to exactly one device adapter. For example, if manufacturer *S* builds a pressure sensor with model name *P250*, then there most likely will be a type *S-P250* in the RMP. For this type, an adapter may be developed and every device, registering with it, will be accessed using this adapter implementation. Note, that the naming of types and models is only for example purposes and does not impose any rule for names of types or models.

As described above, one device may act as a host for others. Since a device may host several devices of the same type, they must be differentiated using a LOCAL_ID. The LOCAL_ID is defined to be a string, being unique for all devices of a specific host. Moreover, it should be human readable, as it may be shown to the user as the device's name.

The host GLOBAL_ID can be used by the discovery service as a reference to determine the host of a device. If an illegal value – zero or less than zero – or no value is provided at all, the discovery service assumes that there is no host and, thus, the device is to be treated as a *thing* in the context of the RMP. Consequently, the host must be registered first, to allow a successful registration of the device.

The hardware and network layer addresses define how the device needs to be addressed. Though for most use cases this information will be inferable from the networking protocols, it should be explicitly declared for completeness. Moreover, the hardware address may be used as another globally unique identifier to identify possible duplicates. As there are communication technologies, that do not require a network layer, for example, Bluetooth, the network layer address is optional and may be omitted.

The discovery service stores the received data and assigns a newly generated GLOBAL_ID to the device. The service may also validate the received information, for example, to check for duplicates or for security purposes. In case of a successful connection, the service answers the device with the new GLOBAL_ID, otherwise the device receives an error.

For the reregistration of a previously known device, it basically is sufficient to provide the GLOBAL_ID associated with the device. Nonetheless, it is advised to include the above information in the message for validation purposes. The discovery service checks if there is a device assigned to the given GLOBAL_ID and, if so, registers the device again with the RMP.

The third function of the discovery service is to collect the adapter configuration for a device. Since device adapters may be parameterized, as described in Section 2.1, these parameters must be provided by the device. Currently, this is a simple key/value map, which is sent to the service by the device, upon successful registration. The necessary keys are defined by the type of the device and the values must, possibly manually, be configured for every device.

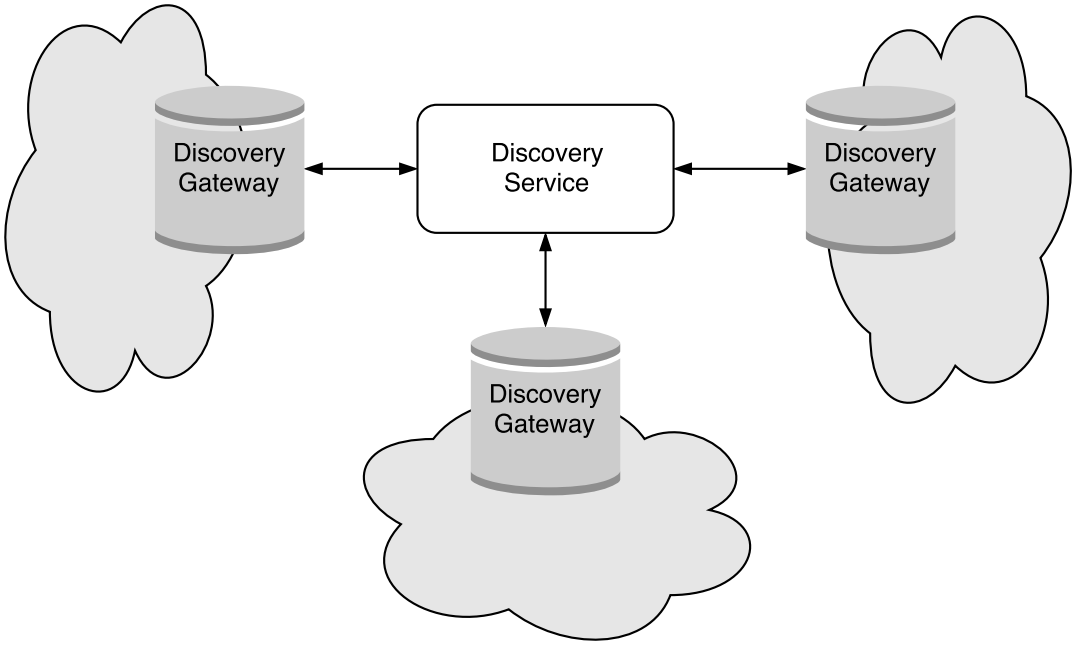

Figure 4.1: Architecture of the discovery service

The discovery service runs together with the RMP, usually on the same machine. As the service should be able to discover devices using arbitrary communication technologies, this concept uses special discovery gateways. These gateways are logically separated from the discovery service and, thus, may run on a different physical machine. For every communication technology and network, that should be monitored, there is at least one gateway, as depicted in Figure 4.1. Those gateways keep a static connection to the discovery service at all times and advertise themselves to devices inside their network. The idea is, that gateways communicate with the devices on a local network, while their connection to the service may use a different communication technology, like the global internet. Devices looking for the discovery service can find a gateway for their network and connect to it. The gateway then uses its own connection to call the functions of the discovery service.

Currently, the discovery service supports IP-based networks and Bluetooth networks. The concept of 6LoWPAN [Mul07] may provide a simple way to extend this list, as it provides an IP network layer for different low-power wireless networks. However, native solutions should be preferred. The following sections describe how the discovery gateways are advertised in the different environments.

4.2.1 IP-based networks

In principle, any service discovery protocol may be used for IP-based networks, as discussed in Section 4.1. The concept presented in this work uses DNS-SD in connection with mDNS, because of the wide spread use of DNS. Moreover, most devices already are DNS enabled,

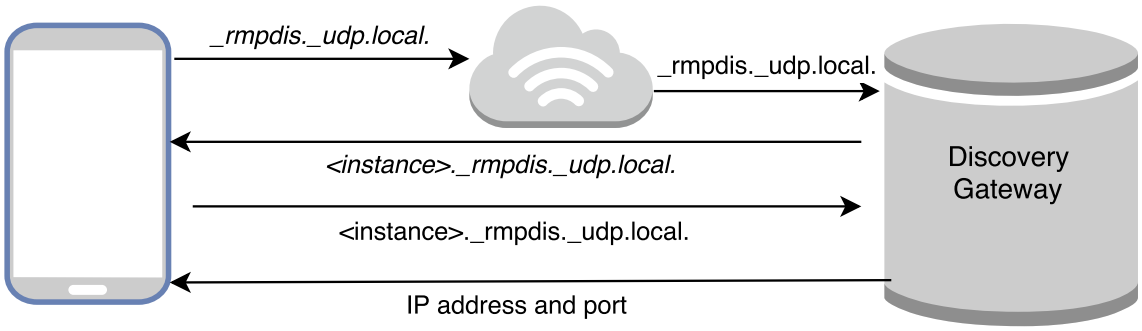

Figure 4.2: The discovery process based on DNS-SD

so using DNS-SD is most likely to be supported natively by many devices. However, it is possible to use other protocols instead or to operate several protocols at the same time.

For the advertisement of the discovery service, a new service name is defined:

_rmpdis.

Since the discovery service offers its function using UDP, as explained in Section 4.3, the complete service name, base on the rules defined by DNS-SD, is:

_rmpdis._udp.

Every discovery gateway is an instance of this service and may be configured to use an arbitrary instance name, to advertise itself in the ".local." domain. In addition, the gateway may also be advertised using a specific domain, like "rmp.uni-stuttgart.de", if there is a use case affording it.

Figure 4.2 depicts how a device discovers the discovery gateway. First, the device issues a multicast query for PTR records, matching the name:

_rmpdis._udp.local.

The gateway checks this query and, as the service name matches the discovery service, it replies with its service instance name:

<instance>._rmpdis._udp.local.

This reply and all following traffic between the device and the gateway should use unicast to minimize network load. As the device receives the gateway's reply, it queries the SRV record with the provided service instance name. The gateway replies with address and port of the discovery service. The device has now discovered the service successfully and may connect to it using the protocol defined in Section 4.3.

4.2.2 Bluetooth networks

On Bluetooth networks, the discovery gateway is advertised using Bluetooth SDP. For this purpose, a new service class is defined. For example, the randomly chosen UUID "809a061f-d834-4ba6-8741-29f0f53ca1b9" is assigned to this class. For the discovery process to finish successfully, it is sufficient to only define the *ServiceClassID* attribute, as this uniquely identifies the discovery service. However, as there might be several discovery gateways to increase radio range, it is possible to define more attributes to differentiate

gateways. Moreover, the *ServiceName* and *ServiceDescription* attributes may be populated for use within the user interface.

To advertise itself, the discovery gateway starts a Bluetooth SDP server or registers with an existing one. Devices looking for the discovery service, start a Bluetooth SDP client and query surrounding Bluetooth SDP servers for their services matching the specified UUID "809a061f-d834-4ba6-8741-29f0f53ca1b9". If such a service is found, the device uses the received handler to query the Bluetooth SDP server for host and port of the service. After having discovered the service successfully, the device may then connect to it using the protocol defined in Section 4.3.

4.3 Protocol

When a device has discovered the discovery gateway, it can connect to it and register itself, using the following protocol. Device and gateway exchange UDP messages containing JSON [Bra14] objects. UDP is selected as the transport protocol because it minimizes the network overhead. Acknowledgments are used to cope with the possible package loss and flow control is not needed, since the messages sent should always fit into a single UDP package. JSON is considered to be the most suitable format to represent structured data on constrained devices [JLF+13].

Each message has a type, which may be retrieved from the JSON object stored in it. The protocol defines four message types:

- HELLO – initial message
- CONF – carries configuration info
- PING – carries one data item
- VALUE – may carry sensor/actuator data

The HELLO message is a flat key/value pair list. It is the first message sent by the device and contains all data necessary to register the device at the RMP, as explained in Section 4.2. If the device has been registered before, the GLOBAL_ID assigned to it may be included in the message to trigger reregistration. Otherwise, the device will be treated as unknown by the discovery service. Furthermore, the HELLO message is designed to be broadcasted into the network, thus providing a fallback if no discovery protocol is available. If the discovery gateway receives a broadcasted HELLO message, it can extract all information necessary to continue the protocol with unicast messages. However, as broadcasts are not always available or often not permitted, this method is discouraged. As an answer to the HELLO message, the device expects a PING message containing the result of the registration.

CONF messages contain information necessary to configure the device adapters. The device sends this information upon a successful registration. Device and service may exchange several CONF messages, yet currently, such a negotiation of parameters is not needed in any use case. Thus, the negotiation of parameters is left for future work.

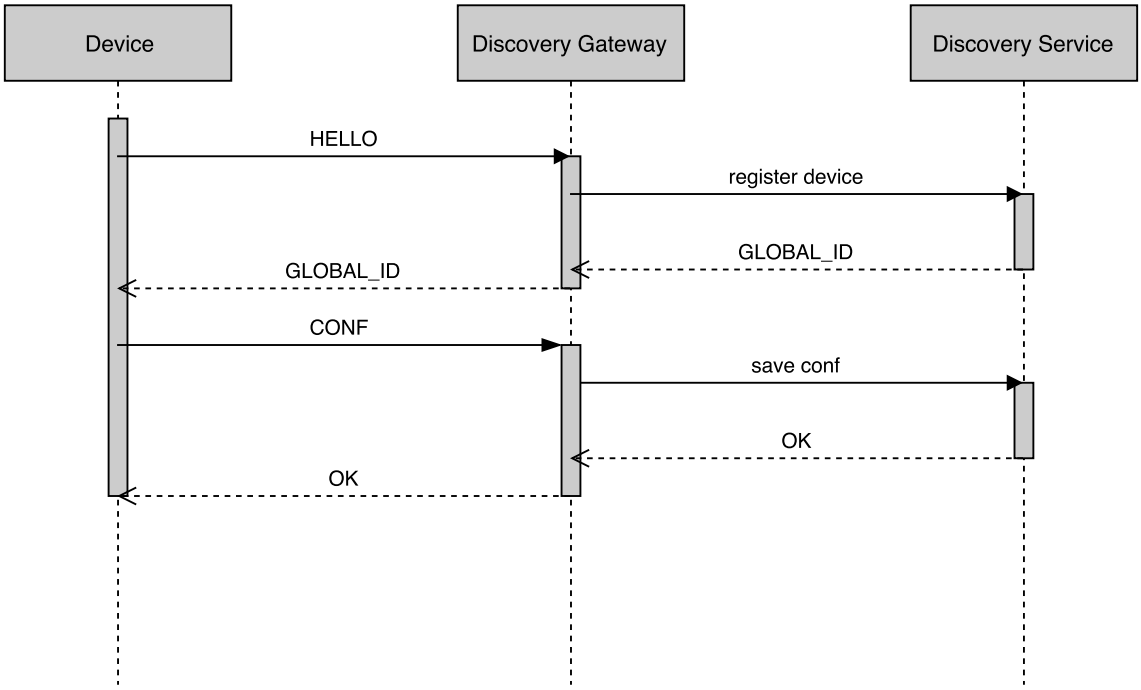

Figure 4.3: Illustration of the protocol for a successful registration

The PING message carries only one item of information. It is mostly used as an implicit or explicit acknowledgment or heartbeat message for monitoring purposes.

The VALUE message type, may be used by adapters deployed on the device to send sensor data. Since the provisioning of these data is out of the scope of this thesis, its precise definition and usage is left for future work.

In Figure 4.3, a complete run for successful registration and binding is illustrated. Sample messages for the protocol run are listed in Listing 4.1.

The device has discovered the discovery gateway and sends the first HELLO message. The gateway receives the message and extracts the necessary data. Note that the gateway does not simply forward the messages to the discovery service, but is able to use and alter the received data. This allows the gateway to perform arbitrary computations with the data, before and after it calls the discovery service. The gateway uses the received information to call the discovery service's function(s) to register the device. The communication between gateway and discovery service is left as an implementation detail, since they may run on different threads in the same process or be deployed on physically separated machines. The discovery service performs validation checks, i.e., duplicate checks or type support, and, if accepted, generates a GLOBAL_ID. This ID is assigned and sent back to the device. At this point, the RMP knows about the presence of the device, thus, the device is treated as *registered*. In case the registration fails or the device is rejected, the discovery service responds with an error message, to inform the device of the failure. Hence, the response is used as an implicit acknowledgment.

Listing 4.1 Sample protocol messages

```
hello_msg = {
  CONN_TYPE: CONN_HELLO,
  TYPE: sensorX,
  LOCAL_ID: mySensor,
  HOST: 100,
  HW_ADDRESS: 12:34:56:78:9A:BC,
  IP: 192.168.0.105
}

hello_response = {
  CONN_TYPE: CONN_PING,
  MESSAGE: 101
}
```

```
conf_msg = {
  CONN_TYPE: CONN_CONF,
  pinlayout: <layout>,
  timeout: 6s
}

conf_response = {
  CONN_TYPE: CONN_PING,
  MESSAGE: ok
}
```

After the device is *registered*, it sends the adapter parameters to the gateway. The necessary data may be automatically computed by the device, like the sensor frequency, or must be manually preconfigured, such as the set of used hardware pins. The gateway again validates the data and then sends it to the discovery service. If the service has gathered all data necessary for adapter deployment, it responds with an acknowledgment message. As now all information, required to use the device, is available for the RMP, the device is treated as *bound*.

In case one of the acknowledgments is lost, the HELLO or CONF message is resent. Thus, the gateway/service must provide a mechanism to recover the answer, either through caching or recomputation.

The given protocol might be extended to include active adapter deployment, but since the deployment is not part of this thesis, it is not further discussed here.

4.4 Monitoring

An important part of monitoring IoT devices is to recognize devices becoming unavailable. There are different reasons for this to happen. For example, the device could stop operating due to a technical error or the lack of power supply. Moreover, changes in the network topology might lead to devices becoming unavailable. Currently, the RMP does not have such a monitoring functionality. Although it is possible to explicitly deregister devices, the RMP cannot recognize missing devices on its own. To cope with that, this concept extends the discovery service from Section 4.2 with an additional monitoring service.

A lightweight approach without further network traffic would be to use the sensor data for an implicit monitoring. As long as sensor data can be acquired, it can be assumed that the device is alive. However, there are common scenarios for which this monitoring technique is not applicable. A trivial example are actuators as they do not produce any sensor data, that could be monitored. Moreover, there are sensors, which do not generate values in a high enough frequency for effective monitoring purposes. For example, a smoke detector,

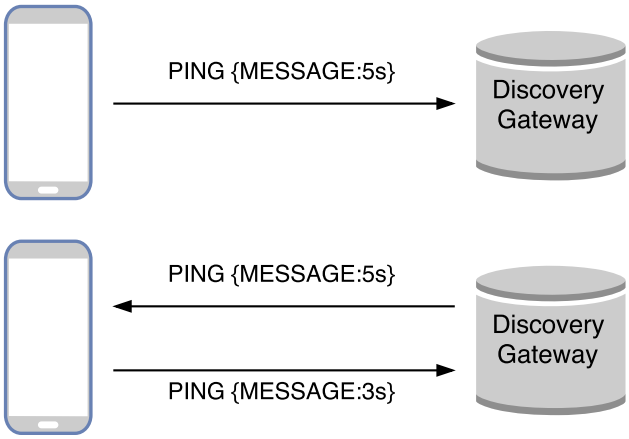

Figure 4.4: The possible monitoring solutions

used for fire detection, ideally never sends any data. Nonetheless, it is important to know if the device is still alive and reachable. Another point is that the data provisioning may be organized using external resources. For example, there are approaches, using a MQTT[2] broker [FBK+16]. These messages are completely separated from the RMP and, thus, do not allow an implicit monitoring.

As a consequence, additional messages must be used. These so called "heartbeat" messages lead to the same problem as repeated queries, discussed in Section 4.1. The higher the frequency, the higher the (possibly useless) traffic. The lower the frequency, the higher the possibility of missing important events. Furthermore, there are devices for which a lower heartbeat frequency is applicable, like the mentioned smoke detector. In this case, a daily heartbeat might be sufficient. On the other hand, there are devices which require a high heartbeat frequency, for example, a motion sensor monitoring the movements of an autonomous robot. Here, an undetected sensor failure might cause high damage to humans or goods. Thus, this concept does not specify a static heartbeat frequency, rather than a dynamic timeout interval. This interval is either preconfigured for the device, negotiated during the binding process or a combination of both.

Having defined a timeout interval, the devices can be monitored using two different methods, as depicted in Figure 4.4. The simple approach is to let the device repeatedly send heartbeats – PING messages, as defined in Section 4.3 – to the discovery gateway, according to the defined interval. The gateway is responsible to signal the monitoring service that the device is alive. This allows an effective monitoring by the RMP. The monitoring service repeatedly iterates the list of registered devices and checks whether the period since the last contact with the device is shorter than the defined timeout interval. However, while the RMP can detect unavailable devices this way, the device itself is not able to determine whether it can reach the RMP or not. This might not be necessary for many use cases, since the RMP will detect the unreachable device and alert a technician to fix the problem.

[2] http://mqtt.org/

In spite of that, there are scenarios, in which it would be useful for the device to know whether it is connected or not. For example, the address of the discovery gateway may change due to an administrative network change. In this case, the device would keep sending heartbeats to the old address, but they do not reach the gateway. Thus, the device will be marked as unavailable by the monitoring service, although device and network are operating correctly. A technician would need to restart the device to trigger a fresh run of the discovery protocol. The device will then find the gateway at the new address and could be reregistered.

The second approach involves a two-way heartbeat to trigger the restart of the discovery protocol automatically. In this approach, the monitoring service triggers the gateway to send the first heartbeat to the device before it times out. The device then responds with its own heartbeat to the gateway, which again marks the device as alive. If the device does not respond to the heartbeat, it can be considered unavailable. Similarly, the device supposes the RMP to be unreachable, if no heartbeat arrives in time. This approach allows an effective monitoring on both sides. Moreover, the two-way approach allows the timeout interval to be adjusted with every heartbeat. The timeout interval may be carried by the PING messages used for the heartbeats. The monitoring service may suggest a different timeout, if it seems applicable, while the device may accept this suggestion or use an arbitrary timeout. It is important that, in the end, the interval specified by the device is used, as there may be devices which use a static timeout and are not able to adjust that due to technical reasons.

4.5 Integration into the RMP

The RMP is a web application with a database backend. Other applications may access the RMP using a RESTful API. This leads to two possible solutions for the integration of the discovery service into the RMP. One approach is to include the service into the current implementation and allow access to the service's functions through the RESTful API. This would allow the discovery gateways, which must be implemented as a possibly standalone application, to use the service with HTTP requests. This is the most uniform solution, however, it is possible that future extensions of the protocol might require a more complex communication between gateway and service. Therefore, it is not recommended forcing the use of the RESTful API. The second possibility is to implement the service as another standalone application, running alongside the RMP. Interaction with the RMP could be achieved by either using the same database or through the REST interface. The former is not recommended, as in good software architectures each application should use its database exclusively.

5 Implementation

A prototypical implementation of the concepts described in Chapter 4 is part of this thesis. The prototype implements the basic functions of the discovery service and includes an Android app, which can be used to register smartphones and their sensors at the RMP.

The current implementation of the RMP is a web application based on Java EE[1] and the Spring[2] framework. For persistence, it uses a MongoDB[3] instance, which is a document based NoSQL database. The web frontend, shown in Figure 5.1, provides the user with lists of available types, currently registered devices, sensors, and actuators. Device in this context equates to the concept of *thing*, described in Section 2.1. Devices are associated with a specific type, while sensors and actuators are associated with a type and a device.

Using the RESTful API, applications may query the RMP for the available devices and their status. Moreover, adapter deployment for *registered* devices may be triggered via the REST interface. The remainder of this chapter describes the prototypical discvovery service in Section 5.1, and the Android client in Section 5.2.

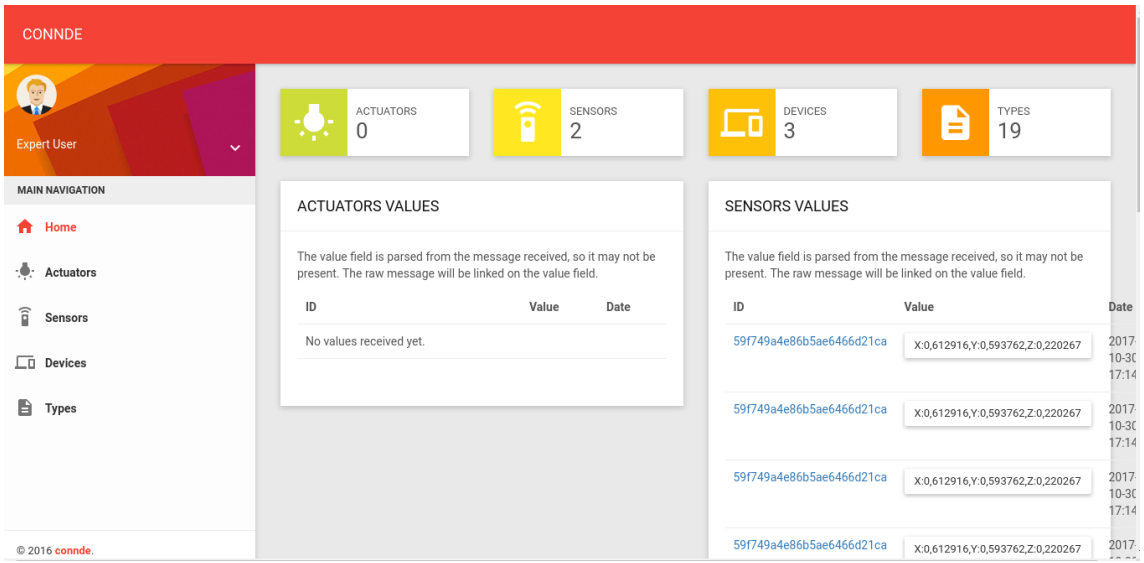

Figure 5.1: Screenshot of the RMP's web interface

[1] http://www.oracle.com/technetwork/java/javaee/overview/index.html
[2] http://spring.io/
[3] https://www.mongodb.com/

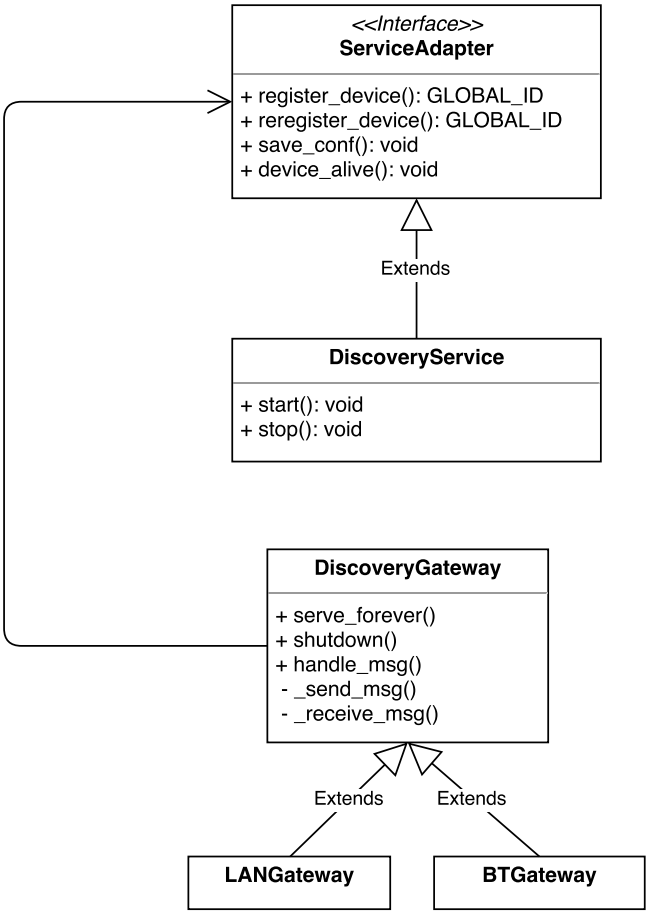

Figure 5.2: Structure of the *discovery* package

5.1 Discovery Service

The prototypical discovery service is implemented using Python[4], which is a flexible language frequently used for prototyping. The implementation provides the two python packages: *discovery* and *advertise*. The packages can be treated as separate libraries and provide a public API to be included by other applications.

The discovery package is structured as depicted in Figure 5.2 and includes an implementation of the discovery service, as well as of the discovery gateway. The service is designed to run as a standalone application, independent of the RMP, and manages its own MongoDB[3] database. A list of all discovered devices and their associated information is stored in the service's own database. This is necessary, as the discovery service must be able to recognize already known devices. For communication with the RMP, the discovery service accesses the RMP's database directly.

[4] https://www.python.org/

Listing 5.1 An example configuration file

```
{
   "self": { "local_id": "raspberry",
      "type": "raspberry3",
      "adapter_conf": {"timeout": 30}},
   "devices": [
      { "local_id": "button",
        "type": "Button",
        "adapter_conf": {
           "pinset": "13,14",
           "timeout": 10}},
      { "local_id": "speaker",
        "type": "Speaker",
        "adapter_conf": {
           "pinset": "15,16",
           "timeout": 15}}
   ]
}
```

The discovery service is implemented within a single python class. It offers methods to be started or stopped and implements the interface of a *ServiceAdapter*. Every discovery gateway uses a *ServiceAdapter* to access the discovery service's functions, described in Section 4.2. Different adapters may be implemented for several communication mechanisms between service and client, for example, RPC [Nel81]. The prototypical gateways can use the service class directly, as they are running in the same process.

The *DiscoveryGateway* class gives an implementation of the protocol, specified in Section 4.3. For this purpose, it offers the `handle_msg()` method, which treats incoming messages accordingly. In addition, the class provides the abstract methods *_send_msg* and *_receive_msg*. To support different communication technologies, subclasses of the *DiscoveryGateway*, which implement these methods accordingly, may be created. Currently, the prototype includes implementations of a *LANGateway*, for IP-based networks, and a *BTGateway*, for Bluetooth networks. The specialized implementations also include the advertisement mechanisms. At the moment, for IP-based networks the prototype uses DNS-SD in combination with mDNS, while for Bluetooth networks a Bluetooth SDP solution is implemented. For this, the prototype uses the libraries *zeroconf*[5] and *pybluez*[6] respectively.

The *advertise* package contains the implementation of the protocol for client devices. It is structured in a similar way as the discovery package. It contains an *AdvertiseService*, which can be used by any application running on the client device. The *AdvertiseService* starts an abstract *Advertiser*. This *Advertiser* implements the protocol and provides abstract methods for discovering the server, as well as for sending and receiving messages. The prototype offers specialized implementations of the *Advertiser* class for IP-based networks and Bluetooth networks.

[5] https://pypi.python.org/pypi/zeroconf
[6] https://pypi.python.org/pypi/PyBluez/

Listing 5.2 Advertisement code for IP-based networks

```
def discover_server(self):
  log.info('Discovering server')

  log.info("Starting DNS-SD discovery")
  dnssd = zerocnf.Zeroconf()
  service_browser = zerocnf.ServiceBrowser(dnssd, '_rmpdis._udp' + '.local.',
                                   handlers=[self.server_discovered])
  while self.discovered_server is None:
    <wait at most five seconds>

  if self.discovered_server is not None:
    return self.discovered_server

  <fallback broadcast> # in case no gateway could be found

def server_discovered(self, zeroconf, service_type, name, state_change):
  if state_change is zerocnf.ServiceStateChange.Added:
    info = zeroconf.get_service_info(service_type, name)
    log.info("Discovered server |%s| at |%s:%s| using DNS-SD", name, str(service_addr),
        str(service_port))
    ip, mac = self._read_own_address(info.address, info.port)
    self.discovered_server = ((info.address, info.port), ip, mac)
```

The *AdvertiseService* currently reads a configuration file from the device, which contains all information about the device and its attached sensors and actuators. However, the contents of the file might be generated upon application start, depending on the use case. Listing 5.1 shows an example of such a configuration file. The device, or *thing*, is a RaspberryPi with a button and a speaker attached to it, using hardware pins 13,14 and 15,16 respectively.

Having all necessary information, the *Advertiser* tries to discover a discovery gateway, as shown in Listing 5.2 (Phrases inside <> describe condensed code). The `discover_server()` method starts a service browser, searching for services of the type *_rmpdis._udp*. When a gateway is discovered, the `server_discovered()` method is called, which reads the devices own MAC and IP address and stores the result.

After successful discovery, the *Advertiser* tries to connect all devices, starting with the RaspberryPi as the host device. For each device the `connect_device()` method, shown in Listing 5.3, creates a HELLO message with the LOCAL_ID, type, IP address, and MAC address of the device. In addition, it adds the hosts GLOBAL_ID if necessary. In case the device has been registered before, the last known GLOBAL_ID is added to the message.

When the discovery service receives an registration request, it performs duplicate checks upon hardware address, type, and LOCAL_ID of the device. Provided the registration was successful, the discovery service inserts the device into the RMP's database and internally adds it to the actively monitored devices. The devices receives a reply indicating the registration result.

The device checks the received answer for a valid GLOBAL_ID. If the devices was registered successfully, a CONF message is created containing the received GLOBAL_ID and the

Listing 5.3 Code to connect a device

```
def connect_device(self, device, ip, hw_addr, global_id):
  # send hello message
  hello_msg = {
    DEV_IP: ip, DEV_HW_ADDRESS: hw_addr.lower(), DEV_TYPE: device[DEV_TYPE],
    LOCAL_ID: device[LOCAL_ID], CONN_TYPE: CONN_HELLO
  }

  if HOST in device: # check if the device has a host
    hello_msg[HOST] = device[HOST]

  if global_id: # check if the device has been registered before
    hello_msg[GLOBAL_ID] = global_id

  self._send_msg(hello_msg)

  hello_reply = self._receive_msg()
  if hello_reply and <contains valid GLOBAL_ID>: # check for valid server response
     global_id = hello_reply[GLOBAL_ID]

     # send conf message
     conf_msg = {
       GLOBAL_ID: global_id,
       CONN_TYPE: CONN_CONF
     }

     adapter_conf = device[ADAPTER_CONF]
     for key in adapter_conf: # copy the entries from the config file to the message
       conf_msg[key] = adapter_conf[key]

     self._send_msg(conf_msg)
     log.debug('Waiting for ACK')
     ack = self._receive_msg()
     <repeat until acknowledgment arrives, otherwise treat the device as unconnected>

     return global_id
```

adapter configuration copied from the local configuration file. This information must at least contain the timeout interval after which the device is considered unavailable.

The discovery service uses the information in the CONF message to configure the adapter. If there is sufficient data, the service sends an acknowledgement to the device. Moreover, it can trigger the adapter deployment, using the REST interface of the RMP.

At this point, the prototype only supports the simple monitoring approach, described in Section 4.4. Thus, the client device sends repeated heartbeat messages after a successful registration. If a device times out, the discovery service deletes the device from the RMP's database, while keeping it stored internally, in case it tries to reregister at a later point.

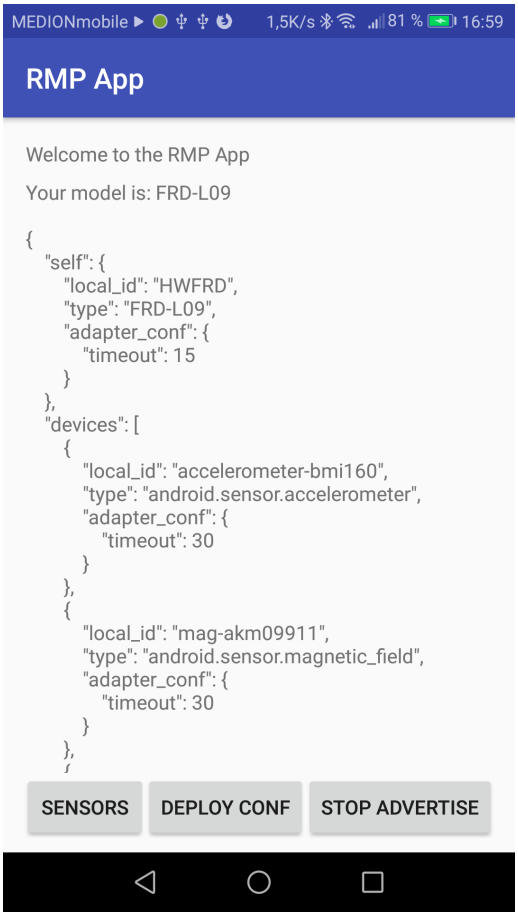

Figure 5.3: User interface of the RMPApp

5.2 Android client

The Android system does not allow applications to access internal sensors from outside the phone. Thus, a prototypical implementation of an Android application is part of this thesis. Since Android applications are typically written in Java, the advertise library was ported from Python. Currently, the Java variant of the library is limited to IP-based networks.

The application includes an activity providing a basic user interface, shown in Figure 5.3. It can be used to list the available sensors and to start or stop advertising of the phone. Upon first start, the application reads the list of available sensors and compiles the configuration file required by the advertise library. The smartphone acts as the host, with its type and LOCAL_ID defined by the Android constants `Build.MODEL` and `Build.DEVICE`. As types for the sensors, the application uses the Android sensor types listed in Table 2.1 on page 23. The LOCAL_ID of the sensor is determined by the sensor's name, assigned by the manufacturer. Moreover, the application offers an background service to advertise the device on the WiFi connection. The service, when active, listens for changes of the phone's connection state and starts advertising when a new connection has been established. If a gateway has been

discovered, the service registers the phone and its available sensors at the discovery service and sends repeated PING messages for monitoring purposes.

The registration at the RMP is performed in the same way, as described in Section 5.1. However, Android prohibits the execution of arbitrary code, since this would pose as a severe security threat. Consequently, device adapters cannot be deployed onto the phone by the RMP. Thus, the prototype application includes its own adapter implementation in the form of a MQTT service. The service assumes a message broker running on the same address as the discovered discovery gateway. It registers with the phone's sensors and sends resulting data to the broker. Information, necessary to post the data successfully to the RMP, is provided by the discovery gateway during the adapter configuration exchange.

6 Conclusion

The emerging IoT leads to new challenges in registering and binding billions of devices. Automated mechanisms must be developed, since for efficiency reasons, these tasks cannot be performed manually. The RMP is one approach that allows an automated binding of devices and automated provisioning of their data. However, the RMP requires manual registration of the devices.

This thesis describes an extension for the RMP enabling it to discover devices automatically. For this purpose, existing discovery mechanisms are examined, whether they can be used for the task. As there is no solution fulfilling all necessary requirements, this work introduces a discovery service extending the RMP.

The idea of this service is to advertise the discovery service and let devices search for it, instead of the discovery service searching for devices. For this purpose, the discovery service may use any discovery protocol, however, it must be able to monitor different networks with different underlying technologies. Thus, this work uses discovery gateways, which act as a bridge between the discovery service and the devices in a network. The discovery gateway maintains a static connection to the possibly remote discovery service and communicates with the devices locally. In doing so, it does not simply forward messages. It rather processes the contents and calls the according functions of the discovery service.

This work is limited to IP-based and Bluetooth networks, using DNS-SD combined with mDNS or Bluetooth SDP, respectively. Running DNS-SD, client devices may search for the discovery service, using the defined service name *_rmpdis._udp*, while using Bluetooth SDP devices may search for the UUID *809a061f-d834-4ba6-8741-29f0f53ca1b9*. However, the aim of future work will be to add support for more communication technologies.

In addition, this work specifies the protocol which may be used by devices in order to connect to the RMP. Devices discover one of the available gateways for their network. Following the protocol, they can connect to the RMP in two steps. First, they announce themselves with an initial HELLO message. The discovery service allocates resources for the device and treats it as *registered*, which means the device is known. In the second step, device and discovery service negotiate the adapter configuration. Having acquired all necessary information, the device is treated as *bound*. The scope of this thesis is limited to these two steps. However, for future work, the protocol might be extended to include the deployment of adapters or the provisioning of device data.

Moreover, the discovery service has the ability to monitor registered devices. Monitoring may be done in two different ways. Either with single heartbeats repeatedly sent by the device or with a two-way heartbeat initiated by the monitoring service. The heartbeat

messages may contain new timeout intervals to allow a dynamic adjustment of the heartbeat frequency. Timed out devices are deregistered at the RMP, yet the discovery service remembers them for future connection requests.

As a proof of concept, the work of this thesis includes a prototypical implementation of the described discovery service and protocol. The prototype is implemented in Python and can be run as a separate process in addition to the RMP. The service accesses the RMP over the provided REST interface. The discovery gateways of the prototype are implemented to run in the same process, yet the API is designed to allow remote gateways.

To allow access to smartphone sensors, the prototype includes an Android app. The app includes a Java implementation of the protocol library and advertises the phone into the connected WiFi network. In order to send values to the RMP, the app protoype includes an adapter which reads the phone's sensors and provisions the data using the MQTT protocol.

To sum up, the described discovery service poses as an effective and stable solution for discovery and binding of IoT devices. It extends the existing RMP, which allows a fully automated solution for discovering and binding of devices, as well as the provisioning of their data. This prevents error-prone and tedious manual configuration, which makes the RMP IoT ready.

Bibliography

[ALG+99] S. Albright, P. J. Leach, Y. Gu, Y. Y. Goland, T. Cai. *Simple Service Discovery Protocol/1.0*. Internet-Draft draft-cai-ssdp-v1-03. Internet Engineering Task Force, Nov. 1999. URL: https://datatracker.ietf.org/doc/html/draft-cai-ssdp-v1-03 (cit. on p. 29).

[And17] Android Develpoers. *Sensors Overview*. 2017. URL: https://developer.android.com/guide/topics/sensors/sensors_overview.html (cit. on pp. 22, 23).

[ASW+99] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, A. Wollrath. *Jini specification*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201616343 (cit. on p. 29).

[Blu14] Bluetooth Special Interest Group. *Specification of the Bluetooth System Covered Core Package Version 4.2*. April. 2014, p. 2272. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification (cit. on pp. 16, 21, 22).

[Bra14] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Tech. rep. 7159. 2014. DOI: 10.17487/RFC7159. URL: https://rfc-editor.org/rfc/rfc7159.txt (cit. on p. 35).

[CDF+14] S. Cirani, L. Davoli, G. Ferrari, R. Léone, P. Medagliani, M. Picone, L. Veltri. "A Scalable and Self-Configuring Architecture for Service Discovery in the Internet of Things." In: *IEEE Internet of Things Journal* 1.5 (2014), pp. 508–521. DOI: 10.1109/JIOT.2014.2358296 (cit. on pp. 26, 27, 29).

[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)* 1 (2016), pp. 97–98. DOI: 10.1109/ISCE.2016.7797388. URL: http://ieeexplore.ieee.org/document/7797388/ (cit. on pp. 25, 29).

[CK13a] S. Cheshire, M. Krochmal. *DNS-Bsed Service Discovery*. Tech. rep. IETF, 2013, pp. 1–49. URL: https://tools.ietf.org/html/rfc6763 (cit. on p. 18).

[CK13b] S. Cheshire, M. Krochmal. *Multicast DNS*. Tech. rep. IETF, 2013, pp. 1–70. URL: https://tools.ietf.org/html/rfc6762 (cit. on pp. 20, 30).

[Edw06] W. K. Edwards. "Discovery systems in ubiquitous computing." In: *IEEE Pervasive Computing* 5.2 (2006), pp. 70–77. ISSN: 15361268. DOI: 10.1109/MPRV.2006.28 (cit. on p. 18).

[FBK+16] A. C. Franco da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. "OpenTOSCA for IoT: Automating the Deployment of IoT Applications based on the Mosquitto Message Broker." In: *Proceedings of the 6th International Conference on the Internet of Things (IoT)*. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany. Stuttgart: ACM, Nov. 2016, pp. 181–182. ISBN: 978-1-4503-4814-0/16/11. DOI: 10.1145/2991561.2998464. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-39 (cit. on p. 38).

[Gar16] Gartner. *Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016*. 2016. URL: https://www.gartner.com/newsroom/id/3415117 (cit. on p. 16).

[GBMP13] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions." In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. ISSN: 0167739X. DOI: 10.1016/j.future.2013.01.010. arXiv: 1207.0203. URL: http://dx.doi.org/10.1016/j.future.2013.01.010 (cit. on p. 15).

[HBF+17] P. Hirmer, U. Breitenbücher, A. C. Franco da Silva, K. Képes, B. Mitschang, M. Wieland. "Automating the Provisioning and Configuration of Devices in the Internet of Things." In: *Complex Systems Informatics and Modeling Quarterly* 9 (2017), pp. 28–43. URL: https://doi.org/10.7250/csimq.2016-9.02 (cit. on pp. 15, 17).

[Ins16] Institute of Electrical and Electronics Engineers. *IEEE 802.11-2016*. Tech. rep. Institute of Electrical and Electronics Engineers, 2016. URL: https://www.techstreet.com/ieee/standards/ieee-802-11-2016?product_id=1867583 (cit. on p. 16).

[Jaz14] N. Jazdi. "Cyber physical systems in the context of Industry 4.0." In: *2014 IEEE Automation, Quality and Testing, Robotics* (2014), pp. 2–4. DOI: 10.1109/AQTR.2014.6857843. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=6857843 (cit. on p. 15).

[JLF+13] A. J. Jara, P. Lopez, D. Fernandez, J. F. Castillo, M. A. Zamora, A. F. Skarmeta. "Mobile digcovery: A global service discovery for the internet of things." In: *Proceedings - 27th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2013* (2013), pp. 1325–1330. DOI: 10.1109/WAINA.2013.261 (cit. on pp. 26, 27, 35).

[JMS12] A. J. Jara, P. Martinez-Julia, A. Skarmeta. "Light-weight multicast DNS and DNS-SD (lmDNS-SD): IPv6-based resource and service discovery for the Web of Things." In: *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)* (2012), pp. 731–738. DOI: 10.1109/IMIS.2012.200 (cit. on pp. 26, 30).

[KK12] R. Klauck, M. Kirsche. “Bonjour Contiki: A case study of a DNS-based discovery service for the internet of things.” In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7363 LNCS. 2012, pp. 316–329. ISBN: 9783642316371. DOI: 10.1007/978-3-642-31638-8_24 (cit. on pp. 25–27, 30).

[LLJC08] J. Leguay, M. Lopez-Ramos, K. Jean-Marie, V. Conan. “An efficient service oriented architecture for heterogeneous and dynamic wireless sensor networks.” In: *Proceedings - Conference on Local Computer Networks, LCN* (2008), pp. 740–747. DOI: 10.1109/LCN.2008.4664275 (cit. on pp. 25, 27).

[Luc05] D. Luckham. *The power of events: an introduction to complex event processing in distributed enterprise systems*. 3. print. Boston, Mass. ; Munich [u.a.]: Addison-Wesley, 2005, XIX, 376 Seiten. ISBN: 0-201-72789-7 (cit. on p. 15).

[Moc87] P. Mockapetris. *Domain Names - Concepts and Facilities*. Tech. rep. IETF, 1987, pp. 1–55. URL: https://tools.ietf.org/html/rfc1034 (cit. on p. 18).

[Mul07] G. Mulligan. “The 6LoWPAN architecture.” In: *Proceedings of the 4th workshop on Embedded networked sensors - EmNets ’07* (2007), p. 78. DOI: 10.1145/1278972.1278992. URL: http://portal.acm.org/citation.cfm?doid=1278972.1278992 (cit. on p. 33).

[Nel81] B. J. Nelson. “Remote Procedure Call.” PhD thesis. Pittsburgh, PA, USA, 1981 (cit. on p. 43).

[RBBM14] A. Radziwon, A. Bilberg, M. Bogers, E. S. Madsen. “The Smart Factory: Exploring Adaptive and Flexible Manufacturing Solutions.” In: *Procedia Engineering* 69.Supplement C (2014), pp. 1184–1190. ISSN: 1877-7058. DOI: 10.1016/j.proeng.2014.03.108. URL: http://www.sciencedirect.com/science/article/pii/S1877705814003543 (cit. on p. 15).

[RSZ06] C. S. Raghavendra, K. M. Sivalingam, T. Znati. *Wireless Sensor Networks*. Ercoftac Series. Springer US, 2006. ISBN: 9781402078842. URL: https://books.google.de/books?id=4zyDBwAAQBAJ (cit. on p. 15).

[SHB13] Z. Shelby, K. Hartke, C. Bormann. *Constrained Application Protocol(CoAP)*. Tech. rep. 2013, pp. 1–118. URL: https://tools.ietf.org/html/rfc7252 (cit. on p. 26).

[SLM05] R. Salz, P. Leach, M. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace Status*. Tech. rep. IETF, 2005, pp. 1–32. URL: https://tools.ietf.org/html/rfc4122 (cit. on p. 21).

[SWS+10] A. S. Shirazi, C. Winkler, A. Schmidt, P. Computing, U. I. Engineering. “SENSE - SATION : An Extensible Platform for Integration of Phones into the Web.” In: *Internet of Things (IOT)* (2010). DOI: 10.1109/IOT.2010.5678455 (cit. on pp. 16, 22, 26).

[TCZN17] F. Tao, Y. Cheng, L. Zhang, A. Y. C. Nee. “Advanced manufacturing systems: socialization characteristics and trends.” In: *Journal of Intelligent Manufacturing* 28.5 (June 2017), pp. 1079–1094. ISSN: 1572-8145. DOI: 10.1007/s10845-015-1042-8. URL: https://doi.org/10.1007/s10845-015-1042-8 (cit. on p. 15).

[VP13] O. Vermesan, Peter Friess. *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. 2013, p. 363. ISBN: 9788792982735. DOI: 10.2139/ssrn.2324902. arXiv: arXiv:1011.1669v3 (cit. on p. 15).

[ZJ15] A. Zaslavsky, P. P. Jayaraman. “Discovery in the Internet of Things: The Internet of Things (Ubiquity Symposium).” In: *Ubiquity* 2015.October (2015), 2:1–2:10. ISSN: 1530-2180. DOI: 10.1145/2822529. URL: http://doi.acm.org/10.1145/2822529 (cit. on p. 25).

All links were last followed on October 31, 2017.

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