Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Crawling Hardware for OpenTOSCA

Pushpam Choudhury

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	M.Sc. Kálmán Képes
Commenced:	24. April, 2017
Completed:	24. October, 2017
CR-Classification:	C.2.1, C.2.2, D.2.11, D.2.12

Abstract

Heterogeneity is the essence of the IoT paradigm. There is heterogeneity in communication and transport protocols, in network infrastructure, and even among the interacting devices themselves. Managing discovery of the different devices in such a paradigm is an extremely complex task. The typical solutions include an abstraction layer, commonly known as the middleware layer, that handles this complexity for the devices, thereby, allowing them to interact with one another. One major limitation of the existing middleware solutions is in their ability to allow for an easily configurable approach required to handle the tremendous scale of heterogeneous components in the IoT. The objective of this thesis is to develop such a highly configurable discovery middleware approach. The proposed approach aims to discover a variety of heterogeneous devices and services depending on a multi-level plugin layer, consisting of independent plugins that interact with each other based on the pipes and filters architectural pattern. To allow for the dynamic configuration of the middleware, a discovery configuration is developed. The output from the middleware includes a list of devices and their capabilities and is accessible via a web interface which can interact with a range of different clients. The proposed approach is validated on a scenario in a real-life environment.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API Application Program Interface. 21

ICMP Internet Control Message Protocol. 23

IERC European Research Cluster on the Internet of Things. 19

IoT Internet of Things. 15

JSON JavaScript Object Notation. 55

MOM Message-oriented middleware. 21

OCF Open Connectivity Foundation. 23

OUI Organizationally Unique Identifier. 60

RDF Resource Description Framework. 23

REST Representational State Transfer. 62

RFID Radio-frequency identification. 19

SDP Service Discovery Protocol. 15

SLP Service Location Protocol. 15

SOA Service-Oriented Architecture. 21

SOC Service-Oriented Computing. 21

SOM Service Oriented Middleware. 21

SSDP Simple Service Discovery Protocol. 23

UDA UPnP Device Architecture. 23

UPnP Universal Plug and Play. 15

URI Unique Resource Identifier. 49

UUID Universally Unique Identifier. 49

WSN Wireless Sensor Network. 20

1 Introduction

The term Internet of Things (IoT) was first coined by Kevin Ashton in 1999 in the context of supply chain management [Ash09]. In recent years, the definition of *things* has evolved with the advancement in technology. Nowadays, one of the main objectives of the IoT domain is to combine various technologies and disciplines to enable connectivity between the Internet and physical devices [FKBT13]. With around 50 billion devices expected to be connected to the Internet by 2020 [Eva11], there is a paradigm shift in which more and more things are becoming interconnected and smarter every day. This presents us with our first major issue: *to enable seamless discovery of heterogeneous connected things, in a network*.

A *thing* can be a physical device like a laptop, a smart-phone or, a service like a Tomcat web server, MySQL database and so forth. These devices and services are heterogeneous in nature due to diversity in environments employed within: (i) the languages used to describe and advertise the specifications, (ii) the content and format of the transport layer protocols, (iii) the discovery behavior, for example, active or passive discovery mode, (iv) the network communication protocols, and, (v) the Operating System (OS) running on them. Before such heterogeneous things can communicate, they must be able to discover each other. Existing discovery protocols, such as, Service Location Protocol (SLP) [GPVD99; GVPK97], Universal Plug and Play (UPnP) [Don+15; Pre+08; UPnP00] or Bonjour [App13], are all limited by the above mentioned constraints. Therefore, a highly configurable discovery framework is needed with a scalable abstraction layer that can utilize the existing technologies and protocols. The scope of this thesis revolves around developing this configurable middleware for discovering the heterogeneous devices and services in the IoT.

The term *configurable discovery* is used in the context of being able to use a specific or a set of discovery protocols, as required to discover a target device. Some of the existing work try to combine the architectural and design similarities among existing discovery protocols [FGB11] for discovering the devices or, require an additional input such as a snapshot of an enterprise application to extract its various topologies [BBKL13]. In contrast, our work provides a configurable and scalable service-oriented plugin-based framework and a bottom-up discovery approach i.e., a network level approach in which plugins are configured to discover devices or services on a particular network. Each plugin uses a single protocol to enable discovery.

1.1 Problem Domain and Motivation

In general, most of the existing Service Discovery Protocols (SDP) for pervasive computing environments [Sil17], such as a *smart home* environment, depend on service advertisement messages from devices as defined by their manufacturers. This presents us with the problem of being able to intercept the message in the first place and then to understand its specifications. This typically requires an abstraction layer, which enables this translation between advertisement messages, so that devices need not worry about the difference in OS, network or transport protocols. Due to the presence of dozens of independent SDP, each addressing a different mix of issues, this message translation layer becomes highly complex and also increases the communication overhead [AHA13]. To fulfill the requirement of integrating existing technologies and protocols, a highly but easily configurable discovery middleware is required.

Our approach aims to maintain the diversity of the middleware layer by enabling the orchestration of different discovery protocols. A brief introduction about some of the protocols is provided in section 2.3. The proposed approach does not require any prior knowledge of the devices to be discovered, although an understanding of the plugins used for the discovery is required. The motivation behind this thesis is to extend the work done in the field of integrating OpenTOSCA to IoT [SBK+16]. The proposed discovery middleware can provide discovery services to the SmartOrchestra project[1] that aims at providing a marketplace for cloud-based IoT platforms.

1.2 Methodology

The framework proposed in this thesis attempts to discover devices and their services from the network layer up to the application layer, by employing a pipes and filters based orchestration of plugins. The plugins are responsible for discovering specific device capabilities. The current approach employs multi-level plugins, with each level aimed at discovering a particular service layer, such as the OS or the web server running on the device. This bottom-up approach is, therefore, both lightweight and independent of device's type, thereby, solving most of the issues related to device heterogeneity.

[1] http://smartorchestra.de/

1.3 Structure of the Report

This remainder of the report is organized as follows:

- Chapter 2 discusses relevant theoretical concepts and background necessary to understand the terms related to the thesis.
- Chapter 3 summarizes literature study related to existing IoT middleware solutions for device and service discovery and an existing plugin-based approach for automated discovery of application topologies.
- Chapter 4 describes the proposed middleware approach for discovering a device and its services using an easily configurable multi-level plugin layer.
- Chapter 5 presents a scenario-based validation of the approach and the corresponding implementation details.
- Chapter 6 includes the summary of the thesis, enlists the goals that have been achieved and suggests possible extensions for the thesis.

2 Fundamentals

A device discovery approach for the IoT needs to be highly customisable as there is a tremendous amount of heterogeneity in the IoT. In this chapter, all the important concepts necessary to understand the proposed customisable discovery approach, are explained. Firstly, the field of IoT and its main characteristics are outlined. Then the concept of device discovery paradigm is discussed, followed by a discussion about some existing discovery protocols. Lastly, the pipes and filters architecture pattern, used in the proposed discovery middleware, is highlighted.

2.1 IoT and its Characteristics

The IoT has created a dynamic network of billions of wireless things communicating with one another. The IoT connects the digital world to the physical world by bringing in new concepts like pervasive computing [Sil17], ubiquitous computing [Wei93], and ambient intelligence [Sad11]. Fig. 2.1 illustrates the European Research Cluster on the Internet of Things (IERC) vision for IoT, where "The Internet of Things allows people and things to be connected anytime, anyplace, with anything and anyone, ideally using any path/network, and any service" [SGFW10].

According to Razzaque *et al.* [RMPC16], the main characteristics of the IoT are as follows:

- *Heterogeneous devices:* Device heterogeneity is the backbone of the IoT infrastructure. The infrastructure must provide support for multivendor products providing different applications. Typically, the IoT is composed of simple embedded devices and sensors to heavy-duty computing devices for routing, switching, data processing, etc. In order to connect all these different devices to the Internet, the infrastructure must enable connectivity via WiFi, cellular networks, and low power radios.
- *Resource-constrained:* Most of the IoT devices have limited computational, networking, and storage capabilities. For example, Radio-frequency identification (RFID) tags may lack any processing capacity or battery power in them.

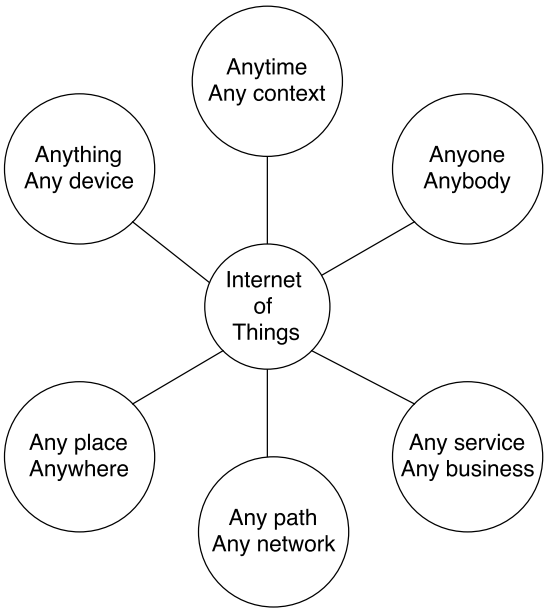

Figure 2.1: Definition of IoT [SGFW10]

- *Real time:* For some of the IoT domains, such as, healthcare and transportation, the on-time delivery of data and services is essential. Delayed delivery may render the application useless and can cause catastrophic effects. For example, in flight control systems, failure to provide real-time updates can cause an accident.
- *Complex network of devices and a large number of events:* The IoT networks are much larger than the traditional networks. In places like supermarket or university, thousands of devices may interact with each other. As predicted by Gartner [RM13], the IoT will be a global ultra-large-scale network containing billions and even in trillions of nodes. Thus, an enormous number of events will be generated as the IoT devices interact with each other. If not handled properly, this may cause event congestion, thereby, reducing the capabilities of the IoT devices.
- *Everything-as-a-service (XaaS):* As more and more things get connected, the services are also expected to grow and be available to consumers for usage. Benefits of a XaaS model is its efficiency, scalability, and ease of usage [BFB+11]. For example, in the field of Wireless Sensor Networks (WSN), sensing is being provided as a service [TS13], which will eventually lead to a XaaS model.
- *Context-awareness:* In order to create value from a large amount of data generated from a large number of sensors, the context in which the data is generated needs to be understood. It basically eases the interpretation of data. For example, location-awareness is an integral part of context-awareness.

- *Distributed:* The data generated by the IoT applications must always be available. To provide this availability of data, a distributed network of nodes is required. Due to this distributed structure, according to Brewer's conjecture [Bre00], popularly known as the CAP theorem, it is impossible for a distributed web service to provide guarantees for consistency, availability and partition tolerance. So an agreeable trade-off will be required.
- *Security:* In order to realize the vision of the IoT i.e., to allow access to any service by anyone, at any time and anyway, all the security leakage in different applications and networks must be addressed. This tremendously increases the complexity of security mechanisms.

Any discovery approach for the IoT must take all the above-mentioned characteristics into account. Nowadays, middleware solutions are gaining popularity in the field of device discovery for IoT. In the following section, the device discovery paradigm for the IoT is discussed.

2.2 Device Discovery Paradigm

In IoT, it is difficult to enforce a common standard for all the diverse devices representing different domains in IoT. Therefore, there is a requirement for an abstraction layer among applications of diverse domains. A middleware provides such an abstraction, by providing an Application Program Interface (API) to physical layer communications up to required services, covering all the complexities of heterogeneity [BSMD11]. Based on existing design approaches the various middleware solutions for IoT paradigm, can be grouped according to [BSMD11], into the following categories:

- *Event-based:* In an event-based middleware, events are used for interaction among different components and applications. The events propagate from the sending application components (producers) to the receiving application components (consumers). Each event has associated with itself a set of typed parameters which helps in identifying a change in producer's state. Message-oriented middleware (MOM) [Cur05] is an example for an event-based middleware. Instead of events, MOM relies on messages that are passed on between senders and receivers.
- *Semantic model-driven:* A semantic model-driven middleware maps physical devices to semantic devices. the mapping can range from one-to-one to many-to-one. Semantic devices are basically software representation of physical devices. Information such as device capabilities, services, and device security properties are usually included in the semantic device description.

- *Service-oriented:* Service-Oriented Computing (SOC) [Pap03] is based on Service-Oriented Architecture (SOA) approaches that relies on software or applications as services. The characteristics of SOC, like technology neutrality, loose coupling, service reusability, service composability and service discoverability, are potentially beneficial to IoT applications. A Service Oriented Middleware (SOM) can alleviate the challenges of IoT's ultra-large-scale network, resource-constrained devices, and mobility characteristics, by provisioning of appropriate functionalities to deploy, publish/discover, and access services at run-time.
- *Agent-based:* For an agent-based middleware, applications are divided into modular programs to facilitate injection and distribution through the network using mobile agents. For such an architecture it is assumed that agents are associated with all devices in the network. This approach provides potential benefits such as asynchronous and autonomous execution, protocol encapsulation, network load and latency reduction, and fault-tolerance. Additionally, agent-based approaches consider the resource limitations. For example, UBIWARE [KKK+08] middleware supports the creation of an autonomous, complex, flexible, and extensible industrial systems.
- *Database-oriented:* For a database-oriented middleware, the sensor network in the IoT is viewed as a virtual relational database system. The applications can query the required data from the database using SQL-like languages.

All these different IoT middleware solutions perform discovery based on the different existing discovery protocols, a discussion for which is presented in the next section.

2.3 Discovery Protocols

In the IoT paradigm, the networks will dynamically change and continuously evolve. Heterogeneous devices connect and disconnect at every instant. Therefore, automated discovery mechanisms are essential, without which it is impossible to achieve a scalable and accurate network management. Dynamic network discovery mechanisms allow dynamic, run-time configuration of connections, thereby, enabling devices to adapt to the changing contexts.

The existing discovery protocols can be broadly classified into two categories:

- *SDP:* SDPs rely on service descriptions according to which discovery is performed. These service descriptions can be in XML format, attribute-value pairs or a Resource Description Framework (RDF).
- *network discovery protocol:* Network discovery protocol checks for the reachability of a network device. Ping is such a network utility which is based on the Internet Control Message Protocol (ICMP) protocol.

In this thesis, the discovery architecture is based on two types of plugins: (i) active discovery plugins, which perform network scanning using the network discovery protocol such as the ICMP, and (ii) passive discovery plugins, which are based on different SDPs. A brief introduction, to some of the SDPs and the ICMP protocol, is provided in the following sub-sections.

2.3.1 UPnP

The UPnP specification, which was originally maintained by the UPnP Forum, formed in October 1999, is now being managed by the Open Connectivity Foundation (OCF), effective January 1, 2016. The UPnP specification has three versions: (i) version 1.0, released in June 2000 [UPnP00], (ii) version 1.1, released in October 2008 [Pre+08], and (iii) version 2.0, released in February 2015 [Don+15].

UPnP provides a platform for pervasive peer-to-peer network connectivity of intelligent appliances, PCs, and wireless devices. The UPnP Device Architecture (UDA) is designed to support zero-configuration, dynamic networking, and automatic discovery for a range of device categories. The motivation behind this specification, according to the official documentation, is to allow a device to dynamically join a network, obtain an IP address automatically, convey its capabilities, know about the presence and capabilities of other devices and, finally, leave the network smoothly and automatically without leaving any unwanted state behind.

The first step in UPnP networking is discovery. The UPnP discovery protocol, known as the Simple Service Discovery Protocol (SSDP), allows a device to advertise itself to the control points or the control point to search for devices, on the network. When a device joins a new network, it must multicast discovery messages advertising itself, its embedded devices, and its services for any control point on the network. Similarly, when a control point is added to a network, it can multicast a discovery message for devices or services of interest or both. Also in the event of leaving a network, a device should, if possible, multicast discovery messages declaring that it will no longer be available. All the discovery messages are based on the SSDP.

After a device becomes available, it shall multicast discovery messages to advertise its capabilities. Each discovery message contains four major parts in the message header:

- *NT (Notification Type):* This field refers to the device type.
- *USN (Unique Service Name):* This field points to the composite identifier for the advertisement.
- *LOCATION:* This field represents the device URL for more information about the device.
- *CACHE-CONTROL:* It represents the duration for which the advertisement message is valid.

When a device is added to the network, it shall send a multicast message with method *NOTIFY* and *ssdp:alive* in the *NTS* header field in the format as shown in listing 2.1.

Listing 2.1 Format for the ssdp:alive message

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age = seconds until advertisement expires
LOCATION: URL for the UPnP description of root device
NT: notification type
NTS: ssdp:alive
SERVER: this field contains the OS version as well as the UPnP version
USN: Field value contains Unique Service Name
BOOTID.UPNP.ORG: an integer number that increases after every initial announement message
CONFIGID.UPNP.ORG: an integer number used for caching description information
SEARCHPORT.UPNP.ORG: an integer number that identifies the port on which device responds
    to unicast M-SEARCH messages from control points
```

The *HOST* field value of the advertisement is the standard multicast address specified for the protocol (IPv4 or IPv6) used on the interface. For IPv4, its value is *239.255.255.250:1900*. The *SERVER*, *BOOTID.UPNP.ORG* and *CONFIGID.UPNP.ORG* header fields are also required in every discovery message. The header field *SEARCHPORT.UPNP.ORG* is optional. The description of these header fields is provided in the listing 2.1.

When a device and its services become unavailable, the device should multicast a *ssdp:byebye* message corresponding to each of the *ssdp:alive* messages it sent as a multicast which is not already expired. Each of the multicast messages must have method *NOTIFY* and *ssdp:byebye* in the *NTS* header field, as shown in listing 2.2.

Listing 2.2 Format for the ssdp:byebye message

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
NT: notification type
NTS: ssdp:byebye
USN: Contains the Unique Service Name
BOOTID.UPNP.ORG: an integer number that increases each time device sends an initial
    announce or an update message
CONFIGID.UPNP.ORG: an integer number used for caching description information
```

The third type of *NOTIFY* message has an *NTS* header field of *ssdp:update*. This message is used by a device to announce to a control point about changes in the root device, the embedded devices and the embedded services. The device can send this message as soon as its IP address changes. An example of this message format is provided in listing 2.3.

Listing 2.3 Format for the ssdp:update message

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
LOCATION: URL for the UPnP description of root device
NT: notification type
NTS: ssdp:update
USN: Contains the Unique Service Name
BOOTID.UPNP.ORG: an integer number increased each time device sends an initial announce or
    an update message
CONFIGID.UPNP.ORG: an integer number used for caching description information
NEXTBOOTID.UPNP.ORG: new BOOTID value that the device will use for subsequent
    announcements
SEARCHPORT.UPNP.ORG: an integer number which identifies port on which device responds to
    unicast M-SEARCH
```

The *NOTIFY* messages do not have a message body, but the message must have a blank line following the last header field.

The second type of message is an *M-SEARCH* message. This message is sent when a control point wants to search for devices in the network. The control point sends a multicast, a search message with search target, on the reserved address and port (239.255.255.250:1900). An example format for the search request with method *M-SEARCH* is shown in listing 2.4.

Listing 2.4 Format for the M-SEARCH message

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: seconds to delay response
ST: search target
USER-AGENT: OS version and UPnP version
CPFN.UPNP.ORG: a friendly name of the control point
CPUUID.UPNP.ORG: uuid of the control point
```

The different header fields of to the *M-SEARCH* message are as follows:

- *MAN:* This header field is required and it must be enclosed in double quotes as shown in listing 2.4.
- *MX:* This required field refers to the wait time in seconds. This value must be greater than or equal to 1 and less than 5.
- *ST:* This required field contains the information regarding the search target.
- *USER-AGENT:* It is specified by the UPnP vendor and includes the device's OS, the UPnP version, and the product version.
- *CPFN.UPNP.ORG:* It represents the friendly name for a control point, which is specified by the vendor.
- *CPUUID.UPNP.ORG:* It represents the UUID of the control point.

The *M-SEARCH* messages can be unicast, in which case the hostname and the port number must be specified in the *HOST* header instead of the reserved multicast address and port specified in listing 2.4.

2.3.2 Bonjour

The Bonjour zero-configuration networking architecture [App13] provides support for publishing and discovering TCP/IP-based services in a local area or a wide area network. Bonjour is Apple's implementation of zero-configuration networking protocols [SC05]. Bonjour basically covers three areas for zero-configuration networking in IP networks:

- allocating IP addresses to hosts,
- using host names instead of IP addresses, and
- automated service discovery on the network.

Bonjour uses self-assigned link-local addressing to assign IP addresses to devices on the local network. When a device joins a network, Bonjour finds an unused local address and assigns it to the device. For name-to-address translation on a local network, the protocol uses Multicast DNS (mDNS), as defined in RFC6762 [CK13], in which DNS-format queries are sent over the local network using IP multicast. Because these DNS queries are sent to a multicast address, each service or device can provide its own DNS capability. When a device or service encounters a query for its own name, it provides a DNS response with its own address. For a correct name-to-address translation, a unique name on the local network is required.

The final element of Bonjour is service discovery, which allows applications to find all available instances of a particular type of service and to maintain a list of named services and port numbers. Service discovery in Bonjour is accomplished by *browsing*. An mDNS query is sent out for a given service type and domain, and any matching services reply with their names. Therefore, a list of matching services to choose from becomes available. Additionally, Bonjour takes the service-oriented view, which means queries are made according to the type of service needed, not the hosts providing them.

A Multicast DNS responder registers the devices or applications that want to publish a service. After registration of a service, three DNS records are created: a service (SRV) record, a pointer (PTR) record, and a text (TXT) record. The SRV record maps the name of the service instance to the information needed by a client to actually use the service and typically includes two pieces of information to identify a service: i) hostname, and ii) port number. The created SRV record consists of following parts:

- the instance name which refers to the name of a service instance,
- the service type which is a standard IP protocol name, preceded by an underscore, and
- the domain which refers to a standard DNS domain.

The PTR records include the service type and the domain, but they do not include an instance name. The TXT record includes additional data needed to resolve or use the service. The TXT record is often empty.

Bonjour provides various APIs at multiple layers for OS X an iOS with support for different languages.

2.3.3 SLP

The SLP version 1 and version 2 as defined in RFC2165 [GVPK97] and RFC2608 [GPVD99] respectively, provides a flexible and scalable framework for providing information about the existence, location, and configuration of networked services. SLP eliminates the need for users to know the names of network hosts. A user only needs to know the description of the desired service, based on which, SLP is able to discover the URL for the service. In SLP, all the activities are carried out by three software entities termed as agents:

- *User Agent (UA):* It is a software entity which is looking for a one or more services and provides client applications with a simple interface for accessing registered service information.
- *Service Agent (SA):* This software entity advertises the location of one or more services. SLP advertisement messages include multicast messages and unicast responses to queries.
- *Directory Agent(DA):* This software entity acts as the centralized repository for the service location information.

These agents communicate with each other to provide the necessary framework. The messages related to service management are as follows:

- *Service Request (SrvRqst):* This message is sent by UAs to SAs and DAs to request for the location of a service.
- *Service Reply (SrvRply):* This message is sent by SAs and DAs in response to a SrvRqst message. It contains the URL of the requested service.
- *Service Registration (SrvReg):* Message sent by SAs to DAs to inform about a service that is available.
- *Service Deregister (SrvDeReg):* Message sent by SAs to DAs to inform about a service that is no longer available.
- *Service Acknowledge (SrvAck):* An acknowledgment message sent by DAs to SAs in response to SrvReg and SrvDeReg messages.

Also in order for UA's to find the different services, SA's send an advertisement message, *SA Advertisement (SAAdvert)*, to let UA's know where they are. Similarly, the DA's send an advertisement message, *DA Advertisement (DAAdvert)*, to let SA's and UA's know where they are. The SLPv2 was designed to be more secure and to provide a scalable solution for enterprise service location. An API for SLP is provided in [KG99].

2.3.4 ICMP

The ICMP [Pos81] is a part of the Internet Protocol (IP) Suite and was designed for the purpose of error reporting by gateway devices or routers during an error in datagram processing. ICMP is an integral part of IP and must be implemented by every IP module. So, all IP network devices have the ability to send, receive or process ICMP messages. The IP is not designed to be absolutely reliable and the purpose of ICMP error messages is to provide feedback about any problem during communication of IP packets.

The ICMP provides error reporting, flow control, and first-hop gateway redirection capabilities. The ICMP messages are sent in various situations, for instance, during failure of a datagram to reach its destination, when the gateway does not have enough buffering capacity to forward a datagram, and when the gateway directs the host to send traffic on a shorter route. The ICMP messages are sent using the basic IP header. The first octet of the data portion of the datagram refers to the ICMP type field, which determines the format of the remaining data. The ICMP header can be divided into:

- *Type:* It specifies the ICMP message format. A subset of the different values of this field along with the corresponding message description is provided in table 2.1.
- *Code:* This field is 8 bit long and is used for further qualifying the ICMP message.
- *ICMP Header Checksum:* It is 16 bits long and provides the checksum for covering the ICMP message.
- *Data:* It is of variable length and contains the data specific to the message type indicated by *Type* and *Code* fields.

Type	**Description**
0	Echo Reply
3	Destination Unreachable
5	Redirect
8	Echo
11	Time Exceeded
12	Parameter Problem

Table 2.1: Some example values for the type field in ICMP header

A ping sweep or an ICMP sweep is a basic network scanning technique, which is used to find out the destination hosts that are alive for a given IP range. The *ping* program includes a client interface to ICMP. Apart from checking if the host is alive, the ping program also collects performance statistics such as the measured round trip time and the number of times the remote server fails to respond. Each ICMP Echo message contains a sequence number, starting from 0, that increments after each transmission and a time-stamp value that indicates the transmission time. Apart from ping sweep, ICMP can be used as a tool for OS fingerprinting as well. This feature is utilized in our implementation presented in chapter 5.

2.4 Pipes and Filters Architectural Pattern

An IoT middleware is a highly complex system consisting of numerous heterogeneous components. In order to integrate these heterogeneous components, the complex process needs to be divided into a sequence of smaller and independent processing steps. As a result of the combination of these independent components, the overall architecture become loosely-coupled, which in turn makes the architecture more tolerant to changes. Such an architecture can be realized using the pipes and filters architectural pattern.

In the pipes and filters architectural style [GS94], every component has a set of inputs and a set of outputs. Each component reads a stream of data at its input and produces a stream of data on its output. This process is accomplished via a local transformation to the input stream and incrementally computing the output. Therefore, these components are termed as *filters*. The connectors between filters transmit the outputs of one filter to the inputs of another. Therefore the connectors are termed as *pipes*. Figure 2.2 provides a visual representation of the pipes and filters architectural pattern.

In figure 2.2, the input to the filter chain is data from another filter or a data source. Similarly, the output of the filter chain is sent to another filter or to a data sink. Pipes and filters, perceived as filter chains, constitute the building blocks of this architectural pattern. The pipes and filters pattern employs abstract pipes to decouple the filter components from each other. The pipe receives an input from a filter component and passes it on to another component for processing, without the knowledge of the first component, based on the desired functionality.

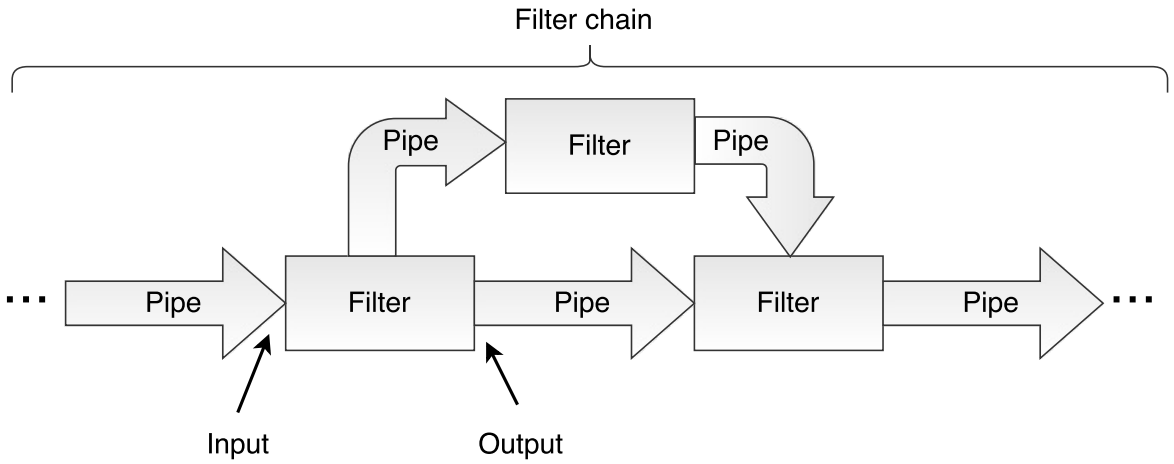

Figure 2.2: Pipes and filters architectural pattern

2.4.1 Properties

The pipes and filters pattern has a number of useful properties:

- it enables the designer to comprehend the overall input/output behavior of the entire system as a simple composition of the behaviors of the individual filters,
- it supports reuse by connecting any two filters together, provided the data to be transmitted is agreed upon between them,
- it allows easy maintenance of the system by adding or removing filters, and
- it supports concurrent execution, where each filter can be implemented as a separate task and executed in parallel with other filters.

2.4.2 Implementation of Pipes and Filters Pattern

This architectural pattern is beneficial for applications where large amounts of data are to be processed, such as with web servers and rendering, imaging or sound processing as well as applications for message processing. For example, in their book on enterprise integration patterns, Hohpe and Woolf [HW03] envisioned a messaging channel based on the pipes and filters pattern. In their architecture, each filter exposes a very simple interface:

- to receive a message from the inbound pipe and process it, and
- to publish the message after processing through an outbound pipe.

The pipe transfers the message from one filter to the next. All the components use the same external interface and therefore, the same set of filters can be used for different solutions by rearranging them to form a new sequence.

2.5 Summary

In order to perform device discovery in the IoT domain, this thesis attempts to generate a multiagent-based middleware solution. These agents or plugins, use multiple discovery protocols to discover the devices and communicate with each other using a pipes and filters design pattern. This chapter has stated all the necessary fundamentals needed to understand the suggested approach proposed in this thesis.

3 Related Work

This chapter discusses the existing literature related to the thesis work. As the focus of this thesis is to develop a device discovery middleware solution for the IoT, we discuss two existing device discovery middleware approaches in section 3.1. Then, due to the plugin-based approach of the proposed discovery service, we look into an existing agent-based IoT middleware solution in section 3.2, and lastly, we discuss an approach that demonstrates how the plugin-based architecture can be used to discover an application topology in section 3.3.

3.1 Device Discovery Middlewares based on SDPs

The SDPs such as UPnP [Don+15; Pre+08; UPnP00], and SLP [GPVD99; GVPK97] enable devices to advertise their capabilities to other devices and search for required services, with minimal or no human intervention. In the proposed discovery middleware, the passive discovery plugins use these SDPs to discover desired devices. In their survey, Feng Zhu et al. [ZMN05] provided a classification for the existing SDPs based on their:

- discovery infrastructure which can either be directory-based or non-directory based,
- discovery scope which are based on network topologies, user roles and context information,
- service selection that is either manual or automatic, and
- service status inquiry which ranges from polling periodically to transient event notifications.

In order to handle the heterogeneity in IoT, approaches are required that can integrate the different SDPs. A brief description of two such approaches are provided in following sub-sections.

3.1.1 SeDiM Middleware

The SeDiM is a *Service Discovery Middleware* solution [FGB11]. It was developed by Carlos Flores and his colleagues at the Lancaster University in February 2011. This approach provides a middleware solution, which supports heterogeneous discovery protocols, to allow interoperation between different service domains. The main objective of this framework is to provide multi-protocol interoperability with legacy applications. The SeDiM middleware utilizes the architectural and design commonalities of the existing discovery protocols.

The SeDiM middleware can achieve interoperability between protocols in the same domain as well as between protocols in different domains. A case study is presented in the paper which highlights this capability of the middleware. After deploying the middleware on the devices, the middleware allows them to locate required services irrespective of the underlying SDP. The authors highlight following features of the middleware to provide these capabilities:

- A framework based on configurable components to allow dynamic configuration of the SDPs.
- An abstraction for discovery events so that protocol-specific messages can be understood and translated between one another. All messages receive by the middleware are converted into this intermediary format.
- A bridge component known as the domain hub, that ensures messages are forwarded between different SDPs.
- An abstraction for the service description so that services from different protocols can be matched. Therefore, all advertisement and search messages are translated into this abstract format.

The main contribution of the SeDiM middleware is that it provides a solution for handling following issues related to heterogeneous SDPs:

- Heterogeneity in discovery model behavior which the middleware solves by using a configurable component framework,
- Heterogeneity in message content which is solved by abstracting the discovery event messages to understand, process, and use translation to the target protocol, and
- Heterogeneity in service description which is addressed through an abstraction for the service description messages.

The limitation for this middleware solution, as pointed out by its authors, is that it does not support semantic device discovery and lacks any mechanism for matching non-functional features. Apart from these, since the focus of this solution is to enable interoperability between legacy devices, no studies have been conducted to check the feasibility of this middleware for the discovery of resource-constrained devices in the IoT.

3.1.2 MUSDAC Middleware

MUSDAC stands for the "MUlti-protocol Service Discovery and ACcess" middleware platform, as presented in [RICL06]. The objective of this middleware solution is to handle the interoperability issues related to the existing discovery and access protocols as well as to manage the inter-connectivity between different networks in a dynamic multi-network environment. This platform is designed keeping the heterogeneity of pervasive computing environment in mind. It enables clients to interact with heterogeneous services as well as services present on a different network than that of the client.

The MUSDAC platform is based on all-IP networking environment consisting of loosely connected and highly heterogeneous networks. This network heterogeneity allows devices connected via different networks such as cellular networks and home networks to interact with each other. Each network has their own MUSDAC instance which is provided as a service through existing discovery protocols. These MUSDAC instances interact with one another to distribute discovery requests and provide remote access to services. An overview of the MUSDAC platform is provided in figure 3.1.

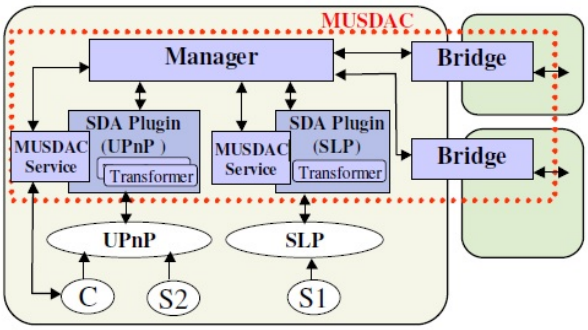

Figure 3.1: Overview of the MUSDAC platform [RICL06]

The different components highlighted in the figure 3.1 are described as follows:

- The Manager component handles all discovery and access requests within a network to provide services for local as well as remote clients.
- The Service Discovery and Access (SDA) plugins whose activities are to interact with and collect service information from individual SDPs, register MUSDAC service for local clients, and access services on behalf of remote clients.
- The Transformer component add context information to the service description generated by the SDA plugins.
- The Bridge component allows the Manager component to perform service discovery and access for remote networks.

One major advantage of MUSDAC middleware approach over that of SeDiM is that in MUSDAC, context-awareness is included in service descriptions. In MUSDAC, context information regarding the network environment, the interacting clients, and the service instance are collected. This allows MUSDAC to provide a better match for the services requested by the clients as it also takes the environment into account. The context information has two parts:

- context parameters which represent the static and dynamic attributes of the entity, and
- context rules which refer to control policies such as preferences, choices and filters used for controlling the desired resources, type of services and so on.

The major contribution of MUSDAC can, therefore, be summarized as a platform for enabling client applications to discover and access desired services irrespective of their communication protocols and location in a multi-network environment. One limitation of the MUSDAC middleware approach, as pointed out by the authors, is the increase in communication overhead during the service discovery phase. This delay in communication may cause problems for the IoT applications which rely heavily on real-time data.

3.2 An Agent-based Middleware for the IoT

In their survey, Chaqfeh and Mohamed [CM12] highlight the existing technical challenges for an IoT middleware. These challenges include interoperability, scalability, abstraction provision, security and privacy, and so on. In [KKK+08], a vision for a middleware for the IoT is described. This vision forms the basis of the UBIWARE research project which utilizes the *agent technology*, and is described in the next section.

3.2.1 UBIWARE project

The objective of the UBIWARE research project is the development of a new generation of middleware platform which is basically a self-managed complex system. The system consists of distributed, heterogeneous, shared and reusable components. These components are of diverse nature comprising of sensors, RFIDs, smart devices and machines, web-services, software applications and others. This approach allows the various components to automatically discover each other and to configure the complex functionality of the middleware based on the functionalities of individual components.

The conceptual architecture of the project consists of following components:

- *resources:* These represent the different domains in the IoT.
- *agents:* The state for each resource is monitored by an autonomous software agent. This agent makes decisions on behalf of the resource such as to discover and request for services as required.
- *adapters:* The adapters mediate the connection between a resource and its agent. These adapters can be sensors, actuators, data structuring elements such as XML or semantic components for generating a semantic representation.

The semantic information envisioned in the project has a two-fold value. The first aspect of the semantic information is to monitor the heterogeneous resources and data integration across multiple resources. The second aspect is to enable behavioral control and coordination of the agents representing those resources. Thus, the context information includes the descriptive information about the services offered by the different resources as well as the perspective information regarding the expected behavior of the resources.

A major component of UBIWARE project is its *agent core*. All the different agents are based on this core. This agent core is depicted in figure 3.2. The agent core has three layers:

- the behavior engine implemented in Java,
- the declarative middle-layer with behavior models corresponding to different agent roles, and
- the third layer containing reusable atomic behaviors (RAB) which are shared and reusable resources interpreted as Java components.

A behavior model contains a set of behavior rules which specify the execution conditions for the different RABs. The behavior engine parses the RDF-based scripts in a behavior model and implements the run-time loop for an agent.

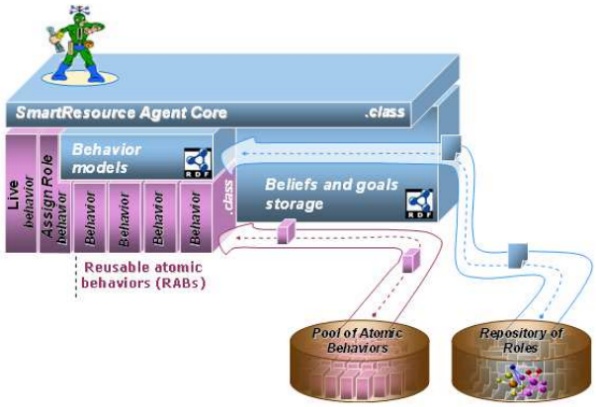

Figure 3.2: Core platform of UBIWARE project [KKK+08].

The major contribution of this project is that it directly addresses the IoT requirements and domains. It envisions to create a new generation of middleware solutions that enable the creation of self-managed complex systems. The UBIWARE research project handles the heterogeneity in the IoT very well and has become popular in the research and development community.

3.3 Usage of a Plugin Layer for Application Topology Discovery

In the proposed discovery middleware, we aim to discover the different capabilities of the discovered devices. This step is based on a configurable plugin layer which is controlled by the discovery service. Our current approach is inspired by the previous work done by Binz et al. [BBKL13] in the field of discovery and maintenance of enterprise topology graphs (ETGs).

An ETG [BFL+12] is a graph containing the fine-grained technical snapshot of an entire enterprise IT. Binz et al. proposed an extensible framework based on a plugin-based approach for automated discovery of different application layers and maintenance of ETGs. In the beginning, the authors highlight following requirements for the automated discovery and maintenance approach:

- To ensure the ETG quality, the completeness, accuracy, freshness and granularity of an ETG is taken into account,
- The framework must have the ability to integrate new types of components and relations to be discovered based on open world assumptions,

- The framework must possess the ability to integrate the different existing technologies to itself,
- The framework must be able to adapt to the frequent changes in the enterprise IT,
- The framework must minimize of operational impact caused by analyzing a production component of the enterprise IT.

The automated discovery approach is an iterative process. The discovery logic is provided by different type-specific plugins which can find information about a component or its relations with other components. The components of the enterprise IT are not aware of the presence of these plugins and do not push any information to them, rather the plugins pull the desired information from the components. Due to this architecture, the framework supports different protocols such as HTTP, SSH, SCP, etc, and different data formats such as XML, text, databases, property files, etc. The iterative discovery process is depicted below in figure 3.3.

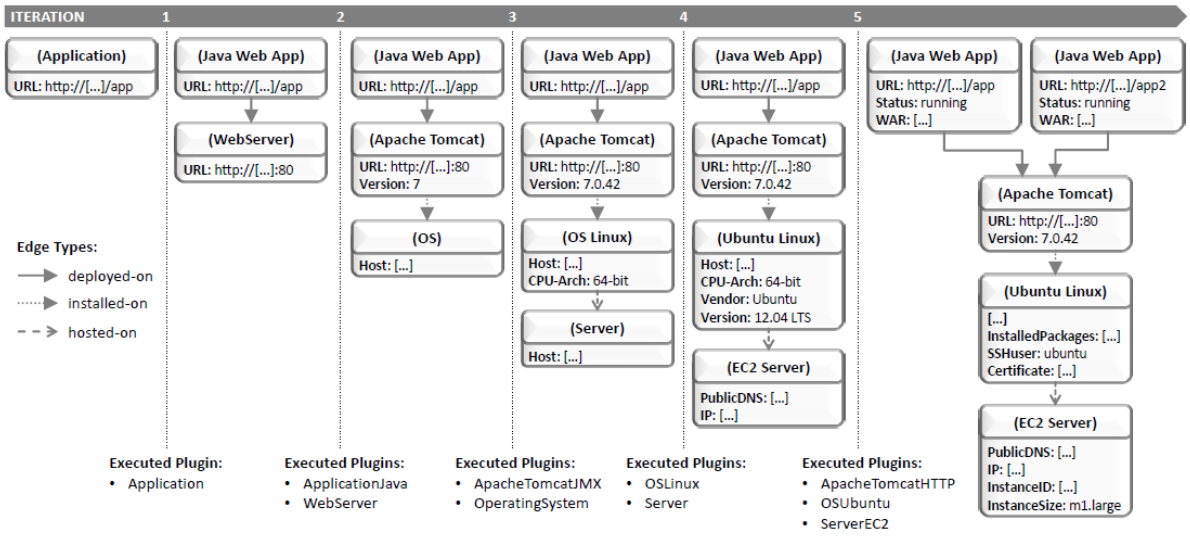

Figure 3.3: An example growth of an ETG before and after five iterations of the discovery approach [BBKL13]

The figure 3.3 shows the growth of an ETG, from a given starting node, after five iteration of the discovery approach. In each iteration one or more plugins are executed. Based on the use case, the starting node must be provided by the user or another system. Each of the executed plugins extract specific information about the provided node. For example, the *WebServer* plugin, executed in iteration 2 of figure 3.3, provides the type and version information of the web server running on the node. The *OperatingSystem* plugin identifies the type of OS hosted on the node, as shown in iteration 3 of figure 3.3. In the next iterations, OS-specific and server-specific plugins are executed to obtain the final ETG.

Through their framework, Binz et al. tried to fill the gap created due to the absence of an automated approach for obtaining and maintaining insights about an enterprise IT. Although our current approach for discovering device capabilities is inspired by this work, there are some major differences in the two approaches. First of all, our current approach discovers the device capabilities using a bottom-up approach, instead of the top-down approach followed by the Binz et al. Also due to the bottom-up approach, our discovery process does not require an application snapshot in the beginning as an input.

3.4 Summary

By studying the literature, concepts and related work stated in this chapter, we present our device discovery middleware in the next chapter. The work mentioned in section 3.3 is useful for the implementation of the prototype of the thesis. Further details about the approach is provided in chapter 4 and chapter 5.

4 Discovery Approach and Framework

The objective of this master's thesis is to design a highly configurable discovery middleware to discover IoT devices on the local network by combining common network protocols and the existing SDPs such as UPnP. The developed middleware also provides a list of services running on the device. For the network protocol based approach, the middleware *pulls* the data from alive hosts in the provided address space, and for the SDP based approach, the middleware collects data from the *push* notifications from the devices. So the proposed middleware supports both (i) active discovery behavior where the middleware searches for all devices currently on the network, and (ii) passive discovery behavior where the middleware listens for transient service announcements.

Both discovery behaviors have their own advantages, for instance, the active discovery provides an accurate depiction of all devices present in the network, whereas passive discovery is better for quickly finding devices that want to be discovered like an Internet Gateway Device. Apart from network-specific information, the developed middleware provides information about the discovered device, such as the device model, device vendor or OS running on the device. This chapter describes the requirements defined for the proposed device discovery middleware.

The architecture behind the proposed approach is based on pipes and filters design pattern of plugins, capable of finding a device and the services hosted by it. The idea is to pass on the information produced by one plugin to the next one. The proposed architecture extracts information using a bottom-up approach, starting from the network layer up to the application layers. For example, at the bottom level, network-specific information such as device's IP address is present, followed by device's model or vendor-specific information, then the OS installed on it or the type of web server hosted by it and so forth.

4.1 Requirements for an IoT Middleware

Based on the characteristics for an IoT middleware, as outlined in section 2.1, we have identified following requirements for our IoT middleware, which can be grouped into two broad categories: 1) the services a middleware should provide and, 2) the architectural features a middleware must have.

1. *Middleware service requirements:* The service requirements for an IoT middleware can be classified as functional requirements meaning the functions a middleware provides and nonfunctional requirements like Quality of Service (QoS) or performance of those functions. The generic functionalities of a middleware are as follows:

    - *Resource discovery:* Due to the heterogeneous nature of the IoT, resource discovery is the first and one of the most challenging tasks for an IoT middleware. It is necessary to automate this discovery process [RKLB09]. In the IoT, resource discovery can be divided into three main groups: i) centralized systems, where resource publication, discovery, and communication are generally managed by a dedicated server, ii) distributed systems, where every node announces its presence and the resources it offers, and iii) hybrid systems, like a centralized peer-to-peer (P2P) system, which includes multiple server nodes. In general, centralized systems are more efficient than distributed systems, whereas distributed systems are more robust as there is no single point of failure and also provide a better scalability. Hybrid systems aim to integrate the benefits of both centralized and distributed systems.

    - *Resource management:* An IoT middleware must provide services to manage the resource constrained devices in the IoT. This means the middleware must monitor the resources and resolve any resource conflicts, to provide the necessary QoS for the IoT application. Especially for service-oriented middlewares, as explained in section 2.2, the middleware needs to provide spontaneous resource management to satisfy application needs.

    - *Data management:* An IoT middleware should provide data management services like data acquisition services, data storage services and data processing services. Additionally, data pre-processing services must also be provided by the middleware to filter or compress the sensed data, before sending it to the IoT application.

Some of the major nonfunctional requirements of a middleware are given below:

- *Scalability:* The IoT middleware must be able to scale according to the growth of the IoT's network. Loose coupling and virtualization of components help in improving the scalability of the middleware.
- *Availability:* An IoT middleware must ensure availability of services, at all times, even if there is a failure in the system. The recovery time for the failure and the frequency of failures must be small in order to achieve required availability requirements.
- *Timeliness:* For real-time IoT applications, the middleware must ensure that the applications receive the data before the deadline expires. For IoT domains like health-care and transportation, on-time delivery of services is critical.
- *Ease-of-deployment:* The setup for the IoT middleware must be simple. Even if user interventions are required during the setup process, it must not require expert knowledge.

2. *Middleware architectural requirements:* The main advantage of a middleware is that it allows the application developers to focus on the design aspects rather than the architectural requirements of the application, which mainly are:
   - *Programming abstraction:* The middleware must provide services as an API to application developers. The abstraction is needed to isolate the development of an application from the functions provided by the underlying heterogeneous infrastructure.
   - *Interoperable:* The middleware must be able to handle heterogeneous devices or technologies without any additional effort needed by the application developer. The middleware should allow heterogeneous components to exchange data and services.
   - *Adaptive:* In order to tackle changes in the underlying network and the application requirements, the middleware must be adaptive. It must have provision for upgrading the technology to be able to evolve as per changes in requirements.
   - *Autonomous:* The middleware must be self-governed. The heterogeneous devices and applications must interact and communicate among one another without direct, or with minimal human intervention.

4.2 Architecture of Device Discovery Middleware

Before diving further into details of the approach, some assumptions are made which are clearly stated as below:

1. The discovery process is limited to a single-hop local-area or WiFi network.
2. The plugins can be deployed and run on the network.

We propose overall architecture of the discovery middleware as shown in figure 4.1.

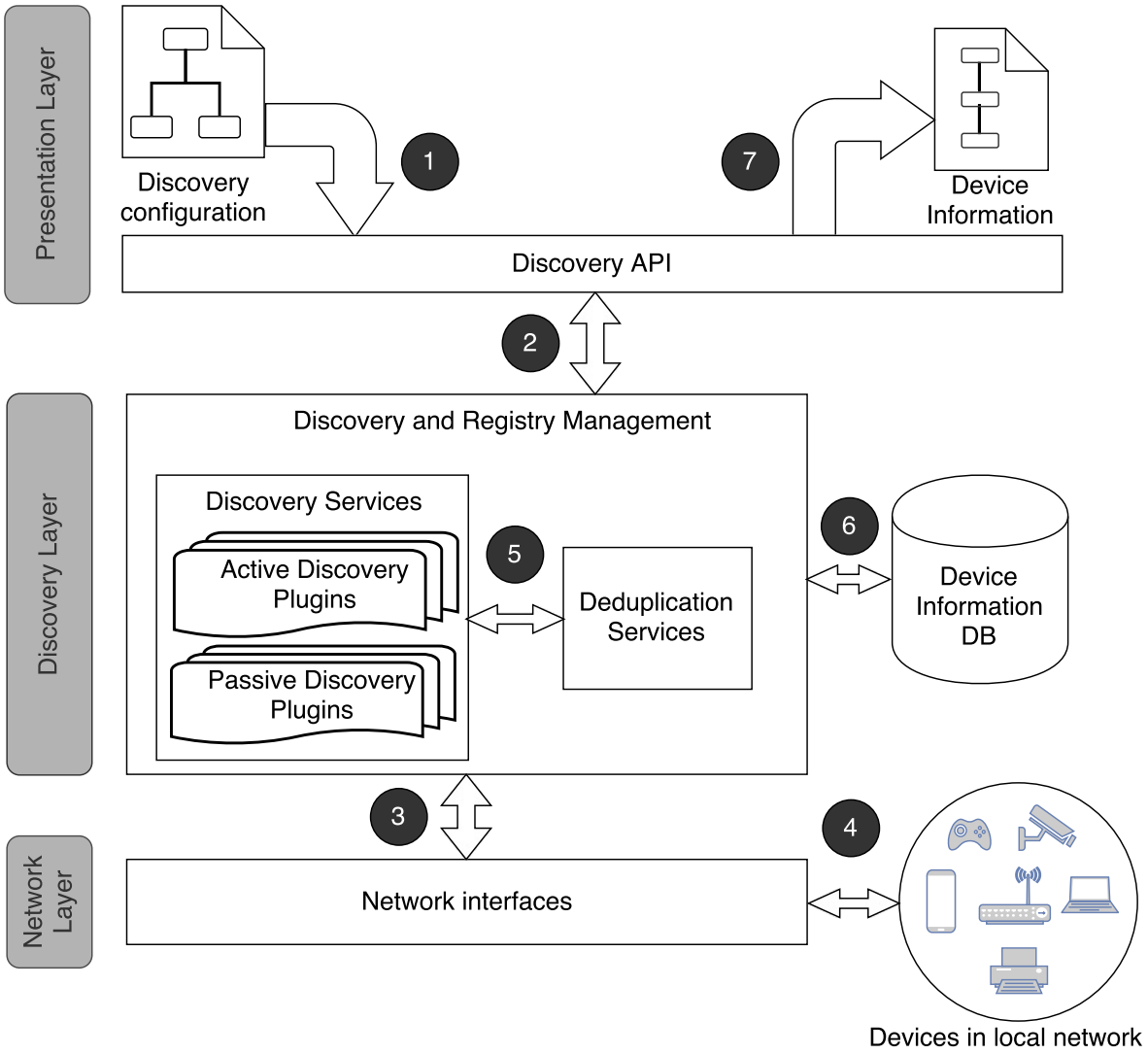

Figure 4.1: Layered architecture for device discovery middleware

The steps involved in the behavioral flow are as follows:

1. The discovery process is started, when a discovery configuration is received by the discovery API. The configuration details information about the list of plugins to be executed for the discovery process. The detailed structure of this discovery configuration file is provided in section 4.2.1.
2. The discovery API forwards this request to the Discovery and Registry Management component which starts the requested discovery process by activating the active or the passive discovery plugins. The detailed explanation of the procedure is present in section 4.2.2.
3. The discovery services search for the devices present in the desired network with the help of the activated plugins.
4. The discovery plugins, based on their type either pull the data from the devices or intercept the push notifications from them. The discovered data include network-specific as well as device-specific information.
5. After execution, the discovery services pass the data to the deduplication services, which perform three levels of data deduplication as explained in section 4.2.3.
6. After the discovery and deduplication of data, the Discovery and Registry Management stores the data in the device information database and returns the discovered information to the discovery API.
7. Finally, the discovery API sends the details about the discovered devices to the requesting client interfaces. The returned data includes a list of device and service-specific information. The middleware supports multiple messaging patterns for communication with the client applications. This component is explained in detail in section 4.2.4.

The proposed architecture, therefore, can be divided into four main structural components:

1. a Discovery Configuration designed for the purpose of managing data-flow among discovery plugins,

2. a Discovery and Registry Management component which manages the discovery of desired devices and the subsequent storage of the discovered information through following two services:
    - a *Discovery Service* which translates this discovery configuration and enables the discovery of the devices, and
    - a *Deduplication Service* that helps to maintain a single instance for each discovered device, and
3. finally, a Discovery API for interacting with the client applications.

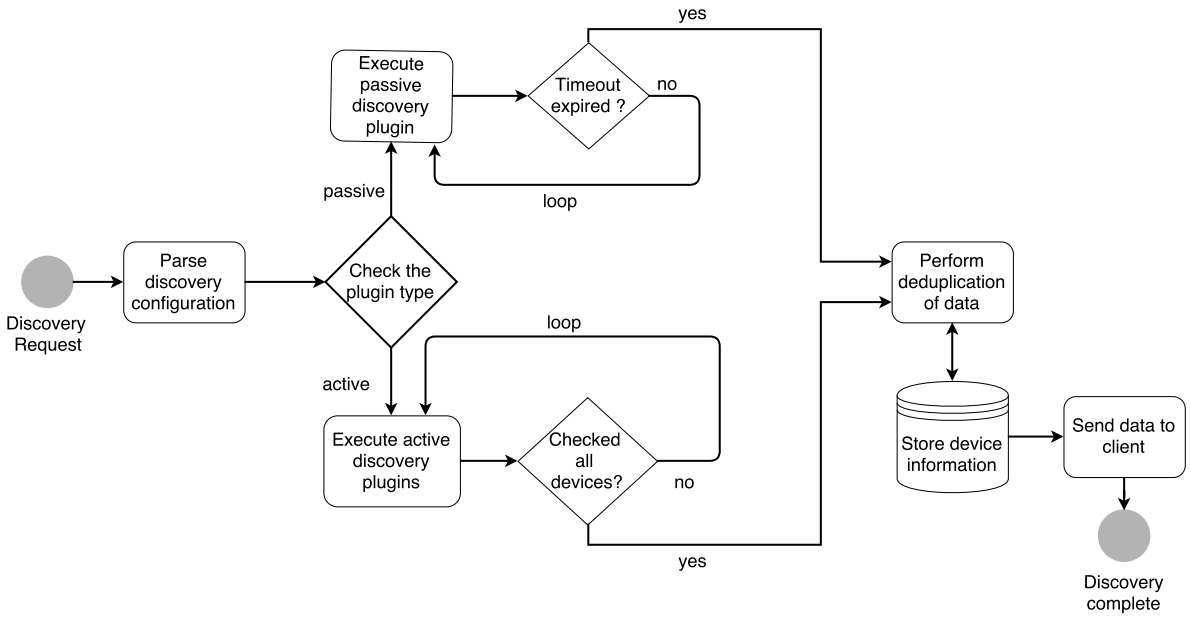

Figure 4.2: Device discovery middleware behavioral overview

Figure 4.2 depicts the behavioral overview of the proposed discovery middleware. After the initiation of the discovery process, the first step is to parse the discovery configuration file to determine the type of plugin to be activated for the discovery process. For active discovery plugins, typically a range of IP addresses is also included in the discovery configuration. The plugins are executed until all devices in the mentioned range are checked. For passive discovery plugins, the discovery configuration includes a timeout value. The passive discovery plugins keep listening to service advertisement messages until the timeout expires. After execution, the plugins forward the discovered data to the deduplication service. In the final step, the discovered data is returned to the requesting application. In the following sections, all the four components are described in detail.

4.2.1 Attributes of the Discovery Configuration

The discovery configuration enables the configurable discovery, based on plugins, for the proposed discovery middleware. This component generates the pipes and filters based design pattern among the discovery plugins. This section describes the different attributes of the discovery configuration. For the discovery process, plugins are divided into two major groups, namely *active discovery plugins* for extracting information using network protocols such as ICMP, and *passive discovery plugins* which extract information from the advertisement messages for the respective discovery protocol. In order to maintain the pipes and filters data-flow between the discovery plugins, the input and the output for each plugin must be clearly linked. Plugins may also require an additional information to execute the desired functionality. Keeping all these requirements in mind, we proposed some attributes for the discovery configuration format, listed in table 4.1.

Attribute	Description	Required
name	The name of the plugin	Yes
type	Distinguishes active discovery plugin from passive discovery one	Yes
config	A map of plugin specific properties	No
input	Represents the entries consumed by a plugin	No
output	Represents the entries generated by a plugin	Yes
nextPlugins	A list of plugin objects consuming the output generated by the current plugin	No

Table 4.1: Parameters for the discovery configuration of a discovery request

The explanation for each attribute is as follows:

1. **name**: It is a mandatory attribute and refers to the unique identifier for a plugin. On receiving this attribute, the discovery service activates the corresponding plugin.
2. **type**: This mandatory attribute refers to the discovery mechanism employed by the plugin to be activated. If the plugin uses network discovery protocol, then the value of this attribute is *active*, otherwise if the plugin is based on an SDP, then the value must be *passive*.
3. **config**: This attribute is optional and includes input parameters required for the activated plugin. These inputs are usually fixed and known in advance. For example, in order to discover the type of web server hosted by a device, a plugin will require the IP address and port number of the device. Another usage for this attribute could be to specify the execution mode for a plugin that has multiple functionalities. For example, in our implementation, we have used Nmap [Lyo09] plugin for identifying alive hosts as well as for identifying the type of OS running on them. Using the execution mode property in the *config* attribute, we can activate either host scanning or OS scanning functionality of Nmap plugin.

4. **nextPlugins**: This optional attribute enables the proposed architecture to realize the pipes and filters design pattern among the activated plugins. Each plugin can act as a parent for multiple other plugins known as child plugins, where each child plugin consumes the data produced by the parent plugin. This attribute, therefore, includes a list of discovery configurations for the child plugins.
5. **input**: This is also an optional attribute which refers to the run-time inputs required by a plugin. This attribute is useful for the child plugins that aim to discover a particular service for a device, and therefore, may require device-specific information such as its IP address as an input. This input information is provided by a preceding plugin at run-time.
6. **output**: It is understandably a mandatory attribute which refers to the output generated by a plugin. In an abstract view, the output generated by a plugin can be labeled as *"table-name:column"*. As the name suggests, the first part of this label points to the database table which stores the output of the discovery plugin, and the second part refers to the column of data generated by it.

4.2.2 Framework for Discovery Service

In this section, we discuss the various steps involved in the discovery service. The discovery service supports device discovery using both active discovery as well as passive discovery plugins. Figure 4.3 details the steps involved in the discovery procedure.

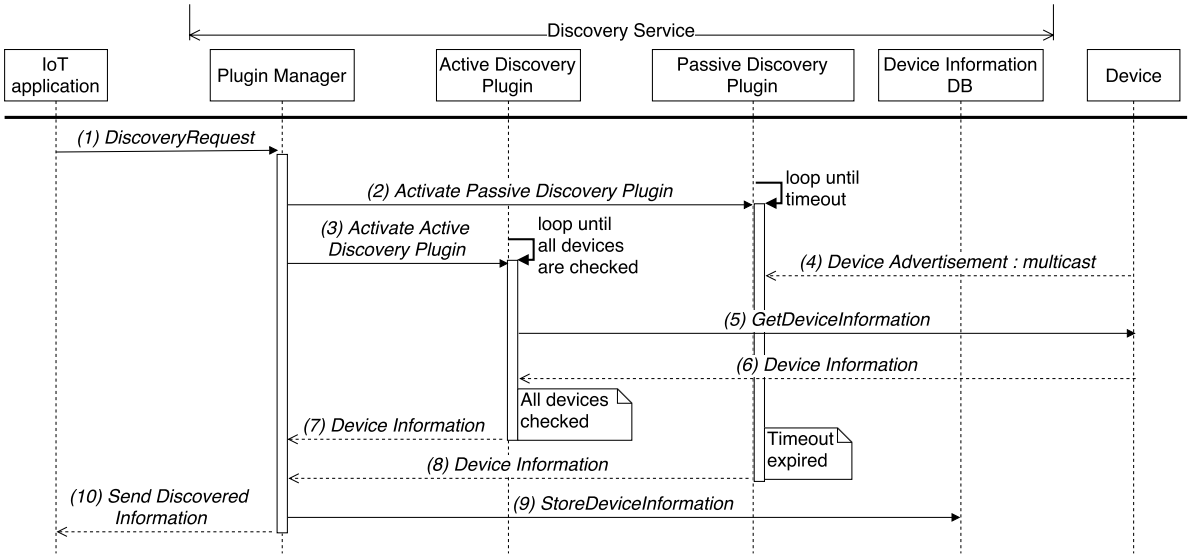

Figure 4.3: The steps involved in the device discovery procedure

The discovery service is initiated from the client end. After receiving the discovery request, the discovery service forwards the request to the Plugin Manager component as depicted by step (1) in figure 4.3. The discovery request includes a valid discovery configuration file, as explained in the earlier section. The Plugin Manager parses the discovery configuration file and checks the value of the attribute *type*. If the attribute has a value "active", then active discovery plugin is activated, otherwise, if the value is "passive", then passive discovery plugin is activated as illustrated by steps (2) and (3) in figure 4.3.

In the first case, the active discovery plugin searches for all devices in the IP range provided in the discovery configuration. The typical information retrieved by such a plugin includes the IP address and the MAC address of the devices. Steps (5) and (6) in figure 4.3 depict this information retrieval process. For the second case, the activated passive plugin listens for advertisement messages from devices, as shown in step (4) in figure 4.3. The advertisement message contains device specific information such as Universally Unique Identifier (UUID), Unique Resource Identifier (URI), model specific information and vendor details. Usually, a timeout value is mentioned for such a plugin, until which it listens for device advertisements.

At the end of execution, both the plugin forward the discovered information to the Plugin Manager, as shown in steps (7) and (8) in figure 4.3. The plugin Manager stores the device information, after deduplication of data, on the Device Information DB and also wraps this data in the required response format to present it back to the client, as depicted in steps (9) and (10) in figure 4.3.

4.2.3 Deduplication Service

The goal of deduplication service is to have only one entry for each discovered device on the network. We have not performed semantic matching of data for the purpose of deduplication. We have achieved the same via syntactic and type checking of the data.

We propose a three-level deduplication process as given below:

- *Configuration level deduplication:* This step is performed at the discovery configuration level. The discovery API receives multiple configurations from the clients. Some of these configurations will have the same discovery process outlined in them. Using the configuration deduplication method, these duplicate discovery configurations can be avoided and a single instance for the discovery configuration can be executed instead.

- *Plugin level deduplication:* This deduplication step is initiated by the discovery service itself. The goal for this step is to effectively manage the execution of a plugin. In situations where multiple discovery configurations share a particular plugin executing the same functionality, instead of creating multiple instances for that plugin, a single instance of it can be created. Although before making this decision, the discovery service must take multiple factors into account such as what happens in case of sudden failure of the running instance. In such a case, another instance of the plugin needs to created and execution must start from the beginning. A better approach is to cache the execution state of the plugin instance such that in case of a sudden failure, the execution can be resumed from the last executed state.
- *Registry level deduplication:* This deduplication step is initiated by the individual plugins. The requirement is to have a single instance of data in the registry. During this step, the deduplication service checks the outputs generated by the requesting plugin against the existing data instances in the registry. Based on the results obtained from this check, the deduplication service either adds or updates the specific data instance in the registry.

The figure 4.4 depicts an example flow for the execution of the three-level deduplication process. As depicted in the figure 4.4, the configuration level deduplication occurs when the discovery API forwards the received discovery configurations to the Discovery and Registry Management component. After the configuration level deduplication, the discovery service initiates the discovery process through mentioned plugins. The second level of deduplication is performed when the discovery service requests for it at this stage. The final level of deduplication takes place when a plugin, after its execution, requests the deduplication service to store its output on behalf of the plugin.

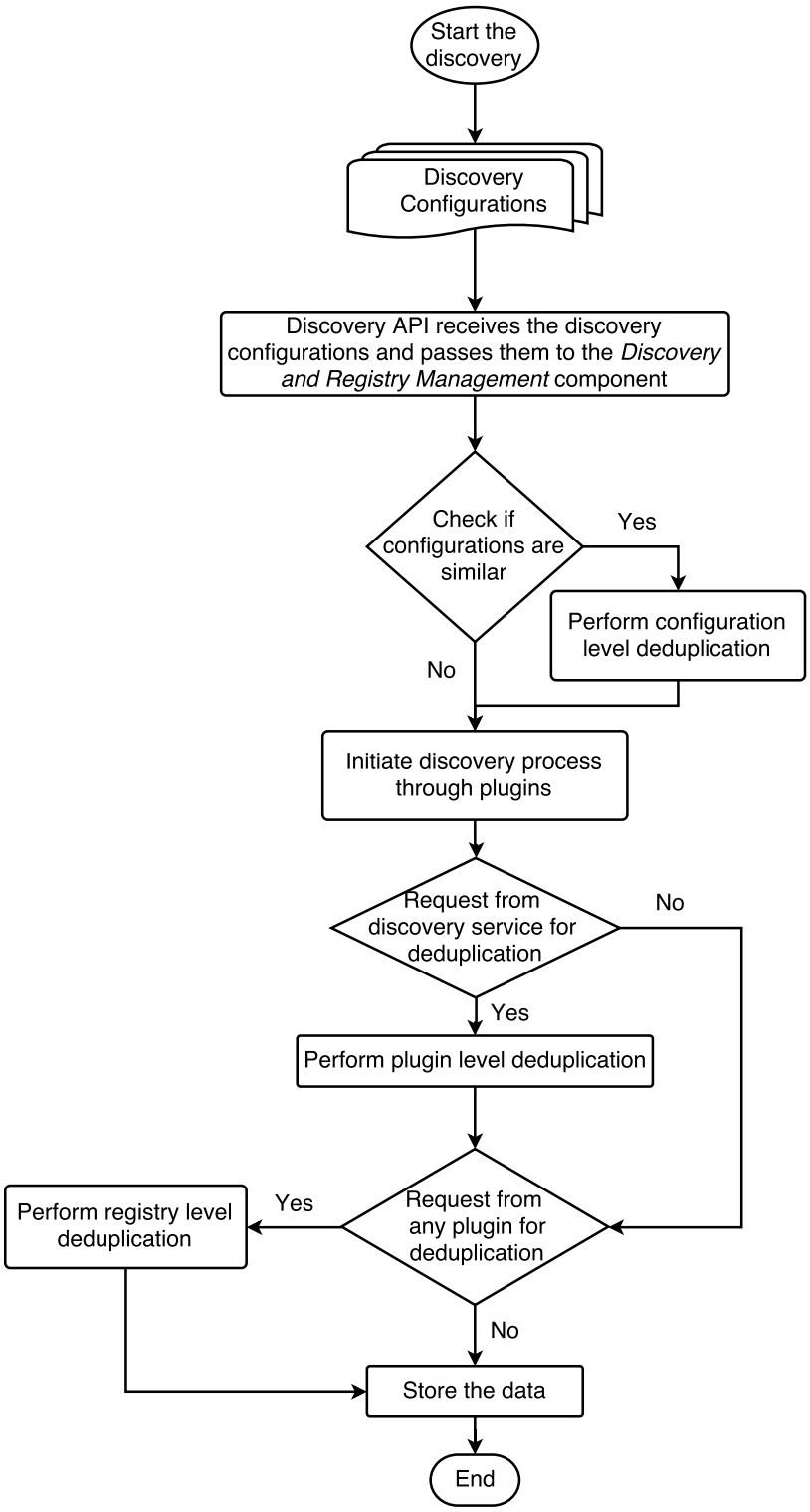

Figure 4.4: An example execution of the three level deduplication process

4.2.4 Discovery API for the Client Applications

In order to cater for the IoT environment, the middleware API supports multiple messaging patterns. In section 5.3.4, the different messaging patterns for the developed prototype are discussed. For all the messaging patterns, the discovery configuration file, as explained earlier, must be sent in the body of the message to initiate the discovery process. For a response it is to be noted that even if no devices are discovered, the discovery is considered to be successful as it indicates no devices are present in the network under observation.

4.3 Summary

In this chapter, an agent-based IoT middleware, working on the principle of pipes and filters design pattern, for the discovery of a device and its services, is proposed. The four main components of the architecture are discussed in detail. The proposed discovery middleware supports both active discovery and passive discovery behavior. In the next chapter, all implementation details regarding the functioning and communication among different plugins are explained.

5 Validation of the Discovery Approach

In chapter 4, we presented our proposed discovery middleware and its components. In this chapter, we validate the proposed approach using a scenario, as explained in section 5.1. Also provided in this chapter, is a mapping of the architecture to the technical implementation. Finally, we discuss about the results and evaluate the overhead for the developed middleware solution.

5.1 Motivating Scenario for Validation

The motivation behind the discovery middleware is to discover the different device and service-specific information using the plugins. Figure 5.1 depicts the motivating scenario and an example for the expected output.

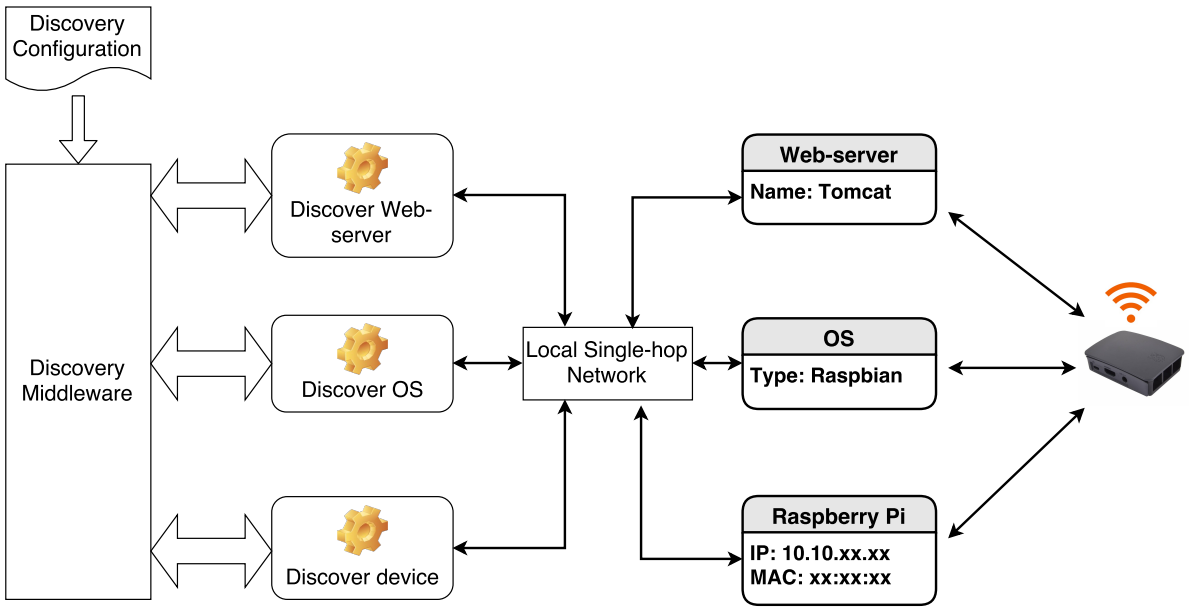

Figure 5.1: Motivating scenario for the validation of the discovery middleware

As shown in figure 5.1, for validation of the proposed discovery middleware, we aim to discover a Raspberry Pi device on the local network. Along with the device, we also aim to discover the OS installed, and the type of web server running on the Raspberry Pi. This scenario demonstrates the pipes and filters design pattern proposed in the discovery middleware, used for discovering the individual components of a device. Before diving into the implementation details, the test environment, set up for the described scenario, is explained in the next section.

5.2 Test Environment

The test environment for validating the middleware solution for proposed scenario includes:

- A Raspberry Pi 3 running Raspbian OS and our localhost running Windows 10. The hardware for the localhost machine includes an Intel Core i5 processor with 8 GB of RAM. On both the devices, we have set up Apache Tomcat Server 9.0.0.M18.
- The discovery middleware is developed in Java and is hosted on the Tomcat Server running on the localhost. The discovery is conducted for the devices in a local single-hop network.
- For communicating with the Request-Response interface of the middleware, the POSTMAN REST client[1] is used, and for interacting with the Publish-Subscribe API, a Google Chrome application namely MQTTLens[2] client is used. For the MQTT case, a public broker from Eclipse Paho project[3] is used, although the implementation is independent of any other public MQTT broker.
- The Nmap [Lyo09] plugin is used for host detection and OS fingerprinting, and the httprint [Sha04] plugin is used for the web server fingerprinting. Both the plugins are installed on the localhost.

[1] https://www.getpostman.com/
[2] https://github.com/sandro-k/MQTTLensChromeApp
[3] https://eclipse.org/paho/

5.3 Mapping of Architecture to Technology

In this section, the four components of the discovery middleware, as detailed in section 4.2, are explained. Firstly, we describe the discovery configuration for the current scenario, where we highlight the attributes for the individual plugins. Next, we present the discovery service, where we explain the working of the plugins in detail, and then, we present the functioning of the deduplication service. Finally, we present the two different APIs for the developed middleware prototype.

5.3.1 Discovery Configuration for the Motivating Scenario

The discovery configuration for the proposed scenario, as explained in section 5.1, is provided in listing 5.1. The discovery configuration provided in listing 5.1 is written in JavaScript Object Notation (JSON) format. The discovery configuration includes two major parts:

- A base plugin for host discovery, and
- Two child plugins for detecting the capabilities of the discovered host.

The base plugin used in this case is a Nmap plugin as represented by the lines 2-9 in listing 5.1. The objective of the Nmap plugin is to discover the network information, the IP and MAC addresses, for all the Raspberry Pi devices present in the local network. In lines 2 and 3 in listing 5.1, the unique identifier and the type of the Nmap plugin are mentioned. As explained in section 4.2.1, the “active” value for the type indicates that Nmap plugin uses active mode of discovery in order to discover the Raspberry Pi devices. Next, lines 4-8 in listing 5.1 lists the different properties for the Nmap plugin under the *pluginConfig* attribute. In line 5 in listing 5.1, the *executionMode* property has a value “endpointDiscovery” which suggests that Nmap wants to discover devices on the local network. Now, in order to discover only Raspberry Pi devices, the Nmap requires an additional property as indicated by *endpointDetails* in line 7 in listing 5.1. The range of IP address to be checked by the Nmap plugin is provided in line 6 in listing 5.1. This range may vary as the network changes. Finally, the list of output produced by the Nmap plugin, which are IP and MAC addresses, is listed in the attribute *pluginOutput* in line 9 in listing 5.1.

Listing 5.1 Discovery configuration for the validation scenario

```
1  [{
2          "pluginName": "nmap",
3          "pluginType": "active",
4          "pluginConfig": {
5                  "executionMode": "endpointDiscovery",
6                  "ipRange": "141.58.50.1-255",
7                  "endpointDetails": "Raspberry"
8          },
9          "pluginOutput": ["host.ip","host.mac"],
10         "nextPlugins":
11         [
12                 {
13                         "pluginName": "nmap",
14                         "pluginType": "active",
15                         "pluginConfig": {
16                                 "executionMode": "osDiscovery"
17                         },
18                         "pluginInput":["host.ip"],
19                         "pluginOutput": ["host.os"]
20                 },
21                 {
22                         "pluginName": "HTTPrint",
23                         "pluginType": "active",
24                         "pluginConfig": {
25                                 "port": "8080"
26                         },
27                         "pluginInput": ["host.ip"],
28                         "pluginOutput": ["host.webServer"]
29                 }
30         ]
31 }]
```

After the discovery of at least one Raspberry Pi device, the next level of child plugins extract the various capabilities for them. These child plugins are listed under attribute *nextPlugins* as indicated on line 10 in listing 5.1 and are as follows:

- *Nmap plugin for OS detection:* Lines 13-19 in listing 5.1 provide the attributes for OS fingerprinting using the Nmap plugin. The *pluginName* and *pluginType* attributes are same as the base Nmap plugin. For OS detection, a value of “osDiscovery” is set for the property *executionMode* in line 16 in listing 5.1. This child plugin requires the IP address of a host for OS fingerprinting. This IP address is generated by the base plugin and included in the *pluginInput* attribute of the child Nmap plugin in line 18 in listing 5.1. Finally, the output of the child plugin, the OS details for the provided host, is mentioned in the *pluginOutput* attribute in line 19 in listing 5.1.

- *Httprint plugin for web server detection:* The attributes for the Httprint plugin is provided in lines 22-28. In lines 22 and 23 in listing 5.1, a unique identifier and the type for the Httprint plugin are mentioned. For detecting the web server running on the Raspberry Pi, the Httprint requires the IP address of the host and the port number. The port number is provided as part of the *pluginConfig* attribute, in line 25 in listing 5.1 and *pluginInput* attribute includes the IP address of the host. Finally, the output for the Httprint plugin is provided in line 28 in listing 5.1.

Using the details discovered by the child plugins, an OS layer, and a web server layer is created on top of the network layer as discovered by the base plugin. A complete list of all the attributes and the functionalities for the discovery configuration is provided in figure 5.2.

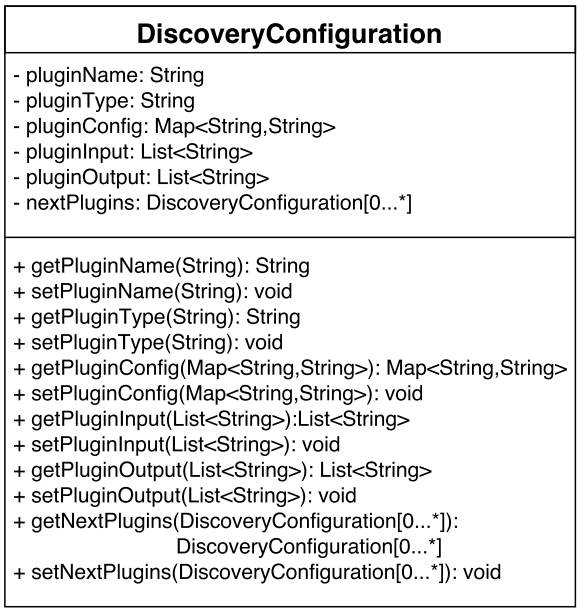

Figure 5.2: Class diagram for DiscoveryConfiguration

5.3.2 Implementation Details for the Discovery Service

The entry point for the discovery service is the *PluginManager* component. After receiving the discovery configuration from the discovery API, the *PluginManager* initiates the discovery process and activates the plugins mentioned in the configuration. Figure 5.3 depicts an excerpt for this component. A simple connection between two classes indicates that they have an interconnection, and can access the other class, e.g., to use the attributes or operations in the desired way.

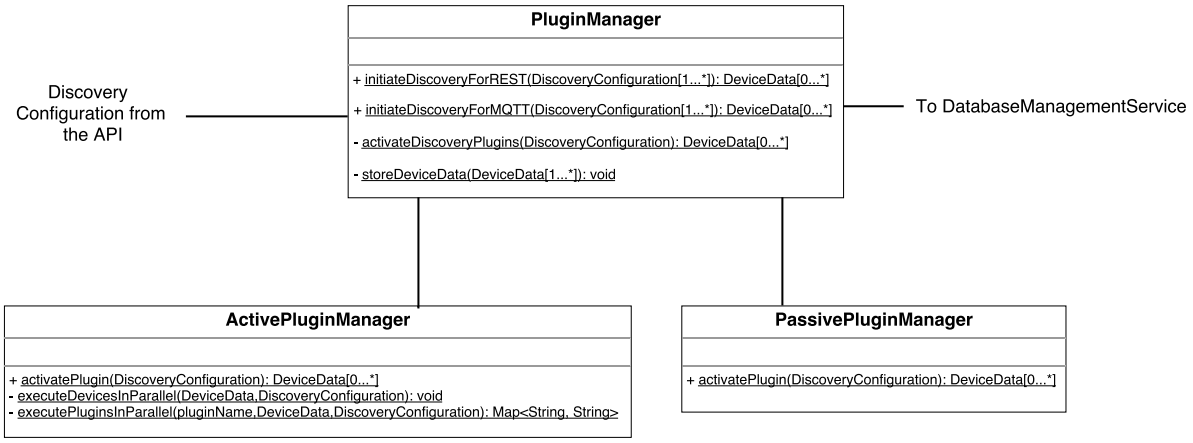

Figure 5.3: Discovery middleware excerpt for PluginManager

The *PluginManager* requires at least one discovery configuration for initiating the discovery process. The *PluginManager* includes separate methods for the two discovery interfaces, namely, *initiateDiscoveryForREST()* and *initiateDiscoveryForMQTT()*, to receive the discovery configuration via the REST interface and the MQTT interface respectively. Details about these interfaces are provided in section 5.3.4.

After parsing the discovery configuration, the *PluginManager* forwards the configuration either to the *ActivePluginManager* or to the *PassivePluginManager* depending on the *type* attribute. The purpose of these two components is to activate the individual plugins as indicated in the configuration. As explained in section 2.3, the *ActivePluginManager* component manages active discovery plugins, and *PassivePluginManager* component manages passive discovery plugins. Additionally, to speed-up the overall execution of active discovery plugins, the *ActivePluginManager* component executes the:

- next level plugins for each of the discovered devices in parallel, and
- individual next level plugins themselves in parallel.

For the validation scenario, initially, Nmap is used for the host discovery. So in this case, *ActivePluginManager* receives the discovery configuration and then activates the individual plugins through *ActivePluginExecutor* component. Figure 5.4 depicts all the plugins used in the validation scenario. In the proposed scenario, we have three functionalities:

- discover the Raspberry Pi host,
- discover the OS installed on the Raspberry Pi host, and
- discover the type of web server running on the Pi host.

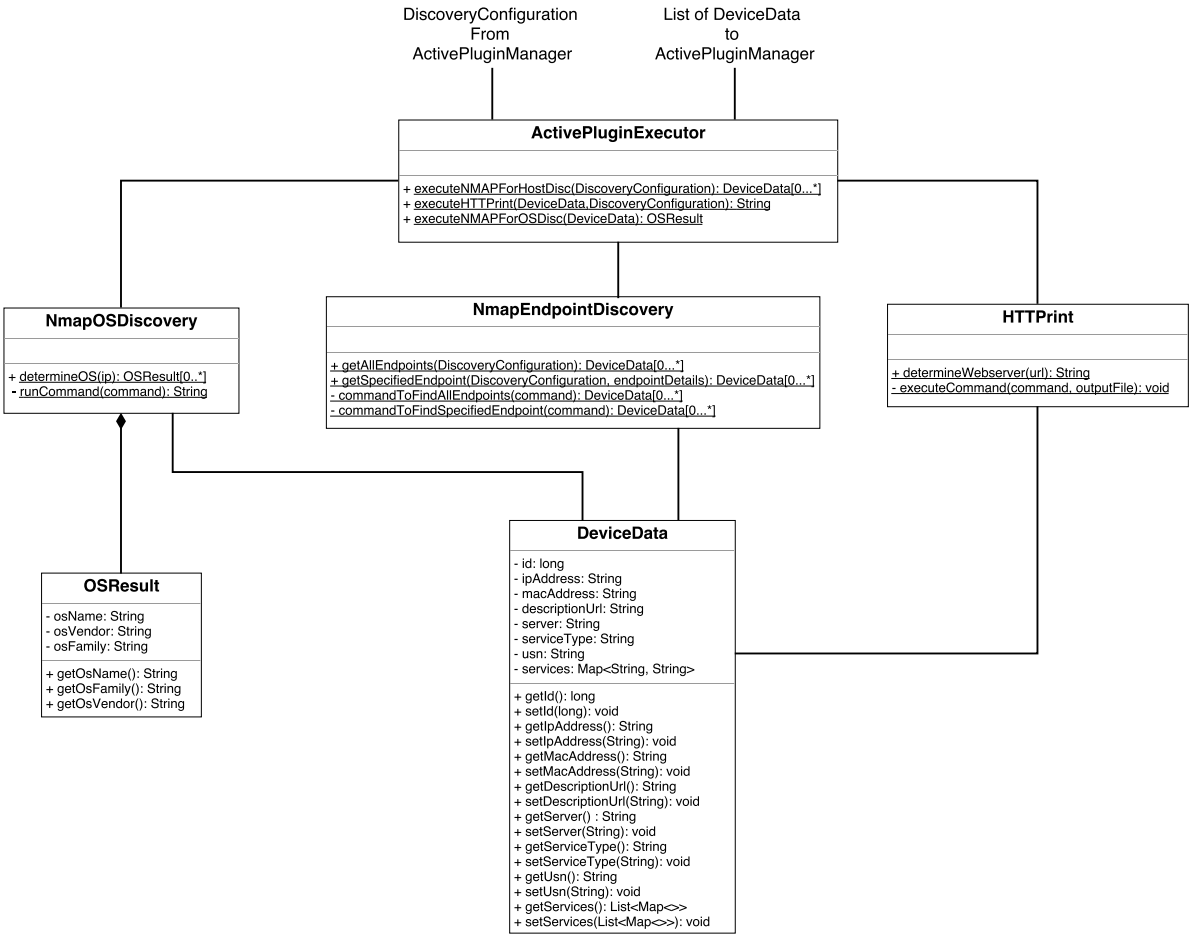

Figure 5.4: Discovery middleware excerpt for ActivePluginExecutor

Implementation details about these functionalities are as given below:

- *Nmap host detection:* For the device or endpoint detection, *NmapEndpointDiscovery* component is used. Nmap uses *PING* scanning technique for detecting the host with flag being “-sn”. This scanning technique finds the targets which are up and does not perform any port scan for them. Also, the timing for the scan can be controlled using flags ranging from “-T0” to “-T5”. We have used “-T4” flag, known as the aggressive scan due to its lower overall scan time without compromising the scan quality. In order to perform the scan, Nmap requires information about its target. The target may be single IP address e.g., *10.0.0.1*, an IP range specification e.g., *10.0.0.1-255*, an IP subnet specification e.g., 10.0.0.0/24, or a set of targets in any of these forms e.g., *172.16.2.0/24 192.168.0.1-29 10.0.0.2*.

For the validation scenario, we discover the Raspberry Pi devices for the provided IP range using a MAC address to Organizationally Unique Identifier (OUI) lookup. OUI is a 24-bit number that is unique for every vendor and forms the first half of a MAC address. For *Raspberry Pi Foundation*, the OUI value is "B8-27-EB".

- *Nmap OS detection:* The OS detection is performed by the *NmapOSDiscovery* component. The Nmap command used by this component includes the "-O" and "–osscan-guess" flags along with the "-T4" flag. Using these flags, an entry for the OS with the closest match to available OS fingerprints is returned. In our implementation, we wrap the returned entry into an *OSResult* object. This object includes information about the OS's name, its family, and its vendor. Also, for performing the OS scan, the IP address of the device is needed as an input. An example command for the OS detection is "nmap -O -T4 --osscan-guess 10.0.0.2".
- *Httprint web server detection:* The web server fingerprinting is performed using the *HTTPrint* component. It requires the IP address and the port number of the host. The concept behind web server fingerprinting is to identify the HTTP servers by their implementation differences in the HTTP protocol. The signature from the web server is checked against a list of existing signatures for different web servers, and then the type of web server is inferred based on its matching percentage with the existing entries. The final output is available in HTML format, CSV format, and XML format. The corresponding flags for the different outputs are "-o", "-oc" and "-ox". For our implementation, we have used the CSV output.

At the end of execution of all the three plugins, a list of device information, which are the instances of the *DevicaData*, is returned back to the *PluginManager* component. This data is passed to the *DatabaseManagementService* which stores it in the database. While storing the data, no check is performed for any duplicate entry in the database. In the next section, we describe how the duplicates in the device information is removed using the *DeduplicationService*.

5.3.3 Implementation Details for the Deduplication Service

Figure 5.5 depicts an excerpt for the Deduplication Service and its functionalities.

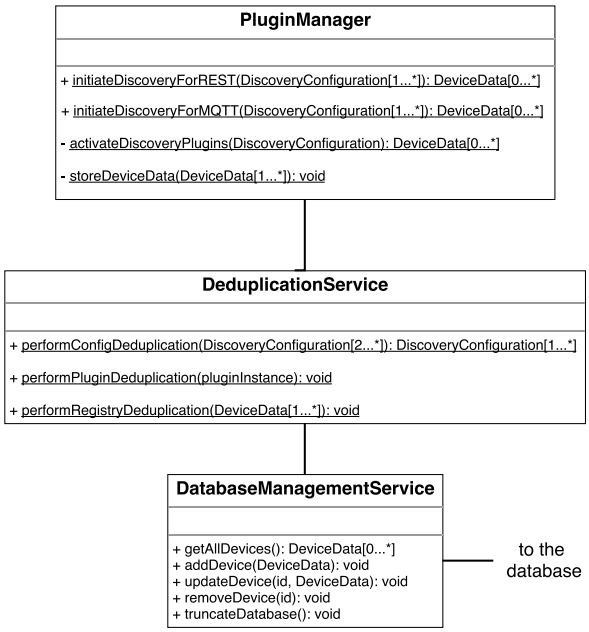

Figure 5.5: Excerpt for Deduplication Service

The *DeduplicationService* service performs three levels of deduplication as explained section 4.2.3. The configuration level deduplication removes duplicate discovery configurations, the plugin level deduplication manages duplicate instances for the same plugin, and the registry level deduplication avoids duplicate entries in the database. As there is only one discovery configuration for the current scenario, configuration level deduplication is not needed. Also, as there are no multiple plugins with the same functionality, plugin level deduplication is not required.

For registry level deduplication, the *PluginManager* component passes the received list of *DeviceData* to the *DeduplicationService*. The method *performRegistryDeduplication()* expects at least one *DeviceData* instance. After receiving the list of *DeviceData*, the method compares each instance against all existing instances in the database. If no match is found, then the received *DeviceData* instance is added to the database, otherwise the corresponding entry in the database is updated with the current *DeviceData* instance.

5.3.4 Implementation Details for the Discovery API

In order to cater to the IoT environment, in addition to the traditional request-response API, the current middleware prototype also supports the publish-subscribe API, which provides a lightweight, easy to implement and low bandwidth system for billions of smart objects [AM15]. The overall discovery process, for a request from an HTTP client, is depicted in figure 5.6.

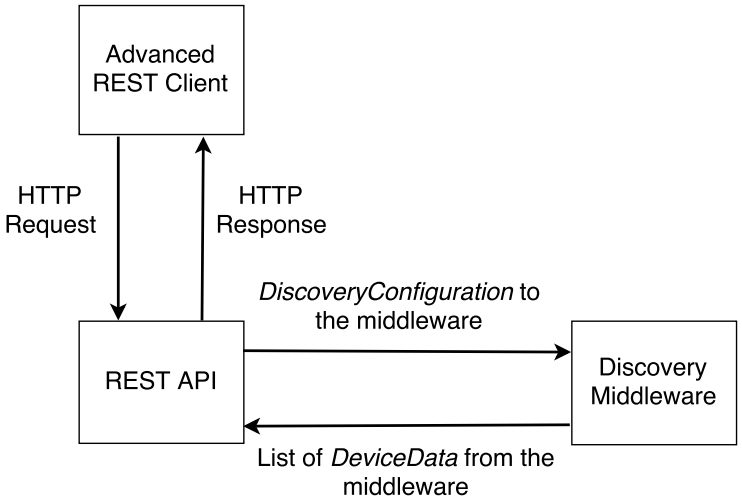

Figure 5.6: REST API for the discovery middleware

We have developed the request-response interface for the middleware using the Representational State Transfer (REST) architectural style [Fie00]. The discovery request is an HTTP POST request with the *DiscoveryConfiguration* as its body. For sending the discovery request we have used the POSTMAN REST client. The URI for the POST request is "*http://Destination-host:8080/DeviceDiscoveryServer/discovery/initiate/REST*".

The response from the REST API includes:

- the list of discovered *DeviceData* instances, and
- a response code for the HTTP request.

For successful operation, a response code of "200" is sent from the API.

The publish-subscribe interface for the discovery middleware is depicted in figure 5.7. In order to communicate with remote MQTT clients, this interface connects to a public MQTT broker. We have used the public broker available at "*tcp://iot.eclipse.org:1883*". For the development of this interface, an Eclipse Java Paho Client is used which is an MQTT client library written in Java for developing applications that run on the JVM or other Java compatible platforms like Android[4].

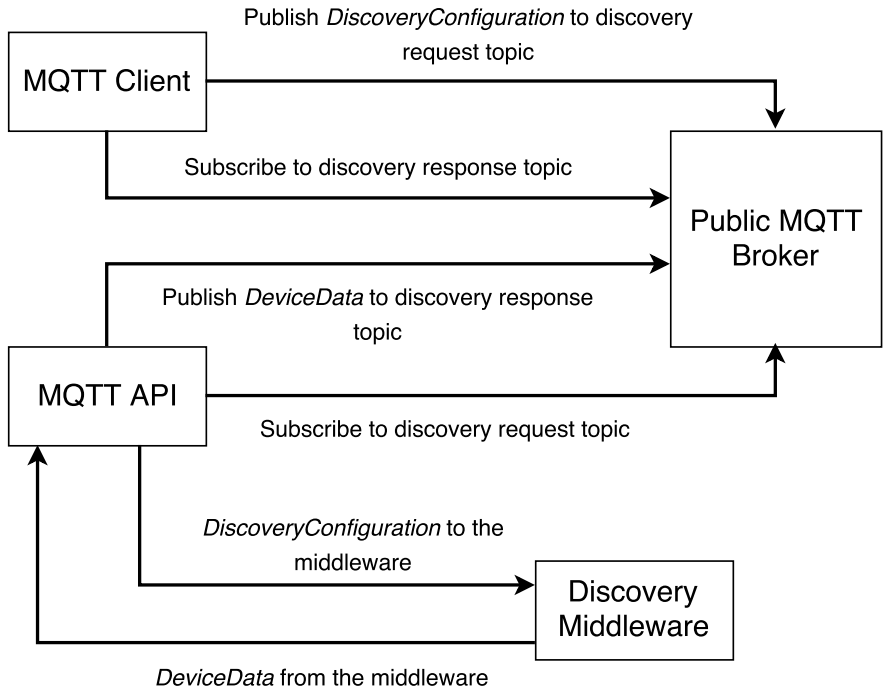

Figure 5.7: MQTT API for the discovery middleware

Initially, the MQTT client publishes the *DiscoveryConfiguration* to the discovery request topic "discovery/request", and subscribes to the discovery response topic "discovery/response" for receiving the desired output from the middleware. The MQTT interface for the middleware connects to the public broker and subscribes to the discovery request topic for receiving the *DiscoveryConfiguration*. After completion of the discovery process, the MQTT interface publishes the list of *DeviceData* to the response topic, and all the subscribed MQTT clients get back the desired list of device information.

[4] https://eclipse.org/paho/clients/java/

5.4 Discussion of the Result

In this section, we discuss the results obtained from the discovery middleware for the proposed used case in section 5.1. The output obtained from the middleware is provided in listing 5.2.

Listing 5.2 Results for the validation scenario

```
1  {
2      "devices": [
3          {
4              "id": 1,
5              "ipAddress": "141.58.50.237",
6              "macAddress": "B8:27:EB:DD:B8:E3",
7              "services": [
8                  {
9                      "Web Server": "Apache-Tomcat/4.1.29"
10                 },
11                 {
12                     "OS Name": "Linux 3.2 - 4.6",
13                     "OS Family": "Linux",
14                     "OS Vendor": "Linux"
15                 }
16             ]
17         }
18     ]
19 }
```

In the listing 5.2, lines 5 and 6 indicate the IP and MAC addresses for the discovered Raspberry Pi device respectively. These values are network dependent. The *id* attribute in line 4 in listing 5.2 represents the database key for the discovered Pi. The attribute *services* include the information about the OS and the web server running on the Pi.

The accuracy of this information depends on the third-party plugins used for the discovery process. As can be seen from the listing 5.2, for the detected web server, the name is correct, although an older version is detected. Similarly, for the OS details, Linux is detected instead of Raspbian. This information can be improved using some other plugins that provide a better match. For example, to improve the OS detection, instead of using Nmap plugin, SSH plugin can be used. The SSH plugin establishes an SSH connection with the Raspberry Pi and obtains the correct value for the OS running on it. In addition to the IP address of the Raspberry Pi, the SSH plugin also requires the username and password information to establish a successful connection.

The current scenario exhibits the configurable nature of the developed discovery middleware. Also, from the results in listing 5.2, we can conclude that the device information is being correctly extracted by the plugins using the proposed pipes and filters design pattern. Therefore the middleware is behaving as expected.

5.5 Evaluation of the Developed Middleware

In this section, we calculate the communication overhead for the discovery process, introduced by our middleware solution running on the localhost. For calculation of the overhead, firstly, we calculate the total execution times for the individual plugins executed sequentially, and then, compare that with the execution time for the middleware solution, for the proposed scenario in section 5.1. All the times mentioned in the following sections is an average for a total of 10 test runs. The calculated run-times are listed in table 5.1.

Overall run-time for sequential execution of individual plugins (in ms)	**Run-time for discovery middleware for an HTTP client (in ms)**	**Run-time for discovery middleware for an MQTT client (in ms)**
18471	19858	20840

Table 5.1: Run-times for individual plugins and discovery middleware

For the sequential execution of the individual plugins, the plugin for host detection must be executed first as the remaining two plugins require the IP address of a host as an input. The average run-time for the sequential execution of the three plugins is 18471 ms, as listed in table 5.1.

To reduce the overhead introduced by the middleware layer, we execute the two next level plugins in parallel, as explained in section 5.3.2. After this enhancement, the average run-time for processing the request from an HTTP client is calculated to be 19858 ms and, thereby, the communication overhead for HTTP scenario is 1387 ms.

For the MQTT scenario, there is an additional 982 ms of overhead than that of the HTTP scenario. This additional time can be explained due to the overhead in communication with the public MQTT broker.

5.6 Summary

This chapter includes all the implementation details for the developed discovery middleware. To enhance the understanding of the technical details, a mapping of the different components of the implementation with the architecture is provided. Finally, an evaluation of the middleware's overhead is done.

6 Conclusion

The objective of this thesis was to develop a configurable middleware solution for discovering devices in the IoT paradigm. The middleware must also identify the capabilities for the discovered IoT devices. For developing the solution, a literature study of existing middleware solutions was performed. In this thesis, various SDPs and network discovery protocols were studied in order to develop the middleware solution. Also, a thorough study on the different enterprise architectural patterns was conducted, and the pipes and filters pattern was selected for its operational advantages for the developed solution.

The discovery middleware developed in this thesis employs a multi-level plugin layer at its core. This plugin layer is responsible for discovering the devices as well as its services. The plugin layer comprises of several independent plugins, each of which aims at discovering the different components of a device. These independent plugins are joined together using the pipes and filters architectural pattern, such that the dependencies for one plugin can be handled by the preceding plugins. For specifying the relations between the different plugins, and for providing the necessary attributes required by a particular plugin, a discovery configuration is developed. This configuration enables the developed middleware solution to be configurable, by specifying the desired discovery behavior in terms of plugins. The developed middleware solution allows three levels of data deduplication, enhancing the run-time behavior as well as the quality of data stored in the database. The clients can avail the discovery services through the discovery API for the middleware. In order to cater to the IoT paradigm, support for the publish-subscribe interface is also provided along with the traditional request-response interface. The middleware solution was validated using a scenario to detect a Raspberry Pi device and different services running on it.

The major advantage of this middleware solution lies in its highly configurable nature. The discovery behavior can be changed by simply rearranging the plugins mentioned in the developed discovery configuration. Also, the developed approach makes it easy for addition and deletion of new features by means of modifying the elements in the discovery configuration. Finally, the developed middleware is generic in the sense that it should be able to detect a range of devices and services by virtue of its configurable plugin layer.

6.1 Further Research

In this thesis, the discovery process is managed by the developed discovery configuration. A possible extension could be to convert this discovery configuration into a workflow model and use it within a workflow engine [LR00], in order to streamline the discovery process and to improve availability, reliability, and scalability of the current approach by using the plugin services within workflows.

The current middleware solution is validated for a local single-hop network. As a typical IoT network spans across multiple networks, another extension for the current work could be to check the feasibility of the discovery process in a dynamic multi-network environment. An existing approach, presented in section 3.1.2, can be studied to realize this extension.

As the current approach discovers device-specific information, a possible extension could be to address vulnerability detection for a device by detecting if the software installed on it is up-to-date or not.

Bibliography

[AHA13] M. Abu-Elkheir, M. Hayajneh, N. A. Ali. "Data Management for the Internet of Things: Design Primitives and Solution." In: *Sensors* 13.11 (2013), pp. 15582–15612. ISSN: 1424-8220. DOI: 10.3390/s131115582. URL: http://www.mdpi.com/1424-8220/13/11/15582 (cit. on p. 16).

[AM15] M. H. Asghar, N. Mohammadzadeh. "Design and simulation of energy efficiency in node based on MQTT protocol in Internet of Things." In: *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. Oct. 2015, pp. 1413–1417. DOI: 10.1109/ICGCIoT.2015.7380689 (cit. on p. 62).

[App13] Apple Inc. *Bonjour Overview*. Apr. 2013. URL: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/NetServices/Introduction.html (cit. on pp. 15, 26).

[Ash09] K. Ashton. "That 'Internet of Things' Thing." In: *RFiD Journal* (June 2009). URL: http://www.rfidjournal.com/articles/view?4986 (cit. on p. 15).

[BBKL13] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "Automated Discovery and Maintenance of Enterprise Topology Graphs." In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. Dec. 2013, pp. 126–134. DOI: 10.1109/SOCA.2013.29 (cit. on pp. 15, 38, 39).

[BFB+11] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, A. Veitch. "Everything as a Service: Powering the New Information Economy." In: *Computer* 44.3 (Mar. 2011), pp. 36–43. ISSN: 0018-9162. DOI: 10.1109/MC.2011.67 (cit. on p. 20).

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. "Formalizing the Cloud through Enterprise Topology Graphs." In: *2012 IEEE Fifth International Conference on Cloud Computing*. June 2012, pp. 742–749. DOI: 10.1109/CLOUD.2012.143 (cit. on p. 38).

[Bre00] E. A. Brewer. "Towards Robust Distributed Systems (Abstract)." In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 7–. ISBN: 1-58113-183-6. DOI: 10.1145/343477.343502. URL: http://doi.acm.org/10.1145/343477.343502 (cit. on p. 21).

[BSMD11] S. Bandyopadhyay, M. Sengupta, S. Maiti, S. Dutta. "A Survey of Middleware for Internet of Things." In: *Recent Trends in Wireless and Mobile Networks: Third International Conferences, WiMo 2011 and CoNeCo 2011, Ankara, Turkey, June 26-28, 2011. Proceedings*. Ed. by A. Özcan, J. Zizka, D. Nagamalai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 288–296. ISBN: 978-3-642-21937-5. DOI: 10.1007/978-3-642-21937-5_27. URL: https://doi.org/10.1007/978-3-642-21937-5_27 (cit. on p. 21).

[CK13] S. Cheshire, M. Krochmal. *Multicast DNS*. RFC 6762. Feb. 2013. DOI: 10.17487/RFC6762. URL: https://www.rfc-editor.org/info/rfc6762 (cit. on p. 27).

[CM12] M. A. Chaqfeh, N. Mohamed. "Challenges in middleware solutions for the internet of things." In: *2012 International Conference on Collaboration Technologies and Systems (CTS)*. May 2012, pp. 21–26. DOI: 10.1109/CTS.2012.6261022 (cit. on p. 36).

[Cur05] E. Curry. "Message-Oriented Middleware." In: *Middleware for Communications*. John Wiley Sons, Ltd, 2005, pp. 1–28. ISBN: 9780470862087. DOI: 10.1002/0470862084.ch1. URL: http://dx.doi.org/10.1002/0470862084.ch1 (cit. on p. 21).

[Don+15] A. Donoho et al. *UPnP Device Architecture 2.0*. UPnP Forum, Feb. 2015. URL: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf (cit. on pp. 15, 23, 33).

[Eva11] D. Evans. "The internet of things how the next evolution of the internet is changing everything." In: *Cisco Internet Business Solutions Group (IBSG)* (Apr. 2011). URL: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (cit. on p. 15).

[FGB11] C. Flores, P. Grace, G. S. Blair. "SeDiM: A Middleware Framework for Interoperable Service Discovery in Heterogeneous Networks." In: *ACM Trans. Auton. Adapt. Syst.* 6.1 (Feb. 2011), 6:1–6:8. ISSN: 1556-4665. DOI: 10.1145/1921641.1921647. URL: https://dl.acm.org/citation.cfm?doid=1921641.1921647 (cit. on pp. 15, 34).

[Fie00] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." AAI9980887. PhD thesis. 2000. ISBN: 0-599-87118-0 (cit. on p. 62).

[FKBT13] M. A. Feki, F. Kawsar, M. Boussard, L. Trappeniers. "The Internet of Things: The Next Technological Revolution." In: *Computer* 46.2 (Feb. 2013), pp. 24–25. ISSN: 0018-9162. DOI: 10.1109/MC.2013.63 (cit. on p. 15).

[GPVD99] E. Guttman, C. Perkins, J. Veizades, M. Day. *Service Location Protocol, Version 2*. RFC 2608. June 1999. DOI: 10.17487/RFC2608. URL: https://www.rfc-editor.org/info/rfc2608 (cit. on pp. 15, 28, 33).

[GS94] D. Garlan, M. Shaw. *An Introduction to Software Architecture*. Tech. rep. Pittsburgh, PA, USA, 1994 (cit. on p. 30).

[GVPK97] E. Guttman, J. Veizades, C. Perkins, S. Kaplan. *Service Location Protocol*. RFC 2165. June 1997. DOI: 10.17487/RFC2165. URL: https://www.rfc-editor.org/info/rfc2165 (cit. on pp. 15, 28, 33).

[HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683 (cit. on p. 31).

[KG99] J. Kempf, E. Guttman. *An API for Service Location*. RFC 2614. June 1999. DOI: 10.17487/RFC2614. URL: https://www.rfc-editor.org/info/rfc2614 (cit. on p. 28).

[KKK+08] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, V. Terziyan. "Smart Semantic Middleware for the Internet of Things." In: *in Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics*. May 2008, pp. 11–15 (cit. on pp. 22, 36, 38).

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0-13-021753-0 (cit. on p. 68).

[Lyo09] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. USA: Insecure, 2009. ISBN: 0979958717, 9780979958717 (cit. on pp. 47, 54).

[Pap03] M. P. Papazoglou. "Service -Oriented Computing: Concepts, Characteristics and Directions." In: *Proceedings of the Fourth International Conference on Web Information Systems Engineering*. WISE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 3–. ISBN: 0-7695-1999-7. URL: http://dl.acm.org/citation.cfm?id=960322.960404 (cit. on p. 22).

[Pos81] J. Postel. *Internet Control Message Protocol*. RFC 792. Sept. 1981. DOI: 10.17487/RFC0792. URL: https://www.rfc-editor.org/info/rfc792 (cit. on p. 29).

[Pre+08] A. Presser et al. *UPnP™Device Architecture 1.1*. UPnP Forum, Oct. 2008. URL: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf (cit. on pp. 15, 23, 33).

[RICL06] P. G. Raverdy, V. Issarny, R. Chibout, A. de La Chapelle. “A Multi-Protocol Approach to Service Discovery and Access in Pervasive Environments.” In: *2006 Third Annual International Conference on Mobile and Ubiquitous Systems: Networking Services*. July 2006, pp. 1–9. DOI: 10.1109/MOBIQ.2006.340448 (cit. on p. 35).

[RKLB09] M. Rambold, H. Kasinger, F. Lautenbacher, B. Bauer. “Towards Autonomic Service Discovery A Survey and Comparison.” In: *2009 IEEE International Conference on Services Computing*. Sept. 2009, pp. 192–201. DOI: 10.1109/SCC.2009.59 (cit. on p. 42).

[RM13] J. Rivera, R. van der Meulen. “Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020.” In: (2013). URL: http://www.gartner.com/newsroom/id/2636073 (cit. on p. 20).

[RMPC16] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, S. Clarke. “Middleware for Internet of Things: A Survey.” In: *IEEE Internet of Things Journal* 3.1 (Feb. 2016), pp. 70–95. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2498900 (cit. on p. 19).

[Sad11] F. Sadri. “Ambient Intelligence: A Survey.” In: *ACM Comput. Surv.* 43.4 (Oct. 2011), 36:1–36:66. ISSN: 0360-0300. DOI: 10.1145/1978802.1978815. URL: http://doi.acm.org/10.1145/1978802.1978815 (cit. on p. 19).

[SBK+16] A. C. F. 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’16. Stuttgart, Germany: ACM, 2016, pp. 181–182. ISBN: 978-1-4503-4814-0. DOI: 10.1145/2991561.2998464. URL: http://doi.acm.org/10.1145/2991561.2998464 (cit. on p. 16).

[SC05] D. Steinberg, S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. O’Reilly Media, Inc., 2005. ISBN: 0596101007 (cit. on p. 26).

[SGFW10] H. Sundmaeker, P. Guillemin, P. Friess, S. Woelfflé. *Vision and Challenges for Realising the Internet of Things*. BU25 02/59 B-1049 Brussels: CERP-IoT - Cluster of European Research Projects on the Internet of Things, 2010. ISBN: 978-92-79-15088-3. DOI: 10.2759/26127 (cit. on pp. 19, 20).

[Sha04] S. Shah. *An Introduction to HTTP fingerprinting*. Net Square Solutions Pvt. Ltd., May 2004. URL: http://www.net-square.com/httprint_paper.html (cit. on p. 54).

[Sil17] N. Silvis-Cividjian. *Pervasive Computing: Engineering Smart Systems*. 1st. Springer Publishing Company, Incorporated, 2017. ISBN: 331951654X, 9783319516547 (cit. on pp. 16, 19).

[TS13] D. Tracey, C. Sreenan. “A Holistic Architecture for the Internet of Things, Sensing Services and Big Data.” In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. May 2013, pp. 546–553. DOI: 10.1109/CCGrid.2013.100 (cit. on p. 20).

[UPnP00] UPnP Forum. *Universal Plug and Play Device Architecture*. June 2000. URL: http://upnp.org/specs/arch/UPnPDA10_20000613.pdf (cit. on pp. 15, 23, 33).

[Wei93] M. Weiser. “Hot topics-ubiquitous computing.” In: *Computer* 26.10 (Oct. 1993), pp. 71–72. ISSN: 0018-9162. DOI: 10.1109/2.237456 (cit. on p. 19).

[ZMN05] F. Zhu, M. W. Mutka, L. M. Ni. “Service discovery in pervasive computing environments.” In: *IEEE Pervasive Computing* 4.4 (Oct. 2005), pp. 81–90. ISSN: 1536-1268. DOI: 10.1109/MPRV.2005.87 (cit. on p. 33).

All links were last followed on October 24, 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