Multicast DNS in wireless ad-hoc networks of low-resource devices

Lu, Y.

Award date:
2012

Technische Universiteit Eindhoven

Department of Mathematics and Computer Science

Multicast DNS in Wireless Ad-hoc Networks of Low-resource Devices

By

Yu Lu

Supervisors

Prof. Dr. J.J. Lukkien
Dr. ir. P.H.F.M.Richard Verhoeven
Dr. Esko Dijk
Dr. P.D.V. van der Stok

November 2011

Contents

1 Introduction **8**
- 1.1 Low-resource Devices . . . 8
- 1.2 Wireless Ad-hoc Network . . . 9
- 1.3 Service Discovery . . . 9
- 1.4 Objective . . . 10
- 1.5 Outline of Thesis . . . 10

2 Background of the Communication Stack **11**
- 2.1 IEEE 802.15.4 Standard . . . 11
  - 2.1.1 Network Topology . . . 12
  - 2.1.2 IEEE 802.15.4 PHY . . . 13
  - 2.1.3 IEEE 802.15.4 MAC . . . 14
  - 2.1.4 Network Joining and PAN Setup . . . 15
- 2.2 Internet Protocol version 6 . . . 17
- 2.3 6LoWPAN . . . 17
  - 2.3.1 6LoWPAN Adaptation Layer . . . 18
  - 2.3.2 Header Compression . . . 18
  - 2.3.3 Header Fragmentation and Reassembly . . . 19
- 2.4 User Datagram Protocol (UDP) . . . 20
- 2.5 Multicast Issues for IEEE 802.15.4 . . . 20

3 Domain Name System **22**
- 3.1 Domain Name System . . . 22
  - 3.1.1 Domain Name Space . . . 22
  - 3.1.2 DNS Name Syntax . . . 23
  - 3.1.3 Resource Records (RRs) . . . 23
  - 3.1.4 DNS Name Server . . . 25
  - 3.1.5 DNS Resolver . . . 27
- 3.2 DNS Operation . . . 27
  - 3.2.1 DNS Query and Response . . . 27
  - 3.2.2 DNS Name Resolution . . . 30
- 3.3 DNS Services . . . 30

4 Multicast DNS **32**
4.1 Multicast DNS Motivations 32
4.2 Multicast DNS Name Space 33
4.3 Multicast DNS Name Resolution 35
4.4 Resource Records (RRs) 36
4.5 mDNS Query 37
4.6 mDNS Response 38
4.7 Multicast DNS Properties 39
4.7.1 Traffic Reduction 39
4.7.2 Source Address Check 40
4.7.3 Multiple Interfaces 40
4.7.4 Multicast DNS Character Set 40
4.8 Multicast DNS and DNS Comparisons 40
4.8.1 Advantage of Multicast DNS 41
4.8.2 Disadvantage of Multicast DNS 41

5 Specification of mDNS Implementation **43**
5.1 Recap for Existing DNS/mDNS Implementations 43
5.2 Design Considerations 44
5.3 Design Goals 45
5.4 Subset of functionalities 45
5.5 mDNS APIs 46
5.6 Elements of Multicast DNS 47
5.7 Implementation of the mDNS Library 48
5.7.1 Transforming Users Request into a Query 49
5.7.2 Sending the Query 50
5.7.3 Processing the Query 51
5.7.4 Processing the Response 52
5.8 Data Structure 53
5.8.1 Message Packet Layout: 54
5.8.2 Database 55
5.8.2.1 Local Database 55
5.8.2.2 Cache 57
5.8.2.3 Cache Look up and Policy 58
5.8.2.4 Internal APIs 59
5.9 Execution Scheme for the Contiki and the mDNS Implementation 60
5.9.1 Event-driven Contiki Kernel 60
5.9.2 Protothreads 60
5.9.3 Running Scheme for Multicast DNS on Contiki 61
5.9.3.1 Process for Multicast DNS 61
5.9.3.2 Process for the User Application 62

6 Conclusion **64**

A Contiki OS **66**
A.1 Communication Stack 66
A.1.1 µIPv6 67
A.1.2 Rime 68
A.1.3 Contiki directory structure 69

B CC2530EM **70**

C IAR Workbench **71**

D SmartRF05EB **72**

E Test Setup **73**

Nomenclature **74**

Bibliography **77**

List of Figures

1.1 Example of Building Automation 9

2.1 Communication Stack 12
2.2 IEEE 802.15.4 Protocol Stack 13
2.3 MAC data service 15
2.4 The procedure to associate a device with a PAN coordinator . . 16

3.1 Domain Name Space for DNS 23
3.2 DNS Name Structure 24
3.3 RR Format 25
3.4 RRs Example of a Domain 26
3.5 DNS Packet Layout 27
3.6 Header Section 28
3.7 Question Section 28
3.8 Example of DNS Response 29
3.9 Name Resolution Example 30

4.1 Multicast DNS Name Structure 34
4.2 Hierarchical Name Structure for Multicast DNS 34
4.3 Wireless Ad-hoc Network Example 35
4.4 Message Sequence Chart for Startup 37

5.1 mDNS Library 48
5.2 The Communication between User Program and Resolver 49
5.3 Resolver Sends the mDNS Query 50
5.4 The Continuous Query 51
5.5 Processing the Query 52
5.6 Processing the Unsolicited Response 53
5.7 Header Section for Query 54
5.8 Header Section for Response 54
5.9 Design of a Local Database 55
5.10 Implementation for Local Database 56
5.11 Data Structure for Local Database 56
5.12 Data Structure for Cache 57

5.13 Example of a Cache 57
5.14 Cache Look up and Policy 58
5.15 mDNS Implementation Structure 61

A.1 µIP and Rime 67
A.2 uIPv6 stack 67
A.3 Overall Organization of Rime Stack 68

B.1 CC2530EM 70

D.1 SmartRF05EB 72

E.1 Test for evaluation setup for the mDNS design 73

List of Tables

2.1 Explanation for Dispatch Value 19

3.1 Common RRs 25

4.1 Comparison between Multicast DNS and DNS 41

5.1 Size of Existing DNS/mDNS Implementations on Linux 44

Abstract

Multicast DNS (mDNS) is the technology used to replace the conventional DNS server in a small area network. As an extension of DNS with the backwards compatibility, Multicast DNS has its own advantages in certain situations. The existing mDNS implementations, such as Bonjour and Avahi, are not suitable to be used in low-resource devices. Thus, this work aims at designing an mDNS implementation blueprint, to fit the characteristics of embedded devices. The CC2530 and Contiki OS are taken as the target platform during the design. Wireless Ad-hoc is the network environment to be considered for the design.

Chapter 1

Introduction

With the quick development of small, low-resource devices in our daily lives, a promising idea is to connect all these small devices using the Internet technology. In that case, every device is able to communicate with any other devices anywhere. Multicast DNS (mDNS)[1] and its companion technology DNS-Based Service Discovery (DNS-SD)[4], are considered as the important start points to achieve this goal. The DNS-SD enables the establishment of application collaboration in an ad-hoc network. This project focuses on the design of an implementation of the mDNS protocol for ad-hoc networks consisting of low-resource devices.

1.1 Low-resource Devices

Nowadays, low-capacity embedded devices, such as those used in building automation and smart grids, have the potential to change our life. Compared with the traditional high-resource devices, these embedded devices have low computational and communication capacities. The memory capacity of low-resource devices is also limited.

In the last decade, the Internet has been developing at an unimaginable speed. It is more convenient if there were Internet services everywhere. Thus, the IP-based low-resource device is important. Users could control the low-resource devices, which together form the building automation, from remote locations using computers or cellphones. With a low-resource device inside every product, mislaid items and physical theft could be avoided because the location of an item could be detected at all times. Figure 1.1 gives an example of building automation. With the low-resource device inside the building facilities, such as the air conditioner, the elevator and the fire alarm, the central server is able to control the whole building.

In this work, the low-resource device supports the IEEE 802.15.4[20] standard and provides the 6LoWPAN[10] and IPv6[19] layers. All these concepts are discussed in Chapter 2.

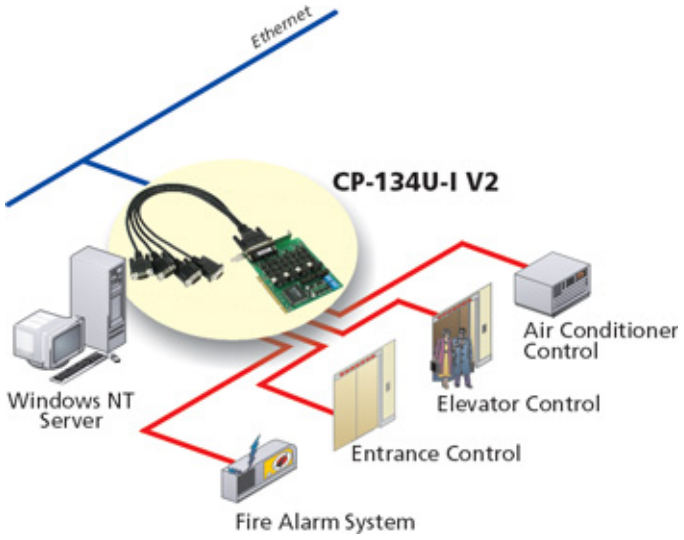


Figure 1.1: Example of Building Automation

1.2 Wireless Ad-hoc Network

For these low-resource devices, communication is often wireless, using low-power and, hence, low-capacity radio technologies and unmanaged networks, called wireless ad-hoc networks. A wireless ad-hoc network is a decentralized wireless network. Each node of an ad-hoc network behaves as an end-point as well as a router. By contrast, a centralized network is a type of network where all users connect to a central server, which is the acting agent for all communications. In principle, an ad-hoc network provides more flexibility and scalability than a centralized network at the expense of more elaborate behavior of the individual nodes. The ad-hoc network is typically applied within a small area and under simple conditions. As the central server equipment and the corresponding setup are not needed in the system, it saves on hardware and labor costs.

1.3 Service Discovery

In traditional wireless managed networks, each device is configured with the information, such as its domain name, IP address, domain name server address (DNS server) and address of the local router. Using the DNS[3] system, nodes can look up addresses of nodes supplying services, such as the email service, the HTTP[31] service or any other services. In the ad-hoc network no such system is installed, hence nodes do not know where to find such information. In order to solve this problem, the concept of ad-hoc service discovery is introduced for these low-resource devices. It is an important addition to ad-hoc networks since it enables sharing of functionality and resources and makes it possible

to run distributed applications without the need for manual installation and maintenance of a service database[10].

Current service discovery technologies including Zero Configuration Networking, Universal Plug and Play, Jini and JXTA[12] are aimed mostly at high-resource devices. In this work, Multicast DNS, as the base for DNS-Based Service Discovery (DNS-SD), is the main technology to be discussed. The core of this project is to propose the design for implementing Multicast DNS on low-resource devices.

1.4 Objective

This project is performed in Philips Research. Recently, Philips Lighting is aiming at the IP-based lighting control system with more intelligence. In order to build the intelligent lighting system, a practical and fast approach to organize the system, which is the mDNS service, is essential. Applying the mDNS service on low-resource devices is especially worthwhile to be investigated.

This thesis project has the following objectives:

- Survey the state-of-the-art in Multicast DNS. Investigate the potential applications of the mDNS service. Demonstrate the value of knowledge and the practical significance.
- Develop the mDNS specification for a reduced version which can realize the basic mDNS process, with emphasis on the design of data structures and the memory usage.

1.5 Outline of Thesis

The remainder of this report is structured as follows. Chapter 2 gives the introduction to the example communication stack used in this work, including the IEEE 802.15.4 standard, IPv6, 6LoWPAN and UDP. The properties of these technologies may affect the design, which is discussed in the work. Chapter 3 introduces the basic concepts of conventional DNS. Chapter 4 gives the Multicast DNS introduction and the comparison between the conventional DNS and Multicast DNS. Chapter 5 describes the detailed specification of the mDNS implementation on low-resource devices. Chapter 6 concludes the work within this thesis.

Chapter 2

Background of the Communication Stack

The focus of this work is how the mDNS service is implemented for an ad-hoc network which consists of multiple low-resource devices. As mentioned in Chapter 1, these low-resource devices support the IEEE 802.15.4[20] standard and provide the IPv6[19] and 6LoWPAN[10] layers. They have low computational and communication capacities, and the available memory is small. In this chapter, the concepts of IEEE 802.15.4, IPv6, 6LoWPAN and UDP[22] are introduced and they determine the conditions for the design of the mDNS service implementation. Figure 2.1 shows the communication stack used in this thesis. The lowest layers of the stack are based on the IEEE 802.15.4 standard, which are described in section 2.1. The IPv6 layer as well as the 6LoWPAN layer lies in the middle of the communication stack, which are described in section 2.2. and 2.3, respectively. The UDP layer is located on top of the IPv6 layer and it is introduced in section 2.4. In section 2.5, a number of implications are listed to indicate how multicast traffic is affected by this stack. The concept of Multicast DNS and the related knowledge are discussed in the Chapter 4.

2.1 IEEE 802.15.4 Standard

Since the personal wireless devices and wireless equipments are manufactured by different vendors, it is particularly important to follow a uniform protocol or standard for the wireless communication of these devices. In addition, these wireless devices are typical of low-power and low-resource. For these concerns, the IEEE 802.15 working group was established to develop a standard for the short-range wireless communication. The goal of the IEEE 802.15.4 working group is developing a low data-rate standard especially for low power embedded devices with a limited resource budget.

The IEEE 802.15.4 standard specifies the physical layer (PHY) and the media access control (MAC) for low-rate wireless personal area networks. Figure

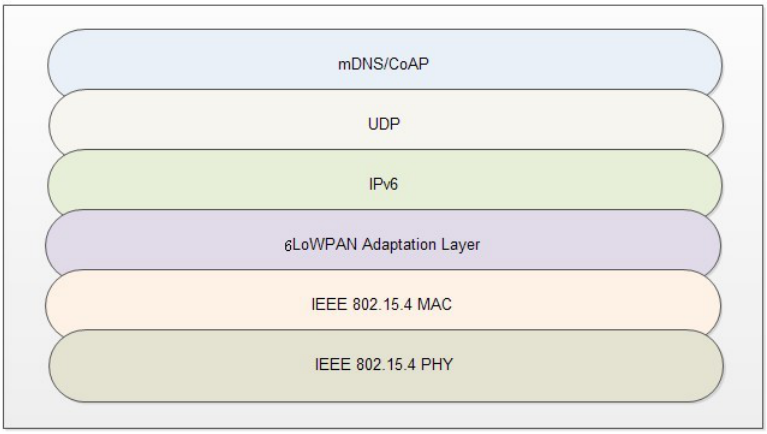


Figure 2.1: Communication Stack

2.2 shows the protocol stack for IEEE 802.15.4. The Logical link control (LLC) layer functions as the interface between the MAC layer and the Network layer in the OSI[28] model. It provides multiplexing mechanisms with which different network protocols (IP, IPX, Decnet and Appletalk) can coexist within a multipoint network. Meanwhile, the corresponding data can be transported over the same network media. The convergence sublayer in the figure is used for the transport of all packet- based protocols, such as Internet Protocol (IP)[21].

With the services provided by the standard and the corresponding higher layer protocol, the IEEE 802.15.4 device is able to setup the wireless personal area network (PAN).

2.1.1 Network Topology

The PAN is used for interconnecting devices around a person's workspace. Thus, the communication range for the PAN is limited to a few meters. Different network topologies are available for the PAN, in which the star topology and the peer-to-peer topology are widely in use. The IEEE 802.15.4 standard supports both network topologies.

- Star topology: The star topology is a network that consists of a central controller which is named as PAN coordinator, and other end devices. A PAN coordinator initializes, terminates and routes the messages in the PAN. It is the most important component in the star-topology-based network. The end devices communicate with each other by sending their messages to the PAN coordinator. Then the PAN coordinator would forward the messages to the destinated end devices.

- Peer-to-peer topology: In the network based on the peer-to-peer topology, each device communicates with other devices directly. It is not necessary

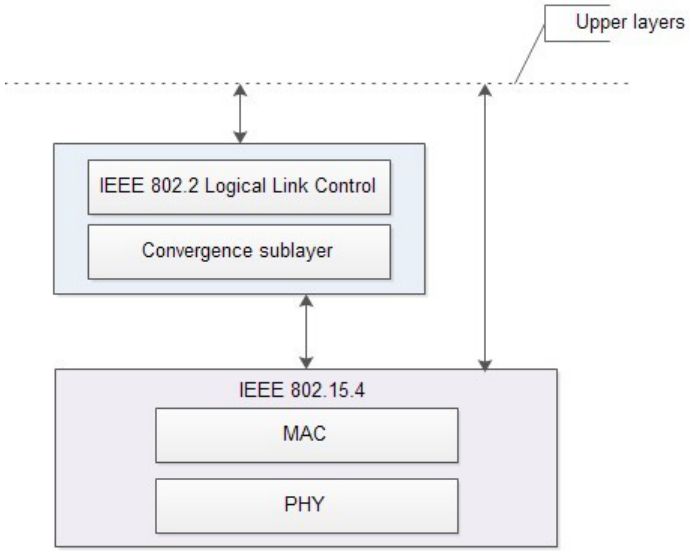


Figure 2.2: IEEE 802.15.4 Protocol Stack

for the PAN coordinator to forward the message. However, the PAN coordinator is still presented in the Peer-to-peer topology. When the device requires to communicate with the node which is outside its immediately radio range (that might happens in the wireless network), a routing scheme would be provided by the PAN coordinator.

In section 2.1.4, the details about setting up a PAN are introduced.

2.1.2 IEEE 802.15.4 PHY

The IEEE 802.15.4 PHY provides the interface for the physical radio channel and the IEEE 802.15.4 MAC layer. By applying the direct sequence spread spectrum (DSSS)[33] technique, the PHY layer reduces the cost of the digital integrated circuit. The strict power management also results in the low operating cycles as well as the low power operation. In general, PHY selects the channel and performs the energy/signal management. A list of tasks for IEEE 802.15.4 PHY is shown below[20]:

- Activation and deactivation of the radio transceiver.
- Energy Detection (ED) within the current channel.
- Link quality index (LIQ) for the received packet.
- Channel clear assessment (CCA) for SCAM-CA.
- Channel frequency selection.

- Data transmission and reception.

There are three optional frequency bands for IEEE 802.15.4 PHY:

- 868.0-868.6 MHz
- 902-928 MHz
- 2400-2483.5 MHz

2.1.3 IEEE 802.15.4 MAC

IEEE 802.15.4 MAC handles all accesses to the physical radio channel. It is responsible for the following tasks[11]:

- Generate network beacons if the device is a PAN coordinator.
- Synchronize to beacons.
- Support the PAN association and disassociation.
- Handle and maintain the guaranteed time slot (GT) mechanism.

Generally speaking, 802.15.4 MAC provides two types of services: the MAC data service and the MAC management service.

MAC Data Service The most important MAC data service is transferring the data to the neighboring devices:

- Data Request: It is a transmit function. When this function is called, the data is taken from the high layer and put into the outgoing FIFO of the radio module.
- Data Confirm: Once the data is sent out, the sending radio gives some indications of the status to the MAC. The status indicates whether the data is successfully transmitted or not.
- Data Indication: When the data from other devices arrives at the FIFO of the radio module, it will trigger an interrupt. The MAC is signaled that the data has arrived and is ready for processing.

Figure 2.3 shows the message sequence chart between two 802.15.4 nodes using the MAC data service:

Besides the data transmission, the MAC data service controls the maximum data length for the IEEE 802.15.4 standard. aMaxMACFrameSize[34] is 102 bytes.

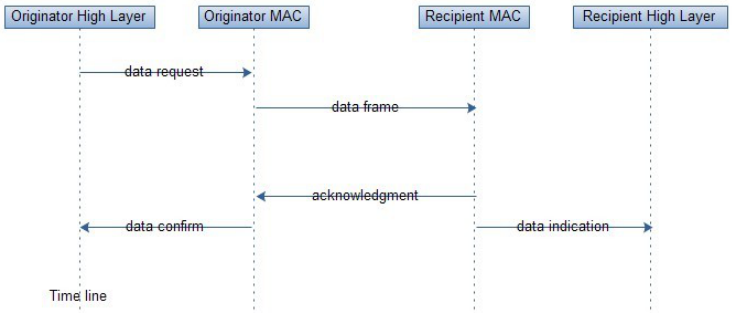


Figure 2.3: MAC data service

MAC Management Service The MAC management service is also called MAC Layer Management Entity (MLME). It is responsible for providing the service interfaces through which the layer management functions can be called. The main services provided by the MAC management are[11]:

- Association/disassociation
- Node notification
- Network scanning/start
- Network synchronization/search

All these management services play an important role when an 802.15.4 device is connected to the existing network or establishing a new network.

2.1.4 Network Joining and PAN Setup

The association is a process in which the device joins the network. The MAC provides the association procedure for the network layer. The network layer manages the device joining the network. IEEE 802.15.4 provides four API calls for the association procedure.

- Association request: This service is used by the network layer when the device requests to join a PAN coordinator.
- Association indication: Once the MAC layer of a PAN coordinator receives the association request from another node, it uses the association indication to inform its network layer about this request.
- Association response: Once the association request from another node is received, the network layer uses association response to inform its own MAC layer about the decision.

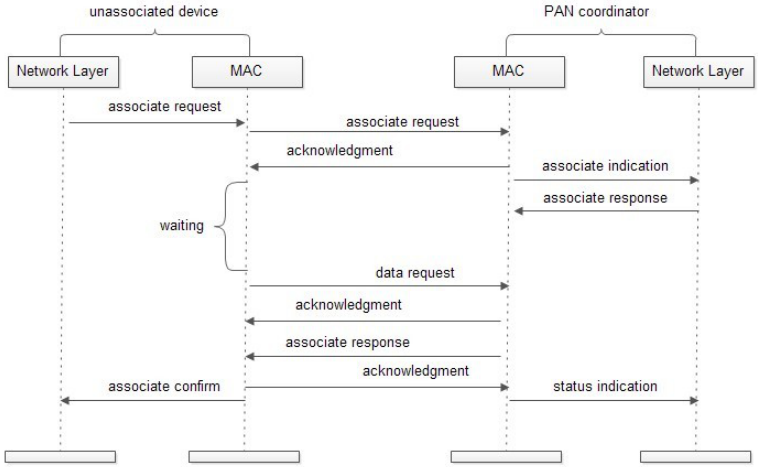


Figure 2.4: The procedure to associate a device with a PAN coordinator

- Association confirmation: In the requesting node, the MAC layer employes the association confirmation to inform the network layer about the decision of the PAN coordinator

With the four API calls provided by the MAC, the 802.15.4 device is able to setup a PAN. Figure 2.4 gives a general procedure that a device follows to associate with a PAN coordinator.

Before the association, the node should implement the scan process. In this process, the node searches for all available PANs as well as the corresponding channels within its reach. It can be done by either the active or passive scan. During the passive scan process, the device turns on its receiver and listens to each channel in a defined period of time. After scanning all the request channels in order, the MAC layer informs the Network layer with all discovered PANs. The active scan is similar to the passive scan, with the addition that a beacon request frame is sent by the node during the scanning process. The node will wait for a beacon frame of the coordinator in a defined period of time. Hence, the active scan is used for non beacon enabled PANs. When a PAN coordinator in a non-beacon enabled PAN receives a beacon request, it will reply with a single beacon frame.

Once the node scans and finds the PAN coordinator, it begins to associate. The higher layer of the node sends the association request to the MAC layer of the same node. The request is transmitted from the MAC layer to the PAN coordinator. Once the MAC layer of the PAN coordinator receives the request, it informs its higher layer with the association indication. The higher layer makes the decision whether to associate that requesting node or not. The decision would be given to the MAC layer by the association response. Afterwards the MAC layer of the PAN coordinator would relay the response to the requesting

node. When the MAC layer of the requesting node receives the response, it sends an associate confirmation to its higher layer.

2.2 Internet Protocol version 6

IEEE 802.15.4 provides the PHY and the MAC layer which are described in the 7-layer OSI model[28]. A variety of host protocols can be implemented on the top of IEEE 802.15.4, among which the Internet Protocol version 6 (IPv6) is one of the promising choices. In this work, IPv6 is recommended to be used, because the number of IP devices will exceed the capacity of IPv4 when wireless sensor networks are connected based on IP technology.

IPv6 is a version of the Internet Protocol (IP) and is responsible for routing packets delivery including the routing through intermediate routers[19]. It provides the means of transferring data sequences from the source to the destination via one or more networks. As a successor of IPv4, IPv6 redefines the address size, the data format and also the address assignment. Moreover, the network security is integrated into IPv6. Some differences between IPv4 and IPv6 are:

- Simplified header format: The length of IPv6 header is fixed. It does not contain as many options as IPv4 header. Although the IPv6 address is 128-bit for source and also destination, the total length for IPv6 header is only 40 bytes. Smaller size allows faster processing. If any option is needed by the IPv6 packet, it will be included in the IPv6 extension header.
- Extended address: The longer IPv6 address format ensures that there will be sufficient IP addresses all over the world. The address space in IPv6 could support 2^{128} addresses in theory. In addition, the extended address allows the hierarchical structure of the address space.
- More functionalities: Based on ICMPv6[35], many new functionalities are added into IPv6, e.g. Neighbor Discovery, Autoconfiguration and Multicast Listener Discovery.

2.3 6LoWPAN

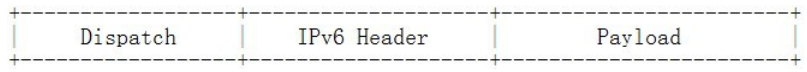
In order to implement IPv6 on top of IEEE 802.15.4, going through the 6LoWPAN adaptation layer is the only solution. In this section, 6LoWPAN is introduced. We mainly focus on how the normal IPv6 packet is compressed by the 6LoWPAN layer to enable IPv6 transport on IEEE 802.15.4 devices.

IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) is a set of standards defined by the Internet Engineering Task Force (IETF)[18], which creates and maintains all core Internet standards and architecture work. It was developed to enable the wireless embedded network and devices to use simplified IPv6 functionalities, which take into account the limitations of embedded devices, such as low power, small memory and limited bandwidth[10].

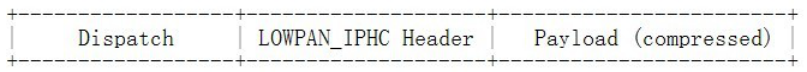
2.3.1 6LoWPAN Adaptation Layer

According to the IEEE 802.15.4 standard, the maximum size of the packet on the physical layer is 127 bytes and the maximum frame overhead is 25 bytes. Only 102 bytes of the packet space are available for the payload of the MAC layer. This is much smaller than the standard size (1280 bytes) of the IPv6 maximum transmission unit (MTU). The best case compression of a link-local multicast UDP/IPv6 header has the size of 23 bytes[32]. That is to say, there are 79 bytes space available for the mDNS packet in an ideal situation. Normally, an mDNS packet has the size of 100 to 200 bytes. Thus, the compression and fragmentation are necessary in most cases. The 6LoWPAN adaptation layer is defined to compresses the IPv6 header and the following headers such as UDP and does the fragmentation if necessary. After the processing by the 6LoWPAN adaptation layer, the IPv6 datagram is encapsulated. The 6LoWPAN adaptation layer is shown in Figure 2.1 and lies between layer 2 and layer 3.

For every encapsulated IPv6 datagram packet, there is always an encapsulation header prefix at the beginning. The header starts with a dispatch value which indicates the type of the 6LoWPAN datagram header. The 6LoWPAN payload follows this header, which includes the IPv6 header and its payload. The figure below shows the structure:



Since 6LoWPAN does the header compression, LOWPAN_IPHC compressed IPv6 datagram is shown below:



There are three types of headers indicated by the dispatch value: the mesh addressing header, the broadcast header and the fragmentation header. The dispatch values as well as the corresponding header types are listed in Table 2.1(x in the table means the value of the bit could be either 0 or 1):

2.3.2 Header Compression

6LoWPAN does stateless header compression for the IPv6 header[14]. Since the compression is stateless, any node on which the 6LoWPAN layer is implemented can decompress the header.

Until now, the header compression is mainly applied on the IPv6 header and the UDP header. The term LOWPAN_IPHC is used to indicate the IPv6 encoding and LOWPAN_NHC is used to indicate the UDP encoding.

Dispatch Value	Header Type
00 xxx xxx	Not a LoWPAN frame
01 1xxxxx	LOWPAN_IPHC compressed IPv6
01 000001	Uncompressed IPv6 address
01 000010	LOWPAN_HC1 compressed IPv6
01 010000	LOWPAN_BC0 broadcast
01 111111	Additional dispatch byte follows
10 xxx xxx	Mesh header
11 000xxx	Fragmentation header (first)
11 100xxx	Fragmentation header (subsequent)

Table 2.1: Explanation for Dispatch Value

- LOWPAN_IPHC: The 40-byte IPv6 header can be compressed into 7 bytes by the LOWPAN_IPHC compression scheme. For the 64-bit link-local IPv6 address prefix[13], it can be omitted. For the last 64-bit, it can be inferred from the link-layer address in case the interface identifiers are autoconfigured. Thus, only a 16-bit native short address of the device itself is necessary for the IPv6 address field in the 6LoWPAN compressed header. The packet length can be inferred from the MAC layer, so it can also be omitted. For all the considerations as above, the encoding structure based on LOWPAN_IPHC is as follows:

  ```
  +-------------------------+---------------------------------+
  | LOWPAN_IPHC encoding    |  Non-Compressed fields follow   |
  +-------------------------+---------------------------------+
  ```

  The first segmentation "LOWPAN_IPHC encoding", which takes up to 2 bytes, shows how the compression is done and what the data are after the header. The Non-compressed field gives the data that cannot be ignored. They are indicated by the LOWPAN_IPHC.

- LOWPAN_NHC: LOWPAN_NHC gives the compression scheme for the UDP header. The structure of LOWPAN_NHC part is as follows:

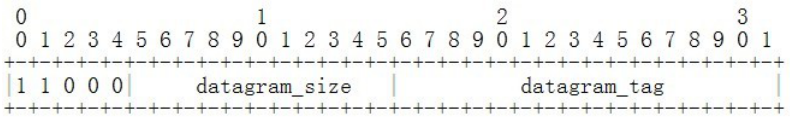
  ```
  +---+---+---+---+---+---+---+---+
  | 1 | 1 | 1 | 1 | 1 | 0 | S | D |
  +---+---+---+---+---+---+---+---+
  ```

  Bits 0-5 are used to indicate the type of the header. The structure above is for the UDP header compression. Bits 6-7 are used to set different options. It is noticed that LOWPAN_NHC could also be used by other layer protocols in the future.

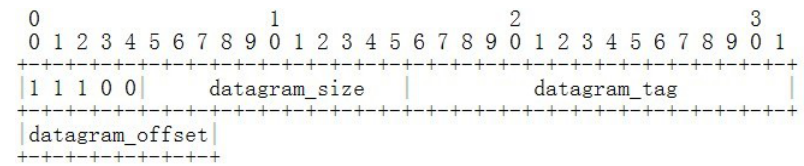
2.3.3 Header Fragmentation and Reassembly

The fragmentation is not compulsory in the 6LoWPAN data processing. If the entire datagram payload fits in the 802.15.4 MTU, fragmentation is not applied. Otherwise the fragmentation is required. The fragmentation is executed by breaking down the payload into multiple fragments. As shown below, there are two types of fragmentation packets. The top one is the header at the first

fragment:



The remaining fragments have a header with one extra 1-byte offset value:



As given in Table 2.1, the first 5-bit of the dispatch value indicates that the two figures above are the structures of the fragmentation headers. The component "datagram_size" indicates the total size of the IPv6 datagram before the fragmentation. The component "datagram_tag" gives the flag to a certain datagram in the sending queue. It is common that different IPv6 datagrams are waiting in the sending queue. This flag helps to identify the same datagram before the fragmentation. Datagram_offset indicates the position of a fragment in the original IPv6 datagram. With all these values, the receiver is able to reassemble the fragmented IPv6 datagram.

2.4 User Datagram Protocol (UDP)

UDP[22] is a network protocol used for the Internet. The programs on the application level can employ UDP to transmit the datagram in the Internet Protocol network without prior setting up either the communication channel or data paths.

Since there are not any implicit handshaking dialogues which provide the reliability and safety, the UDP communication is an unreliable service which may result in the datagrams either arriving out-of-order or missing without notifications. However, UDP avoids the network overhead caused by the error checking and acknowledgments. Thus, the UDP communication is useful for a server answering small queries from a large number of clients. Meanwhile, the property that the UDP communication does not require acknowledgments makes it suitable for use in real-time systems.

There are several network applications using the UDP protocol such as the Domain Name System (DNS) and streaming media applications (e.g., IPTV and VoIP). In this work, the mDNS protocol is also based on UDP communication.

2.5 Multicast Issues for IEEE 802.15.4

In computer networks, multicast communication is the way to send datagrams to a group of specified receivers with a single transmission. Normally, the computer

listens to a multicast IP address and the corresponding port. Once the datagram is received from the network, the hardware is able to filter the non-interested MAC addresses and keep the required one. However, the IEEE 802.15.4 device, such as the target platform CC2530, does not support multicast directly. Thus, in this work, we suggest to use other transmission methods to replace the traditional multicast.

There are two solutions to implement multicast in the IEEE 802.15.4 environment. Since broadcast communication is supported by the IEEE 802.15.4, it is feasible to replace multicast with broadcast. When the datagram size is small enough to fit within a single unfragmented 6LoWPAN packet, it is possible to use broadcast communication without the problems of missing fragments. However, since there is no MAC-level acknowledgment sent back for broadcast transmissions, the transmission is less reliable compared to unicast transmissions. This may cause serious problems when the fragmentation of 6LoWPAN is applied to the datagram. Once the data is fragmented, a single lost fragment will result in the invalidation of the entire datagram. In this case, a relative rational approach is to employ multiple unicast transmissions. Multiple unicast also brings disadvantages. Since the node has to keep track of the neighbour table in order to realize the multicast effect and perform multiple transmissions, the energy consumption and overhead are increased.

Chapter 3

Domain Name System

The Domain Name System (DNS) is a hierarchical naming system built on distributed databases for computers, services, or any resource connected to the Internet or a private network[3]. It is an application-layer protocol.

3.1 Domain Name System

Compared with the memorable domain name, it is difficult for human beings to remember the IP address of a computer. However, this IP address is essential for identifying the location and MAC address of the network interface of that computer. Thus, DNS is invented to translate the so-called Fully Qualified Domain Name (FQDN) into the IP address and vice versa. Because of the existence of the DNS, the Internet resource could be located world-wide. Meanwhile, DNS supports the discovery of services in the Internet. Notice that the domain name is persistent for a host while the IP address could be changed.

In this section, we start from the DNS hierarchical name space and the distributed organization of the responsibility for the DNS system, to give the detailed introduction about DNS.

3.1.1 Domain Name Space

A domain name is the identification name label that defines a realm of administrative autonomy, authority, or control in the Internet[3]. Simply, a domain name represents and identifies a certain Internet or private network resource such as the web site. Domain names are formed by the DNS.

The domain name space consists of a tree of domain names. As shown in Figure 3.1, each node or leaf represents a domain in the domain name space. For each domain, there is one or more Resource Records (RRs) associated with it. The RR holds the information for the corresponding domain, such as host address (A or AAAA) records, name server (NS) records and mail exchanger (MX) records. A domain zone may consist of one domain, or may consist

of many domains. It is a portion of the global DNS name space for which administrative responsibility has been delegated and managed by a name server.

The details of the Domain Name and Resource Records are specified in RFC 1034.

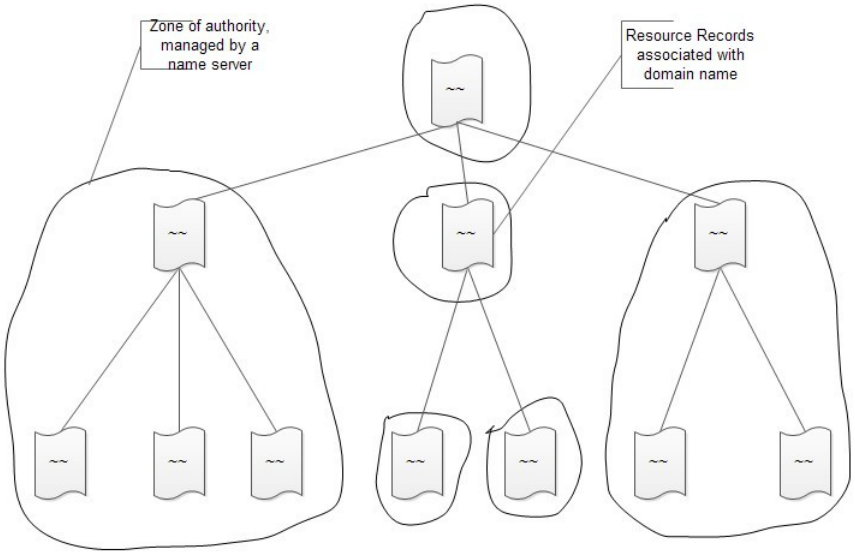


Figure 3.1: Domain Name Space for DNS

3.1.2 DNS Name Syntax

Based on the structure of the domain name space, the DNS name is hierarchically structured, as shown in Figure 3.2. For any path from the root to the bottom, there is a domain name that represents it. For different parts of the domain name space, there is a leveled DNS server in the Internet to handle the specific information. An example is the host name “www.philips.nl”, where the right-most label is the top level domain. “www.philips.nl” belongs to the top level domain “nl”. The label “philips” specifies a sub domain of the “nl” domain. Accordingly, “www” is a sub domain of the “philips.nl” domain. From right to left, the hierarchy of domains ranks from high to low.

3.1.3 Resource Records (RRs)

As mentioned in the section 3.1.1, RRs hold the information for the corresponding domain and all the information stored in the zone files of the Domain Name System. All RRs have the same top-level format as shown in Figure 3.3[2].

- NAME: It is an owner name, e.g., the name of the node to which this RR pertains, it could be “example.local.”.

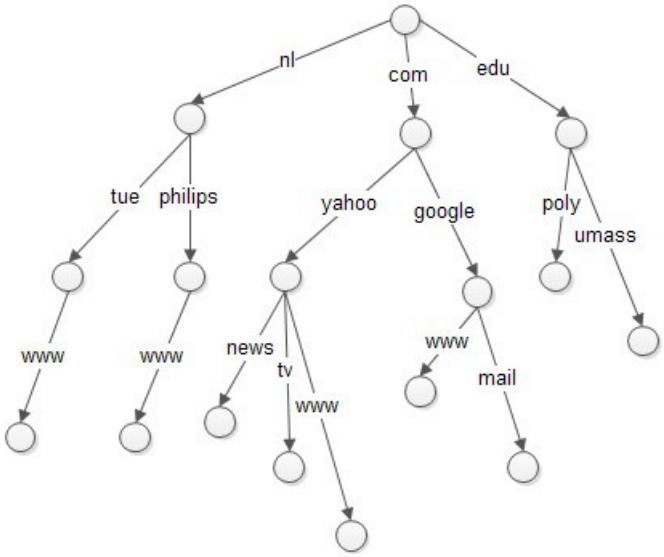


Figure 3.2: DNS Name Structure

- TYPE: It indicates the type of the RR, e.g., AAAA means IPv6 address of a host.
- CLASS: It indicates the class of the RR, e.g., IN means the RR is belong to the Internet.
- TTL: It indicates the life cycle of the RR for a specific owner. After the certain time interval, the information should be consulted again from the source.
- RDLENGTH: It specifies the length of the RDATA field.
- RDATA: It describes the RR, e.g., the RDATA for AAAA type is a specific 128-bit IP address.

Figure 3.4 lists all the information about domain "tue.nl". It includes the RR type such as NS, MX, AAAA and TXT. It is recommended to store the RRs in the format as shown in Figure 3.3 directly in the files of servers. It brings the convenience of generating the response when it is responding to queries. The DNS RRs are the most essential components in the DNS queries and responses which are described in section 3.2.1.

Table 3.1 lists the most commonly used RRs.

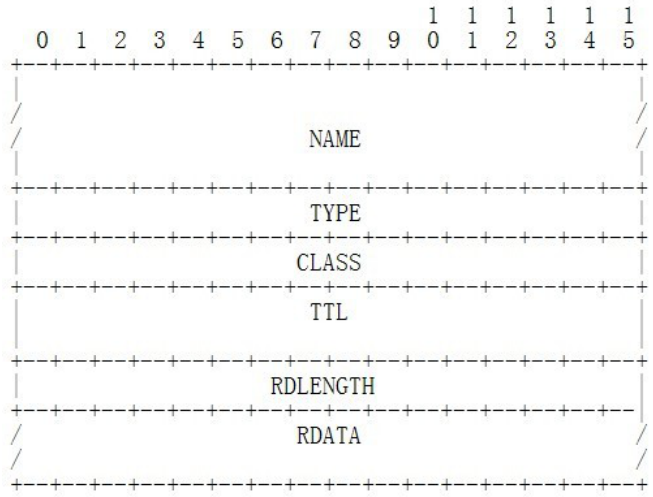


Figure 3.3: RR Format

TYPE	Description	Function
A	IPv4 address record	record a 32-bit IPv4 address
AAAA	IPv6 address record	record a 128-bit IPv6 address
MX	mail exchange record	maps a domain name to a list of message transfer agents for that domain
NS	name server record	specifies a host which should be authoritative for the specified class and domain
PTR	pointer record	pointer to a canonical name
SRV	service locator record	used for locate specific service in specific domain
TXT	text record	for arbitrary human-readable and also machine-readable text in a DNS record

Table 3.1: Common RRs

3.1.4 DNS Name Server

The DNS is implemented by the DNS name server. The DNS name server is a server that stores the DNS records, and responds with answers to queries against its database. There are 4 types of DNS name servers and these "types" are based on their roles during the DNS name resolution process:

- Root name server: It is contacted by the local name server. When a local DNS server receives a query, it contacts the root name server to get the IP mapping of top-level domains.
- Top-level domain (TLD) server: It is responsible for com, org, net, edu, etc and all top-level country domains such as uk, nl and cn. TLD servers

```
> set type=all
> tue.nl
Server:  ns1.tue.nl
Address:  131.155.2.3

tue.nl
        primary name server = ns1.tue.nl
        responsible mail addr = hostmaster.tue.nl
        serial  = 2011092106
        refresh = 43200 (12 hours)
        retry   = 3600 (1 hour)
        expire  = 1209600 (14 days)
        default TTL = 3600 (1 hour)
tue.nl  internet address = 131.155.2.83
tue.nl  nameserver = ns1.tue.nl
tue.nl  nameserver = ns3.tue.nl
tue.nl  nameserver = ns2.tue.nl
tue.nl  MX preference = 100, mail exchanger = mx-a.mf.surf.net
tue.nl  MX preference = 100, mail exchanger = mx-1.mf.surf.net
tue.nl  text =

        "Eindhoven University of Technology"
tue.nl  text =

        "P.O. Box 513, 5600 MB Eindhoven, The Netherlands"
ns1.tue.nl      internet address = 131.155.2.3
ns2.tue.nl      internet address = 131.155.3.3
ns3.tue.nl      internet address = 130.89.2.7
mx-1.mf.surf.net        AAAA IPv6 address = 2001:610:1:80ab:192:87:102:69
mx-1.mf.surf.net        AAAA IPv6 address = 2001:610:1:40aa:194:171:167:221
mx-a.mf.surf.net        internet address = 192.87.102.77
mx-a.mf.surf.net        internet address = 194.171.167.216
mx-a.mf.surf.net        internet address = 195.169.124.156
mx-a.mf.surf.net        AAAA IPv6 address = 2001:610:0:800e:195:169:124:152
mx-a.mf.surf.net        AAAA IPv6 address = 2001:610:0:800e:195:169:124:155
```

Figure 3.4: RRs Example of a Domain

resolve the IP addresses of organizations and return it to the querier.

- Authoritative DNS server: It is responsible for the second-level domain. Here, authoritative DNS server is equal to the organization's DNS server and provides authoritative host names to IP mappings for organization's servers. It is maintained by organizations.

- Local name server: It is also named "default name server". Each Internet Service Provider (ISP) such as companies or universities has one. The query sent from the host first arrives at the local name server, which will forward it to the high-level name servers.

Notice, these 4 types of DNS servers are distinguished according to the process of the DNS name resolution. The root name server is authoritative with respect to addresses of TLD servers, and the TLD servers are authoritative with respect to addresses of authoritative servers at organizations. In essence, the authoritative DNS server means the name server that responds with answers that have been configured by an original source. The original source is specifically configured by the administrator.

3.1.5 DNS Resolver

The DNS component installed in the client side is called a DNS resolver. It is responsible for generating and sequencing the queries that finally achieve a resolution of the resource sought, e.g., translating a domain name into an IP address.

It is common for the DNS server to give the answer of the query after querying other name servers recursively. The simple DNS resolver relies on the recursive DNS server to perform the entire process of finding the information. It is shown in section 3.2.2.

3.2 DNS Operation

As DNS is a distributed database system which is located world-wide, the communication between different DNS components is essential for the DNS operation. In this section, we focus on this aspect and introduce how the DNS system resolve the domain address to the IP address.

3.2.1 DNS Query and Response

All communications inside of the domain protocol are carried in a single format called a message. A normal DNS message consists of several sections. There are 5 sections: Header, Question, Answer, Authority and Additional Sections, as shown in Figure 3.5 and described below.

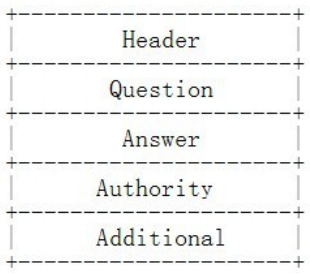


Figure 3.5: DNS Packet Layout

Header Section The header section is always present in a DNS message. The header includes fields that specify which of the remaining sections are present, and also specify whether the message is a query or a response, a standard query or some other opcode, etc[2]. The structure is shown in Figure 3.6.

Each item in the section indicates different information for the DNS packet, which is specified in RFC 1035[2].

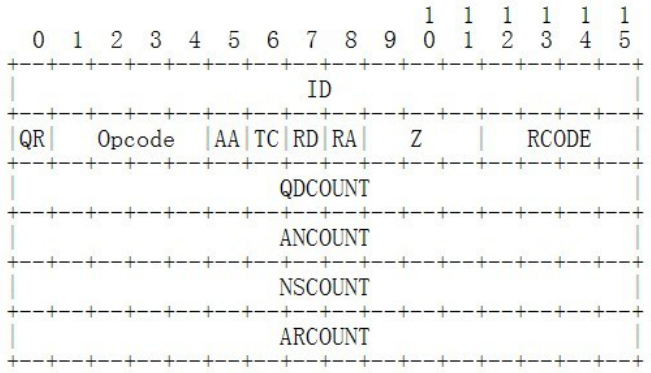


Figure 3.6: Header Section

Question Section The question section is used to carry the "question" in most queries, i.e., the parameter that defines what is being asked. The structure is shown in Figure 3.7.

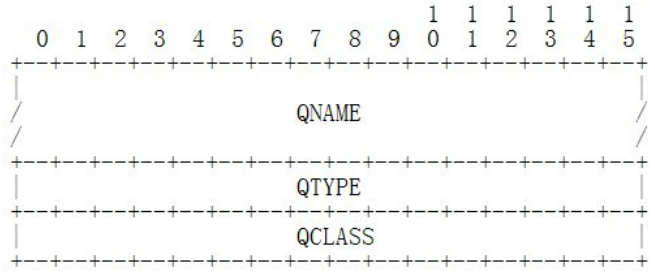


Figure 3.7: Question Section

QNAME, QTYPE and QCLASS together formulate the required information of what is being asked. For example, if QNAME = “www.tue.nl”, QTYPE = “AAAA”, QCLASS = “IN”, it means the IPv6 address of the domain name “www.tue.nl” is asked in the Internet.

Answer, Authority and Additional Sections Answer, authority, and additional sections share the same format, as shown in Figure 3.3, and all of them are called resource record sections. The answer section contains RRs that answer the question; the authority section contains RRs that point toward an authoritative name server; the additional records section contains RRs which relate to the query, but are not direct answers for the question.

DNS Queries: The normal DNS query consists of the header and the question sections. The DNS query is sent to the DNS server by the resolver or by other

servers when the recursive look up is used.

DNS Responses: All the five sections of Figure 3.5 can be present in the DNS response which is generated by the DNS server. Figure 3.8 is a simple response when the domain name "www.tue.nl" is being asked. In the question section, QNAME = "www.tue.nl", QTYPE = "A", QCLASS = "IN". In the answer section, the canonical name of "www.tue.nl" is given which is "webserver.tue.nl" and the address of "webserver.tue.nl" is 131.155.2.83. In the authority record and additional record sections, the name servers of "tue.nl" and the corresponding addresses are given.

```
> set type=A
> www.tue.nl
Server:  ns1.tue.nl
Address:  131.155.2.3

------------
Got answer:
    HEADER:
        opcode = QUERY, id = 4, rcode = NOERROR
        header flags:  response, auth. answer, want recursion, recursion avail.
        questions = 1,  answers = 2,  authority records = 3,  additional = 3

    QUESTIONS:
        www.tue.nl, type = A, class = IN
    ANSWERS:
    ->  www.tue.nl
        canonical name = webserver.tue.nl
        ttl = 600 (10 mins)
    ->  webserver.tue.nl
        internet address = 131.155.2.83
        ttl = 86400 (1 day)
    AUTHORITY RECORDS:
    ->  tue.nl
        nameserver = ns3.tue.nl
        ttl = 86400 (1 day)
    ->  tue.nl
        nameserver = ns1.tue.nl
        ttl = 86400 (1 day)
    ->  tue.nl
        nameserver = ns2.tue.nl
        ttl = 86400 (1 day)
    ADDITIONAL RECORDS:
    ->  ns1.tue.nl
        internet address = 131.155.2.3
        ttl = 86400 (1 day)
    ->  ns2.tue.nl
        internet address = 131.155.3.3
        ttl = 86400 (1 day)
    ->  ns3.tue.nl
        internet address = 130.89.2.7
        ttl = 86400 (1 day)

------------
Name:    webserver.tue.nl
Address:  131.155.2.83
Aliases:  www.tue.nl
```

Figure 3.8: Example of DNS Response

3.2.2 DNS Name Resolution

A simple example of the DNS name resolution process is given in Figure 3.9[36]. In this example, a host "sic.poly.edu" requires the IP address of the host "gain.cs.umass.edu". A DNS query is sent to the local DNS server first. If the local server has the available information in its cache, it will directly respond to the host. Under this condition, the request to the root or top-level DNS server is not required. If the local server does not have the answer, it should forward the query to the root DNS server. After the local server receives the response (the address of the TLD DNS server for that query) from the root server, it will query the address of the authoritative DNS server from the TLD DNS server. Afterwards the local server should query the authoritative DNS server and get the response. Finally, the response will be forwarded to the requesting host "sic.poly.edu". The numbered arrows in Figure 3.9 indicate the order of the different requests and responses.

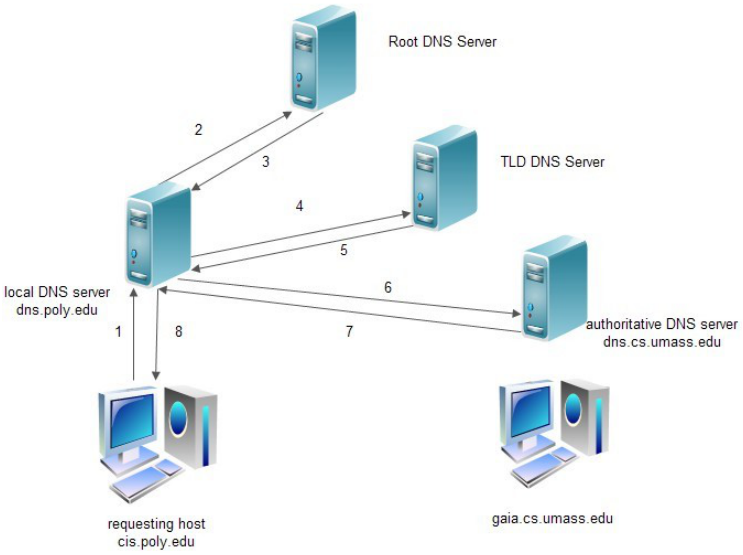


Figure 3.9: Name Resolution Example

3.3 DNS Services

As described in section 3.1, DNS translates the so-called Fully Qualified Name into the IP address and vice versa. In traditional managed networks, the DNS system provides each host the information such as its host name, IP address and domain name server address (DNS server). All the information is configured by the DHCP protocol[29].

Using the DNS system, nodes can look up addresses of nodes supplying services, such as the email service (SMTP)[30], the HTTP[31] service or any other services. For example, a user is requesting the "http://www.tue.nl" URL in the web browser. In order to send this request to the web server named "www.tue.nl", the browser has to know the IP address of that server. The browser should extract the requested domain name and pass it to the DNS client installed in the host. Afterwards, the DNS client will process this request in the DNS system and return the IP address of the requested HTTP server to the web browser. Finally, the web browser will set up a TCP connection with that HTTP server by using the received IP address.

In the above example, "www" is a part of the host name, so the DNS is not used to discover whether the "HTTP" service exists in the "tue.nl" domain or not. For discovery, DNS provides the protocol named DNS-SRV[5] to find and locate the specific service. The DNS-SRV RR specifies the location of the server for a certain protocol and domain. With the DNS-SRV RRs, clients are able to ask for a specific service or protocol in a specific domain, and get the names of available servers back. For example, if a client wants to find out the HTTP server in the "example.com" domain, it may query with the QTYPE "SRV" and QNAME: _http._tcp.example.com

Here, "http" is the desired service, "tcp" is the desired protocol and "example.com" is the domain to which this RR refers.

In the SRV type of the resource record of the response, there would be specific target names to indicate which servers provide the desired service in the specified domain. By the target name, the client is able to look up the IP address of that server and start the communication.

With the DNS-SRV protocol, it would be possible to discover different types of services. However, for many of the existing protocols, alternative discovery solutions exist, such as search engines for HTTP, SSDP for UPnP, or UDDI for web services.

Although DNS-SRV is available for the service discovery, there are some short-comings:

- When multiple service records are reported, the semantics of the results is that they can be used as alternatives for the same functionality.
- Due to the service identifier, e.g., like http, it is only possible to discover services with known protocols.

To remove the short-comings, the DNS-SD protocol is developed. Although DNS-SD is developed together with the Multicast DNS, it does not depend on Multicast DNS. DNS-SD could also be used in the normal DNS system. However, this topic is out of the scope in this work.

It is not always the case that a DNS system is installed in the network. Therefore, the concept of Multicast DNS is introduced in the IETF document[1] and described in Chapter 4.

Chapter 4

Multicast DNS

Multicast DNS is a technology using the traditional DNS programming interface, packet formats and operating semantics, to replace the function of the conventional DNS server, especially in small area networks. It is a promising technology chosen to implement service discovery for wireless ad-hoc networks.

4.1 Multicast DNS Motivations

Multicast DNS is a joint effort by participants of the IETF Zero Configuration Networking (zeroconf)[6] and DNS Extensions (dnsext)[7] working groups. The requirements are driven by the Zeroconf working group; the implementation details are a chartered work item for the DNSEXT group[8]. Multicast DNS is used by Bonjour[23] of Apple Inc and Linux Avahi[24] service discovery systems.

There are in general four considerations for the practical significance of Multicast DNS:

- to work in ad-hoc or unmanaged networks (by using a dedicated, special part of the DNS namespace): It is not always the case that the traditional DNS system is installed in every type of network. In ad-hoc or unmanaged networks, no DNS system is installed, hence the node does not know where to find the information such as other node's domain name, IP address, or domain name server address (DNS server). In order to solve this problem, the concept of ad-hoc service discovery is introduced. It is an important addition to ad-hoc networks since it enables sharing of functionality and resources and makes it possible to run distributed applications without the need for manual installation and maintenance of a service database. Multicast DNS provides the ability to perform DNS-like operations on the local link in the absence of any conventional unicast DNS server[1]. Multicast DNS and its companion technology DNS-based Service Discovery [DNS-SD][4] were created to provide the IP networking with the ease-of-use and autoconfiguration of the system.

- to extend DNS: The traditional DNS queries are remaining valid in the mDNS system and the mDNS Responder (the naming convention of the mDNS standard for the server part) may use the cached knowledge to resolve both the traditional DNS and mDNS queries. Meanwhile, the mDNS Responder might also contact the normal DNS server to resolve the query. All these conditions hold because of Multicast DNS using the same programming interface, packet formats and operating semantics as normal DNS does. The programming interface for the mDNS design in Chapter 5 is quite similar with the DNS resolver library of Linux.

- to work if DNS does not work: Since the same programming interface, packet formats and operating semantics are used, Multicast DNS is able to multicast the normal DNS queries to do the same job as DNS does in a certain context. When the DNS does not work, Multicast DNS can be viewed as the backup solution.

- have an effective, ad-hoc service discovery mechanism: Based on Multicast DNS, DNS-SD and also SRV[5] are used to solve the problem of service discovery.

Based on all these considerations, we introduce the mDNS technology as the main technology in this work. During the introduction, some key characteristics of Multicast DNS will be described in this chapter. Notice, Multicast DNS supports both IPv4 and IPv6. IPv6, as the promising Internet Protocol and the development trend, is the only version of IP to be considered in this work.

4.2 Multicast DNS Name Space

Normally, Multicast DNS is applied only on the link-local scope, which means every node in the local area that can be reached without routing. Specifically, the mDNS data packet will not be forwarded by any router. In this work, the discussion about the Multicast DNS is mainly constrained to the link-local scope.

In the link-local scope, IETF defines one top-level domain ".local." which is reserved for the link-local name used in Multicast DNS. Compared with the hierarchical structure used in traditional DNS name convention, an mDNS name is recommended in a flat structure which is defined by IETF[1]. It allows any device to generate its link-local domain name in the form: "single-dns-label.local.", e.g , "MyComputer.local.". The top-level domain ".local." is meaningful only on the link where it originates. The mDNS name structure is shown in Figure 4.1.

Meanwhile, it is also possible for users to define the hierarchical name by themselves. As shown in Figure 4.2, the link-local name could be "c.printing.local." or "d.printing.local.", which is easy to read and distinguishable for the users. For the view of Multicast DNS, "c.printing" or "d.printing" is still in one domain.

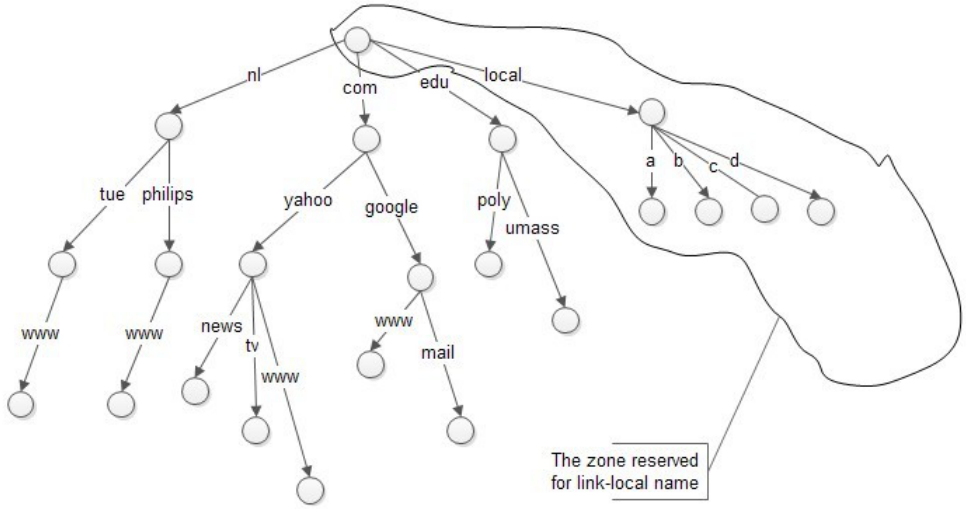


Figure 4.1: Multicast DNS Name Structure

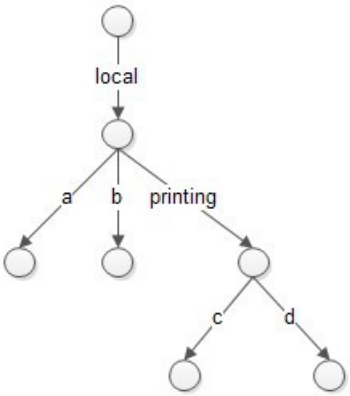


Figure 4.2: Hierarchical Name Structure for Multicast DNS

The domain ".local." is treated the same as any other domain that might appear in the DNS search list but has only local significance. If the domain name ends with ".local.", it means this message should be processed by the mDNS protocol.

4.3 Multicast DNS Name Resolution

The name resolution process for traditional DNS is given in Chapter 3. Compared with the traditional DNS name resolution, Multicast DNS achieves this goal in a more direct way. When Multicast DNS is implemented, every node is able to act as both the server and the client. This is the most essential characteristic for mDNS protocol. In principle, when the total number of nodes in the network is small, Multicast DNS could be extremely efficient.

Assume there are three basic devices, node1, node2 and node3, as shown in Figure 4.3. Every device has its own functionality which is mentioned in the figure.

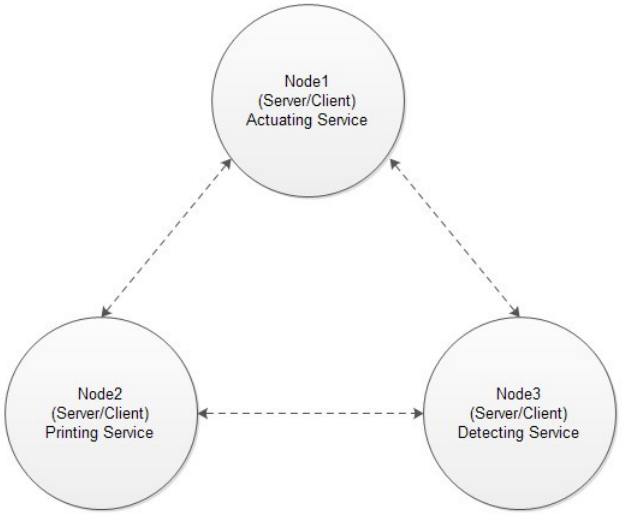


Figure 4.3: Wireless Ad-hoc Network Example

Assume node1 requires to get the IP address of node2. Since there is no conventional DNS server in this small network, node1 does not know where to ask for the information. In this case, node1 sends the multicast query asking for the IP address of domain name "node2.local" to all devices in this local network. Consequently, both node2 and node3 in the local network receive the query and decide to respond to it or not. Since node2 has the authority for this query, it generates the response to answer the question. Obviously, node2 acts as the server for this query. The generated response will be multicast to the local network. Later, node1 will deal with the received message accordingly. For example, it could access node2 and make use of the printing service. There are two approaches for node1 to know that node2 provides the printing service. The first approach is that node1 may cache the periodical service announcement sent by node2. The second approach is that node1 may use the DNS-SD to discover

the service provided by node2. Meanwhile, node3 may also renew the received information in its cache. In this example, node2 has the authority for the query because it has the domain name "node2". On this link-local scope, only the device "node2" can use this domain name once it bootstraps and multicasts the announcement with the domain name "node2".

4.4 Resource Records (RRs)

Compared with the traditional RRs, mDNS RRs have the same format and content. They are described in Chapter 3.

There are two kinds of RR sets for the mDNS protocol: shared and unique[1].

- shared: it is the resource record set where several mDNS Responders have records with the same name, rrtype and rrclass, and all these Responders have the authority to respond to a particular query.
- unique: it is the resource record set that only holds by a certain Responder with the specific name, rrtype and rrclass, and only this Responder has the authority to respond to the specific query with that name, rrtype and rrclass.

Since every node in the local network is able to respond to the certain query and the network traffic is preferred to be reduced, the unique resource record set is more frequently used by mDNS implementations. Before an mDNS Responder claims the ownership of a resource record set, it has to probe to verify that there is no other Responder already claims ownership of that RR set. Multicast DNS achieves this functionality by performing Probing and Announcing when it is starting up. Whenever an mDNS Responder starts up, wakes up from sleep, or receives an indication of a network interface "Link Change" event, it has to perform the two startup steps: Probing and Announcing.

Probing The mDNS Responder needs to make sure that all resource records that are desired to be unique in the local network have been verified. The primary example is the host's address record which maps its unique host name to its unique link-local IP address. If the host name is already in use, the host must change the name and probe again.

In the Probing step, the node multicasts the queries for three consecutive times using the desired resource record name and class. If there is any conflict message received from other Responders in the local network, the Responder has to change the name and repeat the startup step from the beginning. Figure 4.4 indicates the procedure for the startup of node 1. Node 2 and node 3 are in the normal state which means they have finished the startup steps.

Once the node confirms it is unique in the local area, it goes to the Announce step.

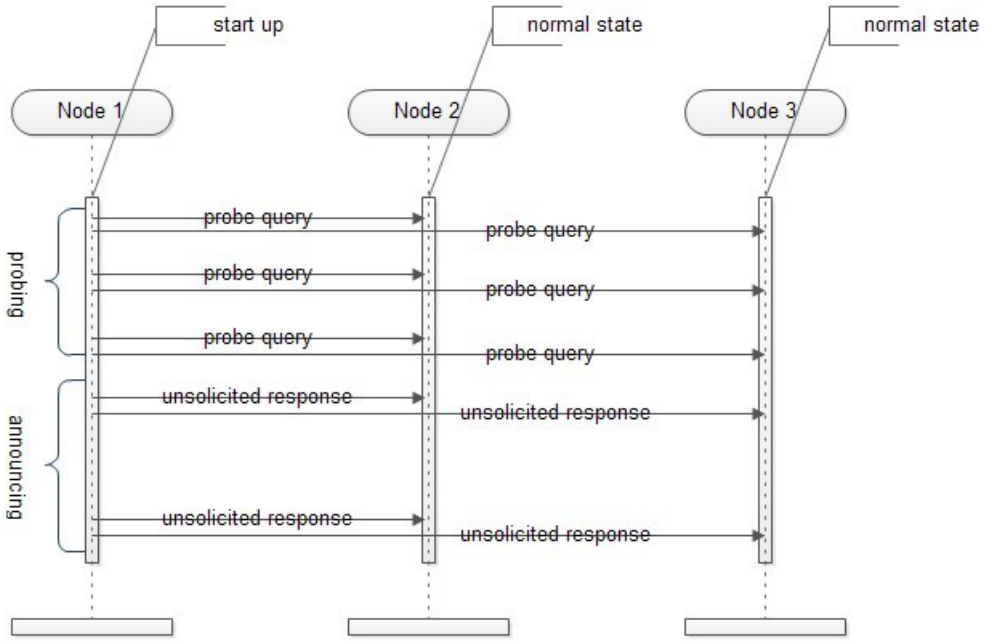


Figure 4.4: Message Sequence Chart for Startup

Announcing In the Announcing step, the Responder must send an unsolicited mDNS response containing all the newly registered resource records in the answer section to the link-local scope. This unsolicited mDNS response is used to claim the ownship of the specific resource records. The unsolicited response must be sent at least twice, and the delay between additional unsolicited responses should be increased by a factor of two for each response.

4.5 mDNS Query

The semantics of the mDNS query is quite similar to the normal DNS query. There are two kinds of mDNS queries: one-shot mDNS queries and continuous ongoing mDNS queries[1].

- One-shot mDNS queries: In principle, one-shot mDNS queries could be sent from a simple DNS resolver (the client part of DNS). The mDNS resolver (the client part of Multicast DNS) may just send the query blindly to FF02::FB (the specific multicast IPv6 address for mDNS). This functionality requires only minor modifications to the traditional DNS resolver library. This kind of simple DNS resolver only takes the first response it receives. The queries sent by this DNS resolver must not use the UDP source port 5353, which is reserved for the fully-compliant mDNS resolver.

- Continuous mDNS queries: When more than one response is needed by a certain query, or the querying operation continues until no further responses are required, the fully-compliant mDNS resolver, rather than the simple DNS resolver, is needed. If the IPv6 address of another host is required, one-shot mDNS queries are enough. Nevertheless, if the operation is browsing to present DNS-SD services on the local network, the mDNS resolver will continuously send the query to ask the available services existing in the network. Then, the multicast responses are required until the user application asks to stop. On this condition, the compliant mDNS resolver is necessary. The compliant mDNS resolver must send the query from the UDP source port 5353, and listen for mDNS responses sent to UDP destination port 5353. "5353" is the well-known UDP port assigned to Multicast DNS.

In order to reduce the traffic in the local network, Multicast DNS allows multiple questions in one query. It is identical to several queries which involve a single question each. The mDNS query is able to require unicast mDNS responses and it is even possible to unicast the mDNS query to a specific destination.

4.6 mDNS Response

For any mDNS response, the resource record section of that response must be explicitly authoritative by the mDNS Responder. Once an mDNS Responder claims the ownership of a resource record set in the Announcing step, it is authoritative for that resource record. Compared with the normal DNS response, a small difference is that the Question Section of the mDNS response must be empty.

There are the explicit rules for matching the question[1]:

- The record name must match the question name.
- The record rrtype must match the question qtype unless the type is "ANY".
- The record rrclass must match the question qclass unless the qclass is "ANY".

Responding to address queries is one of the most significant roles played by Multicast DNS. The name resolution procedure is already introduced in section 3.3. The mDNS Responder is able to respond to Multi-Question Queries, and meanwhile reduce the number of response packets.

- Responding to Multi-Question Queries: If it receives the query which contains more than one question, the Responder should generate the response if it has the authority to answer any of the questions. The Responder should randomly delay for a certain time before sending the response, for the consideration of reducing the network congestion.

- Response Aggregation: When it is possible, the Responder should wait for a certain time in order to aggregate more responses into one response packet. For example, when a Responder is going to send a response with the address record, it receives the query about the service provided by itself, then the Responder is able to aggregate these two records into one response packet. It is also a way of reducing the traffic load. This functionality is done by the Responder.

4.7 Multicast DNS Properties

Next to the development of conventional DNS, Multicast DNS has its own characteristics. All these characteristics make it possible to implement Multicast DNS in reality.

4.7.1 Traffic Reduction

Multicast DNS could generate a large amount of traffic because of the characteristics of multicast. In order to avoid the potential burden to the network caused by Multicast DNS, some particular techniques are applied[1].

- Known-Answer Suppression: When the client sends out the query to which it already knows some answers, it should populate the Answer Section of the mDNS query with these answers. The mDNS Responder must not answer the question if the answer is already included in the Answer Section of the query. This technique is only used for the query which asks for the shared resource record of the Resolver. For example, there could be several nodes in the local area providing the printing service. If a user application already knows one printing machine, but it does not want to use it, the known-answer suppression scheme could be applied. The mDNS Resolver will not ask the unique resource record which belongs to itself.

- Multi-Packet Known-Answer Suppression: When there are too many answers in the Known-Answer Section, the TC (Truncated) bit must be set. Then multiple query packets rather than one would be sent. All these query packets contain a part of the known answers.

- Duplicate Question Suppression: If a host receives a query of which the resource record is the same as it is about to ask, it will cancel the plan of sending that query.

- Duplicate Answer Suppression: If a host receives a response of which the resource record is the same as it is about to send, it will cancel the plan of sending that response.

4.7.2 Source Address Check

For all mDNS responses, the IP TTL of them should be set to 255. Based on the value of the IP TTL, the mDNS Resolver checks whether the received message is originated on the local link or not. If the IP TTL of the received DNS response is not 255, the Querier will silently discard it.

When an mDNS query is carried out, the host sending the mDNS query must only process the response that originates from the local link. There are two ways to test whether a response originates on the local link or not:

- For all the responses, the multicast address in IP headers must be FF02::FB.
- For the response with the source IP address prefix FE80::/10, it must be sent from the local link.

4.7.3 Multiple Interfaces

One host can have different network interfaces. All interfaces from the same host may choose the same host name and that host name should be defended by all the active interfaces. If the host name is not unique for one of the interfaces, the host should configure a new host name for all the interfaces.

4.7.4 Multicast DNS Character Set

Historically, the conventional DNS is encoded as US-ASCII text, so it is lack of any support for non-US characters and limited to letters, digits and hyphens. However, actually the DNS specification does not constraint the limitation to DNS names.

In contrast, all names in the mDNS implementation must be encoded as the precomposed UTF-8 “Net-Unicode” text. Since US-ASCII can be considered as a subset of UTF-8, Multicast DNS is compatible with the existing DNS API, library and clients. It is not recommended to use non-US characters to Multicast DNS at this stage. However, it leaves the space for the further development to break through restrictions of the conventional DNS character set.

4.8 Multicast DNS and DNS Comparisons

The traditional DNS programming interface, packet formats and operating semantics is the foundation of Multicast DNS. Multicast DNS inherits the property of DNS and makes some changes to suit the specific usage environment. Thus, there are some commonalities and differences between Multicast DNS and DNS. Table 4.1 lists the comparisons of these two techniques from different points of view. Notice, the mDNS packet could be larger that 512 bytes. However, in order to make it compatible with the conventional DNS packet, it is not recommended to exceed the limitation of 512 bytes.

	Multicast DNS	Traditional DNS
Domain name	Flat structure, maximum size: 255 bytes	Hierarchical structure, maximum size: 255 bytes for each domain
IP address	Link-local/Global IP address	Global IP address
Name server record	No	Yes
Start of authority record	No	Yes
Source/Dest UDP port	5353	53
UDP packet size	Larger than 512 bytes	512 bytes
Question number in query	One or more	One
Known answer suppression	Yes	No
Query ID field	Ignore	Use
Question Section in response	Does not exist	Exist
Server	Each node	Specialized DNS server
Send method	Multicast/Unicast	Unicast

Table 4.1: Comparison between Multicast DNS and DNS

4.8.1 Advantage of Multicast DNS

Multicast DNS realizes the name resolution between IP addresses and host names. The DNS-like operation is executed on the local link in the absence of any conventional DNS server. There is no requirement for administration or configuration of the mDNS system. In addition, no infrastructure is needed for the system to implement the mDNS service. Even when the infrastructure of the system fails, it still works[1]. It is also worth mentioning that the local DNS domain names are free for use which saves the annual cost for the global domain name.

Furthermore, Multicast DNS allows the detection of the name collisions during the probing process without extra error detections, which also saves the network resources. Last but not least, according to the basic principle of Multicast DNS, if the IP prefix of the response is FE80::/10, it can explicitly detect that the response is received from the local network.

4.8.2 Disadvantage of Multicast DNS

Multicast DNS is not suitable to be used in the network with a large number of nodes. Since most queries and responses are sent by multicast, it may generate a large volume of traffic. When there are too many nodes in the network, the burden of the mDNS transmissions on the link will severely influence the performance of the network. Assume there are 1000 nodes in the local network. For the startup of each node, there will be 1000 multicast probing queries and unsolicited responses periodically. At the same time, there will be also the multicast queries and responses generated by different nodes. Apparently, that

will cause an immense burden on the network. It is straightforward that the conventional unicast DNS is more suitable to be applied in such situation.

Multicast DNS cannot be used in multiple IP subnets and it is recommended to be applied only on the link-local scope. This is a huge limitation. In contrast, the conventional DNS is the accepted standard which is used world-wide.

Chapter 5

Specification of mDNS Implementation

This work focuses on the Multicast DNS applied to low-resource devices. There are several differences between the mDNS application on low-resource devices compared to those on normal devices.

When Multicast DNS is applied on normal devices, it works as the normal DNS system. All RRs, such as MX, NS and SOA, have to be considered and handled. On low-resource devices, several functionalities can be left out. Meanwhile, it is also impossible to implement all functionalities of Multicast DNS on low-resource devices because of the constraints on the data/program memories.

In this chapter, a specific design is proposed for the Multicast DNS implementation, which is suitable to be applied on low-resource devices. In section 5.1, the existing DNS and mDNS implementations are introduced. We briefly discuss why it is necessary to give this specific mDNS implementation design on the low-resource devices. Section 5.2 and 5.3 describe the design considerations as well as the design goals. Section 5.4 defines the implementation details. The mDNS APIs are given in section 5.5. Section 5.6 introduces the structure of the designed mDNS Library while section 5.7 gives the details about how the proposed mDNS library is implemented. The data structure for the mDNS design is given in section 5.8. At the end of this chapter, this work provides a potential solution for implementing the mDNS service on the Contiki OS.

5.1 Recap for Existing DNS/mDNS Implementations

In order to realize the mDNS service, one of the direct solutions is modifying the existing DNS implementation. The Linux system library "libresolv.so" handles the resolver part of the DNS. Since it takes 75 KB of the Flash and 11 KB of the RAM for the simple resolver functionality, it is difficult to transplant

the "libresolv.so" on the low-resource devices. Take the chip CC2530[17] as an example, its RAM is 8 KB so that it is far from meeting the requirement of the libresolv.so. The detailed introduction to CC2530 is given in Appendix B. Nowadays, BIND[9] is by far the most commonly used DNS server software in the Internet. In combination with the libresolv.so, Bind is also too big for the target.

	Flash (bytes)	RAM (bytes)
libresolv.so	76864	11732
Bind	41442	1380
Bonjour	68817	6180
Avahi	335382	3740

Table 5.1: Size of Existing DNS/mDNS Implementations on Linux

There are also several specific mDNS implementations. Among these implementations, Bonjour[23] and Avahi[24] are the most ubiquitous but very large and unsuitable for embedded devices. In general, the code for the existing mDNS implementations is rather large and unwieldy for low-resource devices. As shown in Table 5.1, Avahi takes 327 KB of the Flash which is larger than what CC2530 can afford. On a Linux platform, it is necessary for Bonjour to take some of the libraries of Avahi when it is running. Apparently, it is more than what CC2530 can afford.

Based on these considerations, it is reasonable to design and write the code from scratch for implementing the mDNS protocol on the low-resource devices. Thus, this work provides a specific mDNS implementation design.

5.2 Design Considerations

This design of an mDNS implementation aims at making the Multicast DNS running on low-resource devices. There are several issues to be considered:

- The functionality consideration: It is neither necessary nor realistic to implement all the DNS functionalities on small devices. The key point is making the mDNS protocol working in the ad-hoc network consisting of low-resource devices. Meanwhile, the mDNS message should be recognizable by the conventional DNS server and the mDNS Library is able to process the normal incoming DNS messages.

- Low-resource device consideration: Compared with normal devices, low-resource devices have small memories and low computing capabilities. It is especially important to store the data in an economical way and reduce the complexity of data processing.

- Network consideration: The capabilities of the wireless ad-hoc network are not as powerful as other types of networks. The packet size should

not be large in order to improve the success rate of data transmission. For example, IEEE 802.15.4 standard restricts the packet size to 127 bytes in the MAC layer[10], which means there could be potentially a large amount of packets passing through the network. As introduced in section 2.3.1, the normal mDNS packet takes more space than a single 802.15.4 packet. In this case, the fragmentation is necessary in the IEEE 802.15.4 wireless network environment. Accordingly, there are more generated packets in the network. The transmission delay should be controlled to avoid the network congestion.

- Energy consumption consideration: The battery life is an important issue for low-resource devices. The radio and processor activities should be controlled to improve the battery lifetime.

5.3 Design Goals

This work gives specific design goals in order to give a reasonable blueprint and achieve the economical and ideal implementation quality.

- Design Multicast DNS suitable for low-resource devices. That is to say, this mDNS design targets at the device with small memory and low computing capability. The Contiki OS and CC2530 serve as an example with respect to the footprint and capabilities. Meanwhile, the specific APIs of the Contiki OS could be called by the Multicast DNS when necessary (of course, the specific APIs could also be called by other OSs in order to implement Multicast DNS based on this design).
- Use the flat structure for the mDNS name.
- Design the database to store the authority RRs of the host itself and the RRs received from other nodes in the local area.
- Make the mDNS transaction to be independent of other applications running in the system (Contiki OS).
- Design the APIs to intercommunicate with the system.
- Where there are tradeoffs between the cost of acquiring RRs information and the update frequency, the source of the data should control the tradeoff in order to make the protocol applicable for the low-resource devices.

5.4 Subset of functionalities

In order to give a reasonable blueprint for the mDNS implementation design, there should be a definite statement specifying what functionalities are going to be implemented. This subsection lists the detailed functionalities from different perspectives:

- The mDNS implementation does not support the recursive look-up, because it is unnecessary to carry out such operation in the multicast environment.
- Consider the IPv6 address only. Since the mDNS design is based on the low-resource devices on the local area and most IEEE 802.15.4 devices are able to automatically generate the link-local IPv6 address[10], it is much more convenient to neglect the IPv4 address. This design will not influence the communications among different devices on the local link.
- The Local Database should store the local information and the Cache should store the database of the remote nodes. These databases are all based on storing RRs.
- The following RRs will be supported in the mDNS implementation: AAAA, PTR, TXT and SRV.
- In this design, the "CLASS" has only one option "IN".
- In order to minimize the size of the mDNS packet, neither the known-answer suppression nor the multi-packet known-answer suppression are implemented in the design.

5.5 mDNS APIs

The mDNS API is designed for the system applications when they want to access the mDNS service. In this design, when the mDNS API is called by the user application, an internal message would be transmitted to the mDNS thread and activate it. This is shown in Figure 5.1. Once the API is called, the user application should be blocked. The API listed below serves the Resolver part of the mDNS Library and is defined in the C language. For the Server part, there could be other mechanisms to activate it which are introduced in section 5.9.

int mDNS_internal_query(int q_type, string out, string fqdn, string service);

@**param** q_type The type of the user query.

- When q_type is set to ns_t_aaaa, the user query asks the IPv6 address of the fqdn.
- When it is set to ns_t_ptr, the user query asks the PTR records for the fqdn.
- When it is set to ns_t_txt, the user query asks the TXT records of the fqdn.

- When it is set to ns_t_service, the user query asks the specific service of the fqdn.

@**param** out The string for the result of the required information. It could be the IPv6 address, the PTR record, the TXT record and the result for the service query. If the result does not exist in the mDNS environment, out will be empty.

@**param** service It could be different service types, such as HTTP. This value is only valid when the q_type is set to ns_t_service. For other cases, it is ignored.

@**return** On success, return 0. On failure, return -1.

The API below is used to start the mDNS Library. Once it is called, the mDNS Library begins to listen to the incoming messages from the network.

```
int Start_mDNS();
```

@**return** On success, return 0. On failure, return -1.

5.6 Elements of Multicast DNS

A conventional DNS system mainly consists of three parts[3]: the domain name space with resource records, the name server and the resolver.

Among these components, the domain name space with resource records is quite similar to that used in the Multicast DNS. When the Multicast DNS is applied, the node acts as both the client and the server. Therefore, this work combines the name server with the resolver into a single library, named mDNS Library. Thus, in this work, mDNS system consists of two parts: the domain name space with resource records and the mDNS Library.

Figure 5.1 shows the construction of the mDNS Library and the interaction of the Library within the environment. The mDNS Library can be conceptually separated into two parts according to the functionalities: the Resolver part and the Server part. Meanwhile, there are two memory areas allocated within the mDNS Library, named Local Database and Cache, which are discussed in section 5.8. The interaction between the Resolver and the environment is shown in step *a*, *b* and *c* in Figure 5.1. The interaction between the Server and the environment is shown in step *d* in Figure 5.1. All these issues will be discussed in the next section.

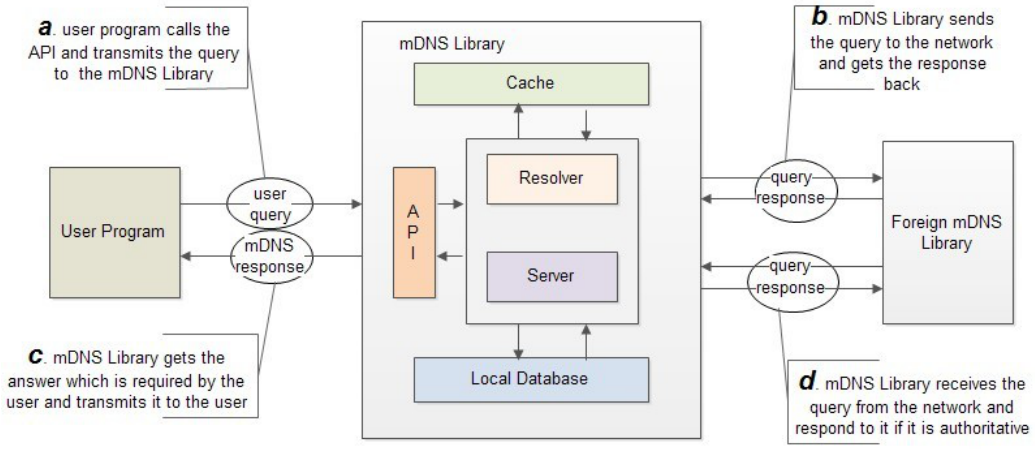


Figure 5.1: mDNS Library

5.7 Implementation of the mDNS Library

The mDNS Library is an individual thread on the host OS. Considering when the mDNS thread is running, it should not disturb other applications running in the OS. For example, when the mDNS library is waiting for the UDP response, it could not block other applications to use the UDP service. This work uses the Contiki OS as the target OS platform, on which the concurrent activities can be realized in a specific way, as discussed in section 5.9.

As mentioned in the previous section, the mDNS Library could be conceptually separated into two parts: The Resolver (client) part and the Server part. The Resolver and Server take one single process inside the mDNS Library which is introduced in section 5.9.

Resolver Part: The Resolver of the mDNS Library is able to communicate with the normal user program. When the mDNS API is called by user programs, the Resolver looks for the required information in its Cache. This procedure is shown in step *a* in Figure 5.1. If the required information is not available, the Resolver will generate an mDNS query and send it to the local network. Once the answer is received, the mDNS Library will refresh its Cache and give the response back to the user program. This procedure is shown in step *b* in Figure 5.1. The user program should be blocked until the required response is received. Once the required information is received, the user program is unblocked. This procedure is shown in step *c* in Figure 5.1. There should be a timer set for the user program. If there is no available information in stipulated time, the mDNS Library will return -1 and unblock the user program.

Server Part: The Server of the mDNS Library keeps on communicating with the foreign mDNS Libraries according to the rules of the mDNS protocol. They are communicating based on the mDNS queries and responses. When the query is received from the network, the Server will check whether to respond to the query or not. If the Server is authoritative to that query, it will generate an mDNS response and send it. This procedure is shown in step *d* in Figure 5.1.

In this section, the behavior of the mDNS Library is divided into 4 sections. These behaviors cooperate together to realize the functionalities of the Resolver and the Server.

5.7.1 Transforming Users Request into a Query

As shown in step *a* in Figure 5.1, when the user program inside the OS calls the mDNS API, the user query is transmitted to the resolver. The Resolver should transform the client's request into a suitable format which can be processed by itself. To be specific, the request is transformed into the corresponding RR set with the specific QNAME and QTYPE. The resolver is able to search the RR in the Cache. When the QNAME and the QTYPE correspond to a single name and a type in the Cache, the Resolver will return an internal response to the user program with the corresponding RR set. This procedure is shown in Figure 5.2. The internal response is the return value of the mDNS API described in section 5.5, and the user program makes use of the value in the string "out".

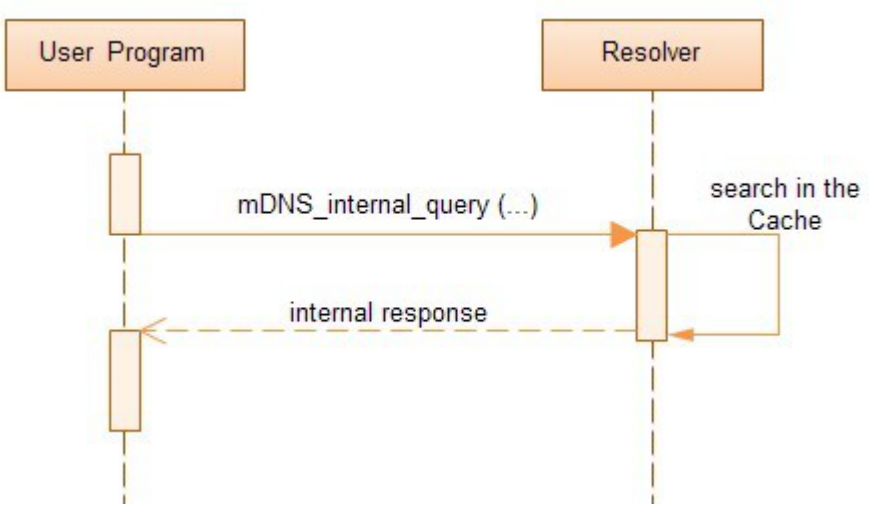


Figure 5.2: The Communication between User Program and Resolver

When the user query is sent, the user application should be blocked until the required information is retrieved from the Resolver, as shown in Figure 5.2. Meanwhile, once the user query is received, the mDNS thread is activated.

A specific block/activate mechanism is employed by the Contiki OS, which is introduced in section 5.9.

5.7.2 Sending the Query

As discussed in the previous section, if the required information is not found in the Cache, the Resolver would generate an mDNS query and send it to the network. This procedure is shown in Figure 5.3.

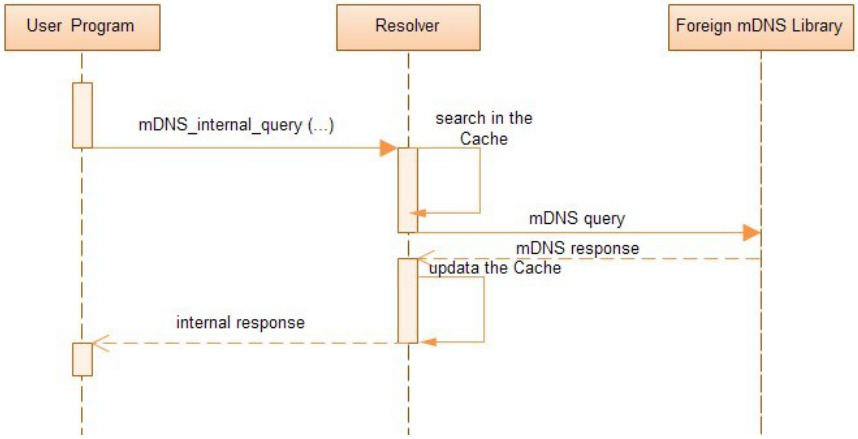


Figure 5.3: Resolver Sends the mDNS Query

One of the basic tasks for the mDNS Library is to format the query which is specified by the user application's request and send the generated query to the local link. Every section of the response should be valid according to the mDNS specification[1]. The normal way of doing this is assigning each item with the correct values in a buffer and then sending them as a whole.

There are only two choices for the mDNS Library:

- Multicast the query to the local link.
- Unicast the query to a certain node on the local link.

In this design, there is only one case to use unicast for the message transmission. When a conflict informing the RRs are already occupied by existing nodes is detected, the conflict response should be unicast to the start-up node. This is for the consideration of avoiding the unnecessary traffic. In other cases, multicast is recommended as the only choice.

In any case, this work assumes that the mDNS Library is multiplexing attention between the request from the user application and the internally generated request.

For mDNS continuous queries, this work recommends to use the same query sending method inside the mDNS Library and send an mDNS query periodically. Figure 5.4 indicates the procedure of the continuous query.

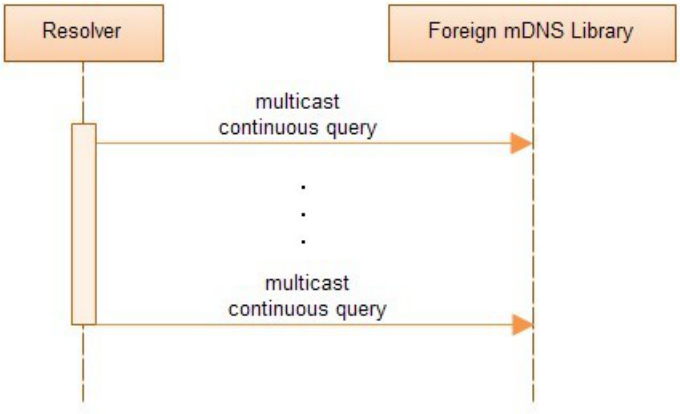


Figure 5.4: The Continuous Query

5.7.3 Processing the Query

When the mDNS Library receives the query from the network, it acts as the Server. Once the mDNS query is received, the Server will extract the useful information and search for it in the Local Database to see whether to respond to the query or not. This procedure is shown in Figure 5.5.

There are specific matching rules for the mDNS Library to process the query:

- The record "name" must match the question name
- The record "rrtype" must match the question qtype.
- The record "rrclass" must match the question qclass.

When the query is in line with all these matching rules listed above, the matched record set would be inserted into the answer section to compose a response. At the same time, the server starts to generate other sections of the response for the query. In short, the response should be valid according to the mDNS specification. Similar to the generation of an mDNS query, the mDNS Library should assign each item with the correct values in a buffer, and send them as a whole.

The algorithm for matching the query to the stored RRs depends on the data structure used for storing the RRs. The detailed matching algorithm is listed below:

1. Extract the length of the QNAME and search for it in the Local Database. If there is no match for the length, ignore that query.
2. Extract the remaining part of the QNAME and search for it in the Local Database. If there is no match for the name, ignore that query.

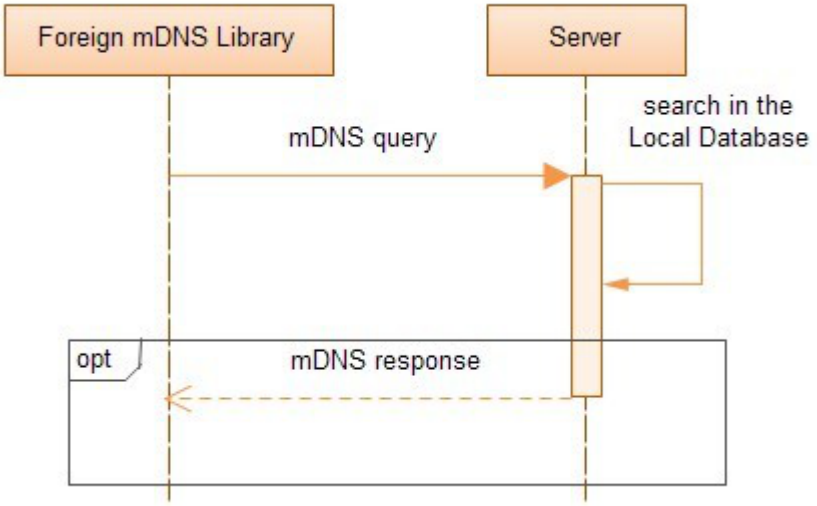


Figure 5.5: Processing the Query

3. If the QTYPE is also matched in a single path, copy the RR set in that path from the root into the answer section.

4. Set all the values in the Header Section and send the response.

The detailed data structure for Local Database and Cache is introduced in section 5.8, which describes more about how the data could be stored and searched.

Compared with normal DNS, the recursive service is excluded in the mDNS matching algorithm.

5.7.4 Processing the Response

There are two kinds of mDNS responses:

- The mDNS response which answers the mDNS query. This kind of response is shown in Figure 5.3.
- The unsolicited response which announces its resources and services when the node starts up. This kind response is shown in Figure 5.6.

The mDNS implementation recommends to adopt the same approach to process both kinds of the responses. The mDNS implementation may cache any mDNS response messages it receives, for the possible use in the future. Compared with the normal DNS, the mDNS protocol does not care about which query asks the certain question. In contrast, Multicast DNS only cares about whether the information is useful or not. Thus, the mDNS Library should only check whether the same RR set exists in the Cache. If the RR set is already stored,

the old one should be replaced with the new one. If the RR set does not exist in the Cache, it should be inserted into the Cache. The storage of Cache is described in subsection 5.8.2.3.

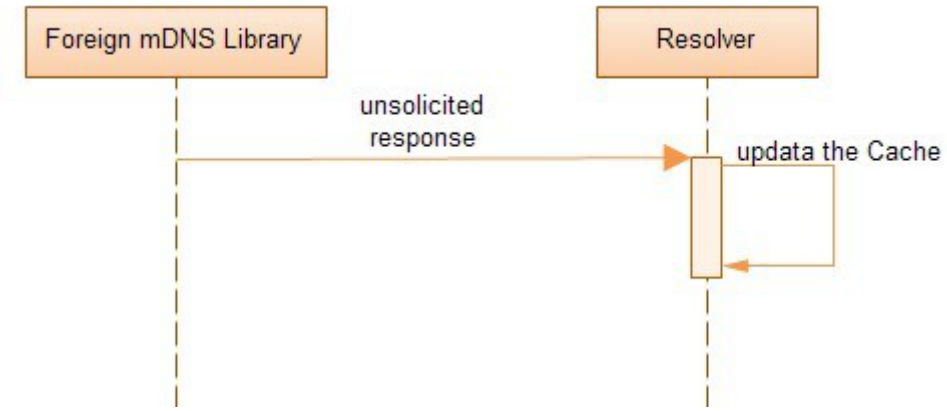


Figure 5.6: Processing the Unsolicited Response

When the response is received, there are three steps to be done:

- Parse each section in the message and insure that all RRs are correctly formatted.
- Analyze the RR and then store it in the Cache.
- If the RR is requested by a user program and has not been replied, the Resolver should transmit an internal response to the requesting user program. This procedure is shown in Figure 5.3.

5.8 Data Structure

The data structure for storing the data as well as the packet layout are essential for designing and implementing the mDNS protocol. This section focuses on these fields with the following design issues:

- The database is not allowed to be large on the small devices used in the local area. The target platform CC2530 is equipped with a relatively large flash (256 KB) and a relatively small RAM (8 KB). Based on this premise, the Local Database could be statically stored in the Flash for all the time. When the device is bootstrapping, the required data could be loaded to the RAM. The Cache is dynamically allocated in RAM.
- Traverse the tree using case-insensitive comparison. When the mDNS Library processes the query/response, the comparison between the extracted

RRs and the information stored in the memory should be hierarchical to improve the efficiency.

5.8.1 Message Packet Layout:

In order to reuse the existing DNS implementations, Multicast DNS uses the same message layouts as the normal DNS implementations. In order to provide mDNS specific extensions, some fields in the DNS packet layout are reused or overloaded by Multicast DNS.

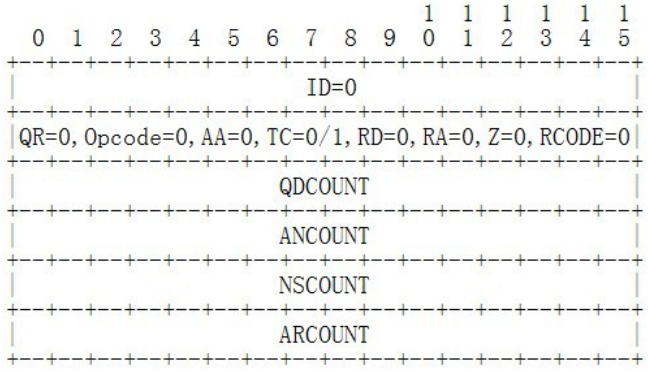


Figure 5.7: Header Section for Query

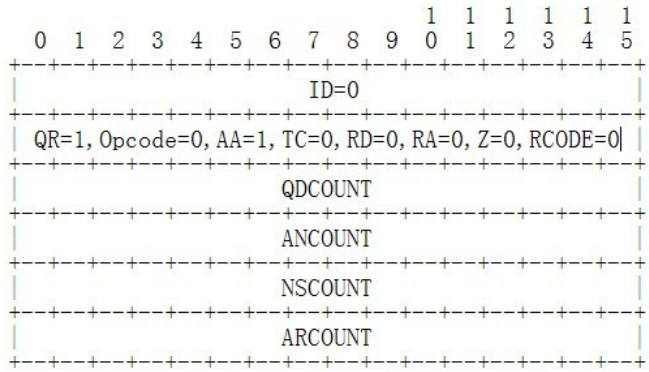


Figure 5.8: Header Section for Response

Figure 5.7 and 5.8 give the most common settings for mDNS queries and responses. For Multicast DNS, the ID field is always set to 0. Since Multicast DNS focuses on the useful answers rather than who sent the useful information, it is unnecessary for Multicast DNS to match the response ID with the query ID. The OPCODE field must be 0 for all mDNS queries and responses. The

AA bit must be 1 for all mDNS responses and 0 for all mDNS queries. In this work, the TC bit is always set to 0 for mDNS queries and responses. The RD, RA, Z and RCODE field are always set to 0. The other items in the Header Section are used as specified in RFC 1035.

For the remaining sections of mDNS messages, there are no differences compared with normal DNS messages. The only exception is that the Question Section should be empty in the mDNS response, that is, QDCOUNT is set to 0. It helps in reducing the size of the mDNS packet.

5.8.2 Database

As introduced in section 5.6, there are two memory areas allocated for the mDNS Library, named the Local Database and the Cache. The database stored in the mDNS Library is allocated in these two areas.

5.8.2.1 Local Database

The Local Database is used to store the information for the node itself. The information is stored in either the Flash or the ROM and expressed by the RRs, such as AAAA, PTR and TXT. A major issue of the Local database design is determining a structure which allows the mDNS name server to deal with matches of the mDNS name server's host.

For DNS, the domain name consists of a sequence of labels. Each label is represented as a one octet length field followed by that number of octets. For an mDNS domain name, there is only one top-level domain ".local." and all the characters before the ".local." are taken as a single part. Thus, the first octet length could indicate the entire length of the name. This work takes the "LENGTH" as a particularly important item which is stored in the database. Figure 5.9 is the design of the Local Database. Assuming that the IP address of "example1.local." is required, the LENGTH table is first looked up. Since it is matched (i.e. the length of "example1" is 8), the NAME table is looked up. Since "example1" does exist in the NAME table, the look-up continues with the RRs table. Finally, the IPv6 address of "example1.local." stored in the RRs table will be returned to the system.

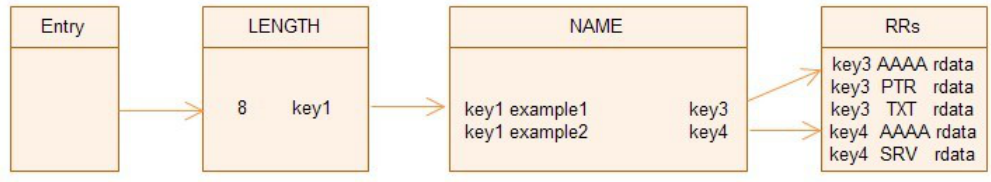


Figure 5.9: Design of a Local Database

Figure 5.10 gives the implementation details about the design of the Local Database. "LENGTH" is the root followed with a tree structure list. Following

the root "LENGTH" are "NAME", "TYPE" and "RDATA" in order. Take the same example in the previous paragraph, assuming that the IP address of "example1.local." is required. The LENGTH, NAME, TYPE tables should be indexed in order. Finally, the left most path in the tree is matched with the requirement and this set of information will be returned.

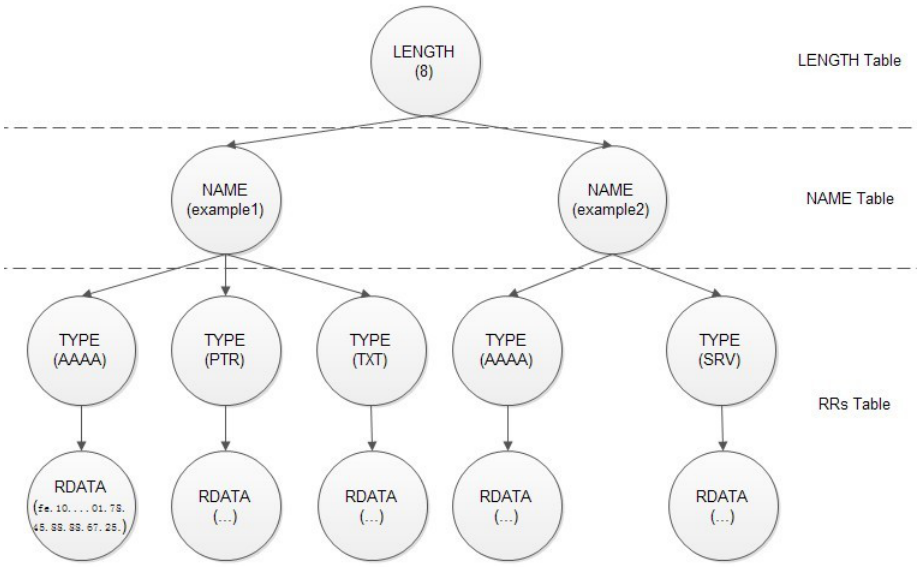


Figure 5.10: Implementation for Local Database

In the designed tree, each path from the root to the bottom node contains a set of information. Any possible length of "NAME" is followed with an individual tree which stores several RRs. Figure 5.11 gives the data structure of the Local Database. When a certain set of RRs stored in the tree is required, it should be copied to the RAM in order to generate the response message.

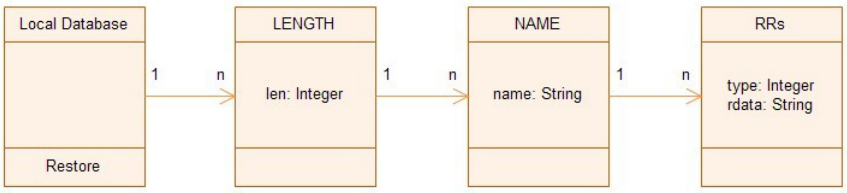


Figure 5.11: Data Structure for Local Database

The information stored in the Local Database is statically allocated in either the Flash or the ROM. The NAME is determined at the startup stage and negotiated with neighbors. In order to avoid the name conflict, every node

should have a backup name. Notice that only the class "IN" is taken into account in this design. For sake of saving the memory space, it is not necessary to include the class item in the Local Database.

5.8.2.2 Cache

The Cache is used to store the information of other nodes in the local area. A similar data structure is used as for the Local Database. It should be relatively dynamical compared with the Local Database. Once the RRs are copied from the received response messages, they are stored in the RAM whose size is limited. Taking this into account, this work limits the size of the Cache to a reasonable range.

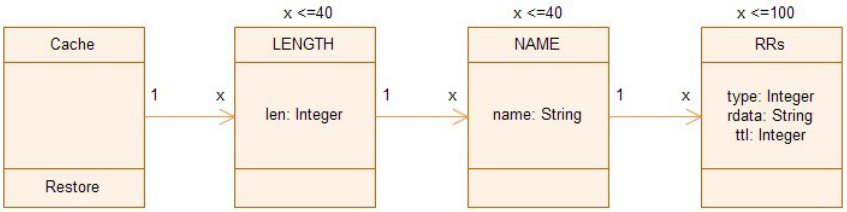


Figure 5.12: Data Structure for Cache

As shown in Figure 5.12, the tree structured Cache limits the entry number for different storage levels. In the first entry, there could be 40 different lengths for the name. In the second entry, there could be 40 names for different RRs. In the third entry, there could be in total 100 RR sets to be stored. That is to say, a single length can be followed with 100 RR sets in an extreme case. Figure 5.13 gives an example of a Cache. From the Cache entry of the figure, 2 name lengths are stored in the LENGTH table, which means 2 domain name space trees are stored in the Cache.

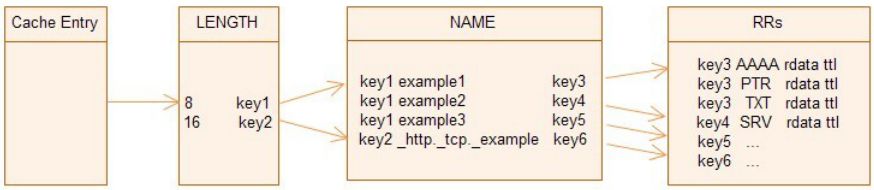


Figure 5.13: Example of a Cache

The typical Contiki configuration takes roughly 2 KB of the RAM and the CC2530 solution provides 8 KB RAM. Thus, there is probably 6 KB of RAM available at run time. In theory, the designed Cache is able to store 100 sets of the RRs which would occupy maximum to 5 KB of the RAM. In other words, the taken space by the Cache does not exceed the available space of RAM.

Compared with the Local Database, another difference is that the TTL of the RR is taken into account for the Cache. This does make sense and will be discussed in the next subsection.

5.8.2.3 Cache Look up and Policy

Based on the data structure of the Cache, this subsection introduces the Cache look up procedure and policy. As shown in Figure 5.14, there are two requests for the Cache: (1) the read request and (2) the write request. When the Resolver is asking for some information, the Cache needs to be read. When the Resolver receives useful messages, the Cache needs to be written. The look up procedure for the Cache is hierarchical.

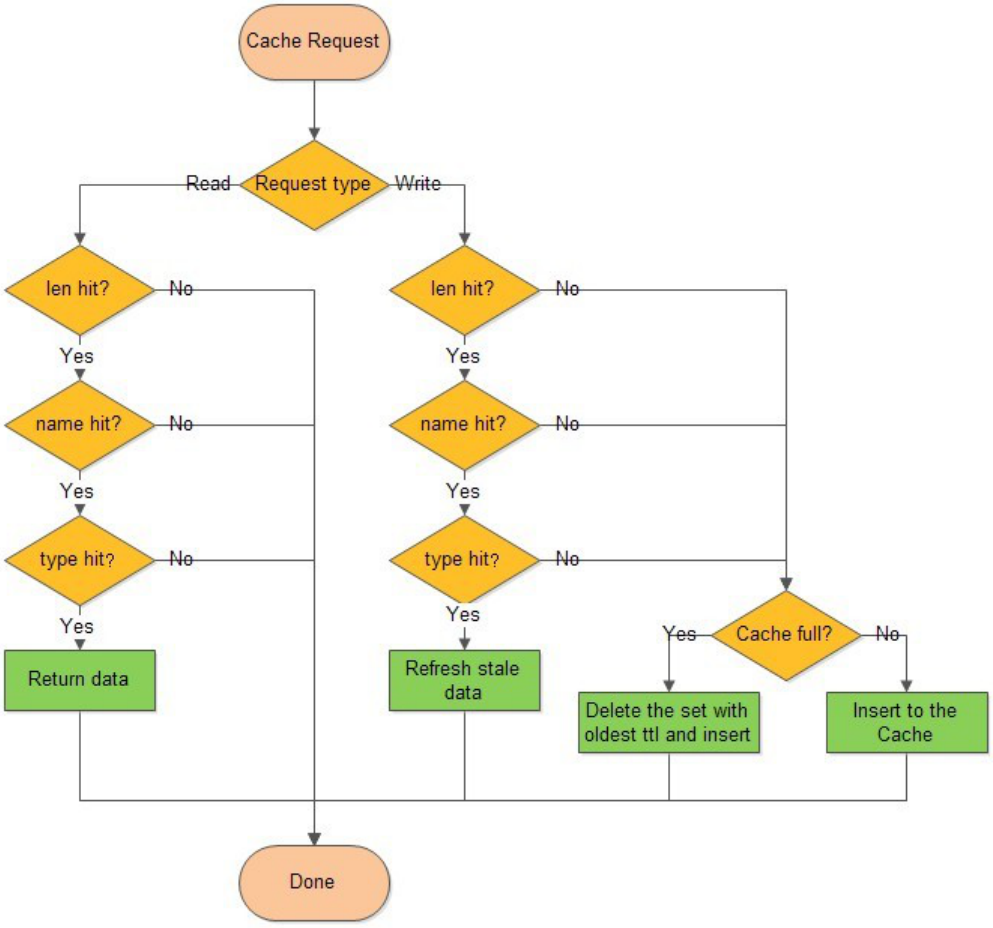


Figure 5.14: Cache Look up and Policy

When a brand new RR set is written to the Cache and the Cache is full, the

mDNS Library will delete the RR set with the oldest ttl except for the length. If there are other names with the same length, this length entry should be kept. Otherwise it should be deleted. After that the new RR set should be inserted into the Cache.

5.8.2.4 Internal APIs

As shown in Figure 5.1, there are interactions between the Cache and the Resolver inside the mDNS Library. Normally, the Resolver processes the received message and stores the useful information in the Cache. On the other hand, the Resolver searches for the required information in the Cache when the internal user program asks for it. This work provides the internal API between the Cache and the Resolver. The first API listed below is used to search for the useful information in the Cache and the second one is used to store the information in the Cache.

int mDNS_internal_Cache_Search (string out, string fqdn, string type);

@**param** out The string for the result of the required information. It could be the IPv6 address, the PTR record, the TXT record and the result for the service query. If the result does not exist in the mDNS environment, out will be empty.
@**param** fqdn It consists of two parts: the length and the name. These two parts will be searched in the Cache respectively.
@**param** type It is the type of the RRs, such as AAAA, PTR, TXT and SRV.
@**return** On success, return 0. On failure, return -1.

int mDNS_internal_Cache_Store (string fqdn, string type, string rdata, int ttl);

@**param** fqdn It consists of two parts: the length and the name. These two parts will be stored in the Cache respectively.
@**param** type It is the type of the RRs, such as AAAA, PTR, TXT and SRV.
@**param** rdata It is the data of the RRs.
@**param** ttl specifies the time interval that the RR may be cached before the source of the information should again be consulted.
@**return** On success, return 0. On failure, return -1.

5.9 Execution Scheme for the Contiki and the mDNS Implementation

In this section, the basic execution scheme for the Contiki OS is introduced. Based on that, this work provides a potential solution for implementing the mDNS service on the Contiki OS.

5.9.1 Event-driven Contiki Kernel

Contiki kernel is event-based[15]. It means a process is invoked when the specific events (e.g., the time-out) occur. Meanwhile, the process must not be blocked when it is activated. A detailed introduction to Contiki is given in Appendix A.

Compared with traditional multi-threading processes, the event-driven strategy has its own advantages, such as the low memory requirement. To be specific, the event-driven kernel only needs a single stack while the multi-threading kernel requires one stack for each individual thread. The low memory requirement makes the event-driven kernel applicable for low-resource devices.

5.9.2 Protothreads

Protothreads are lightweight stackless threads designed especially for the memory constrained system[15]. So it is an ideal solution for the embedded system. Protothreads provide the linear code execution for event-driven systems implemented in the C language.

```
int a_protothread(struct pt *pt){
  PT_BEGIN(pt);
  /* ... */
  PT_WAIT_UNTIL(pt, condition1);
  /* ... */
  if(something) {
     /* ... */
     PT_WAIT_UNTIL(pt, condition2);
     /* ... */
  }
  PT_END(pt);
}
```

The example above shows the basic use of protothreads. It is obvious that the control flow in protothreads is sequential. Once a protothread is invoked, it is executing in sequence. It may be blocked somewhere until a certain condition is satisfied. As shown in the example code, a thread is waiting for "condition1" as soon as it is started. Later it is blocked again until "condition2" is satisfied and then continues. The programing constructs “if” and “while” are commonly

used in the protothread coding. They determine whether a condition is satisfied or not.

The PT_ macros create local continuation points, such that the function continues where it was stopped previously. As a result, the scheduler of Contiki is able to execute multiple protothreads by means of cooperative multitasking. For example, the PT_WAIT_UNTIL macro will cause the function to evaluate the condition until it becomes true.

Next to protothreads, Contiki introduces processes which provide a higher abstraction, but still use protothreads inside. In particular, Contiki provides methods to start and stop processes and to schedule their execution.

5.9.3 Running Scheme for Multicast DNS on Contiki

In this work, it is recommended to use one process for the mDNS service and another process for the user application. The developer can use the mDNS service in the user application by calling functions from the mDNS API.

The general structure of the mDNS implementation is shown in Figure 5.15:

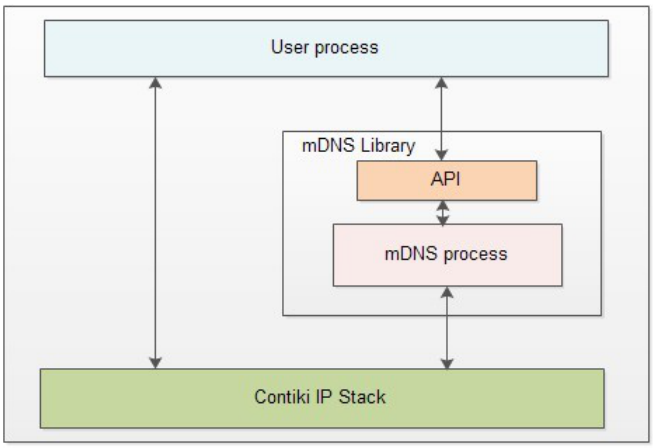


Figure 5.15: mDNS Implementation Structure

5.9.3.1 Process for Multicast DNS

The code shown below is the skeleton of the mDNS process. Once the process is started, it goes into the while loop and the "PROCESS_YIELD()" is executed. The program will immediately jump to another thread until the specific events occur or requests return to this mDNS process, such as a timer that expires or a new message that is received. As the condition for the while loop is always true, the mDNS process will not terminate once it is started.

```
PROCESS_THREAD(mDNS, ev, data) {
  PROCESS_BEGIN();
  while(1) {
     PROCESS_YIELD();
     if(initialized) {
        if(internal_query_received(ev)){
           mDNS_internal_Cache_Search (...);
        }
        else if(mDNS_query_received(ev)) {
          mDNS_internal_LocalDatabase_Search(...);
        }
        else (mDNS_response_received(ev)){
           mDNS_internal_Cache_Store (...);
        }
     }
  }
  PROCESS_END();
}
```

5.9.3.2 Process for the User Application

In order to use the mDNS service on the Contiki OS, this work suggests to create a user process. In the specific user space, the Contiki application can call the mDNS API and make use of the mDNS service at any time.

```
PROCESS_THREAD(user, ev, data){
  PROCESS_BEGIN();
  Start_mDNS();
    while(1) {
       PROCESS_PAUSE();
       /*...User Space...*/
       mDNS_internal_query(...);
       /*...User Space...*/
    }
  PROCESS_END();
}
```

As shown above, the mDNS service is automatically started as soon as the process is started. Inside the "while" loop, the PROCESS_PAUSE() called to enable the cooperative multitasking and allow other processes, including the mDNS process, to perform their task. The developer creates this own application in the user space by calling the mDNS API to get the required information from the mDNS service, such as the link-local IP address of other devices in

the local network. Since the mDNS_internal_query function waits until it receives a response from the mDNS process, the implementation uses the PROCESS_THREAD macro to create a local continuation point.

Chapter 6

Conclusion

This work mainly focuses on the design of implementing the mDNS protocol on low-resource devices.

In Chapter 2 of the thesis, the concepts of IEEE 802.15.4, IPv6 and 6LoWPAN are introduced. These issues are the hot topics for the development of low-resource devices. Thus, this work chooses them as the basis for the design of mDNS implementation. To be specific, the Contiki OS and its applicable hardware platform CC2530, are used together as the target system in the design. Since there are sufficient Contiki APIs which can be called by the OS, the underlying layer is actually not directly related to the design, which facilitates the work. Nevertheless, the low computing capacity and the small memory as well as the constraint for the MTU of the 802.15.4 packet should be taken into account in the design.

As the basis of the IP networks, DNS plays an extremely important role in the network operations. Multicast DNS, as an effective complement and extension for DNS, has significant research value. Multicast DNS realizes the name resolution and the DNS-like operation is executed on the local link in the absence of any conventional DNS servers. There are no requirements for either administration or configuration of the mDNS system. It is also unnecessary to implement the mDNS service on the managed infrastructure. It is worth mentioning that Multicast DNS is not only applicable on the local link, but also has the potential for developing in a wider range of networks. All these concepts are introduced in Chapter 3 and Chapter 4.

In Chapter 5, the APIs of Multicast DNS are taken as the start point of the design. In order to keep the consistency with the normal DNS system, the APIs designed in this work are quite similar to the DNS resolver library on Linux. In order to reduce the complexity of the mDNS design in our practice, this work combines the Resolver and Server into a single library, called mDNS Library. A light-weighted design for the Local Database and the Cache is quite essential for the mDNS implementation. In the design, a tree data structure with hierarchical tables is applied. According to the flat structure of the local domain name, this work uses the length of the domain name as the root of

the tree structure, which improves the indexing efficiency. Moreover, neither the known-answer suppression nor the multi-packet known-answer suppression is implemented in the design in order to minimize the size of the mDNS packet. This may potentially avoid the transmission problems in the IEEE 802.15.4 multicast environment. Based on the target platform, this work provides a potential solution for implementing the mDNS design in the Contiki OS.

Appendix A

Contiki OS

Contiki is an open source, highly portable, multi-tasking operating system for memory-efficient network embedded systems and wireless sensor networks. Contiki has been used in a variety of projects, such as road tunnel fire monitoring, intrusion detection and wildlife monitoring[15].

Contiki is especially designed for low memory devices. A typical configuration of Contiki is 40 KB for ROM and 2 KB for RAM. This means very little memory space is needed for the Contiki OS.

Contiki supports IP communication and both IPv4 and IPv6 can be applied with Contiki OS. In this work, IPv6 is chosen and Contiki is used as the target platform.

Contiki provides a convenient environment for the development of Multicast DNS, such as the complete Contiki APIs for application developers. In this work, mDNS service is recommended to call these system functions directly from the application level. Thus, it is unnecessary to consider about the data processing of the software stack, e.g., the compression scheme in the 6LoWPAN adaptation layer.

A.1 Communication Stack

There are two communication stacks inside the Contiki OS: µIP and Rime. The application can choose any of them or even both. µIP is used for the communication with Ethernet, while Rime is the stack designed for Low-power radio. As shown in Figure A.1, when the datagram from application 1 is transmitted to the Ethernet, it only goes through the µIP stack. If the datagram is transmitted to the low-power radio, it goes through µIP and Rime in order. Sometimes the datagram from the application level can only go through the Rime stack only, and directly to the radio, such as application 3 in Figure A.1[25].

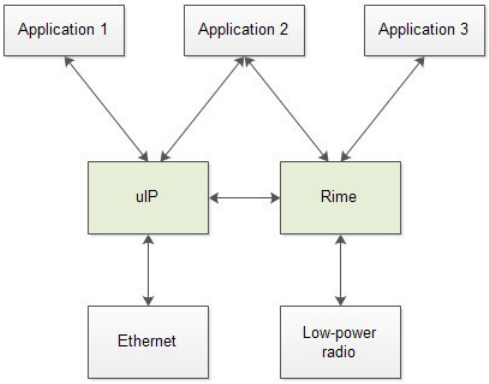


Figure A.1: µIP and Rime

A.1.1 µIPv6

µIPv6 is the IPv6 stack which consumes very little memory space. It is designed especially for the operating system that runs on low-resource devices. It takes about 11.5 KB for the code and requires no more than 2 KB of RAM. For this reason, it is suitable for the constrained hardware platform. The µIPv6 stack is built on the top of the Contiki OS. With the latest Contiki version, IPv6 is supported for the target hardware platform.

µIPv6 does not depend on any particular MAC or link layer. The interface between µIPv6 and lower layers consists of two wrappers for the link-layer input/output functions, the link-layer address, and a couple of constants. This creates a level of abstraction which enables the easy integration of many different MAC and link layer protocols[16], as shown in Figure A.2:

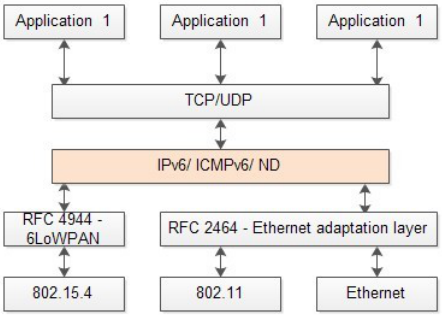


Figure A.2: uIPv6 stack

There is a particularly essential characteristic of the µIPv6 stack: for any incoming and outgoing message, it only uses one global buffer to carry it. The size of the buffer is defined by the MAC header plus 1280 bytes. So different

lower layers can give different buffer sizes. There could be additional buffers to support fragments reassembly if necessary.

For µIPv6, the main data structures are the interface address list and the cache storing the neighborhood devices, prefix list and default router list which are required by the Neighbor Discovery. There are two periodic timers which update the database and manage the data structure respectively.

A.1.2 Rime

Rime is a lightweight communication stack designed for the low-power radio. Rime provides a wide range of communication primitives which are suitable for implementing communication-bound applications or network protocols. All these primitives are in a hierarchical structure, from a simple anonymous broadcaster to mesh network routing. A complex protocol consists of several simple parts, where the more complex module makes use of the simpler ones. Figure A.3[25] lists the main modules:

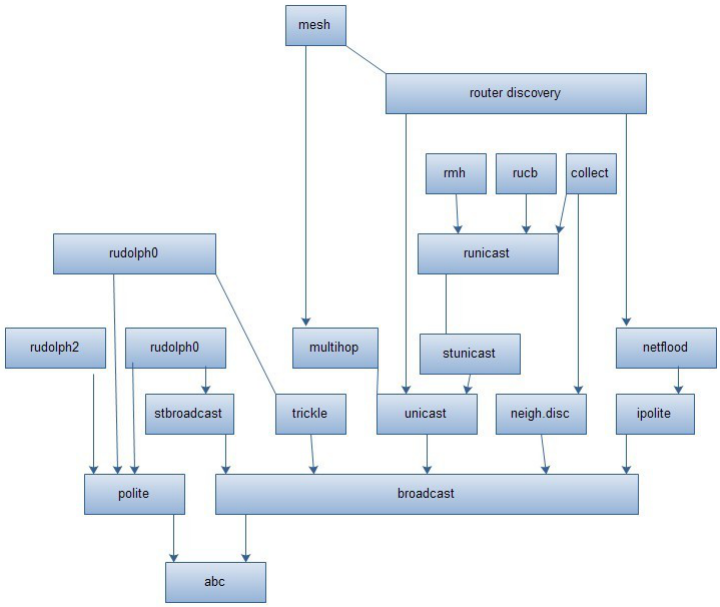


Figure A.3: Overall Organization of Rime Stack

- *abc*: it represents the anonymous broadcast in the bottom level of the Rime stack. The *abc* module receives all the packets coming from radio driver and transmits them to the upper layer.
- *broadcast*: this module gives the source address to the outgoing packet and passes it to the *abc* module.

- *unicast*: this module gives the destination address to the outgoing packets and passes it to the *broadcast* module.
- *multihop*: when a packet is about to be sent, the *multihop* module asks the route table for the next hop and sends this packet to it in the unicast way. When the *multihop* module receives the packet, if the current node is the destination, this packet should be transmitted to the upper layer; otherwise the *multihop* module asks the next hop for the packet and transmits it.

A.1.3 Contiki directory structure

In the Contiki OS distribution, there are 7 directories in total. Each directory has its own functionality.

- app/: there are several architecture independent applications inside the app directory. Each subdirectory contains one application, such as ftp, telnet and web browser.
- core/: this directory includes the system source code. Each subdirectory is for a different part of the system, such as lib, loader and net.
- cpu/: this directory includes all the CPU specific code in each subdirectory. It specifies the CPUs supported by Contiki, such as the CC2430, msp430 and x86.
- doc/: this directory includes all system documentation which describes the system in text files.
- examples/: this directory includes some example project subdirectories. With these example projects, users can do some system tests and code modifications.
- platforms/: this directory includes all the platforms supported by the Contiki OS, such as the win32 and netsim.
- tools/: this directory contains the software for building the Contiki OS for a specific platform.

Appendix B

CC2530EM

CC2530 is a true system-on-chip solution tailored for the IEEE 802.15.4, ZigBee, ZigBee RF4CE and Smart Energy applications[17]. In this project, the CC2530EM is used as the example hardware platform. This module can be used for RF remote control, building automation and industrial control and monitoring.

Figure B.1: CC2530EM

The CC2530 chip has a large flash memory area. It combines a fully integrated, high-performance 2.4GH RF transceiver with an 8051 MCU, 8KB of RAM, up to 256 KB flash memory, and also other powerful supporting features and peripherals.

CC2530 has various operating modes. The transition time between operating modes is short, which ensures the low energy consumption for CC2530.

Appendix C

IAR Workbench

The commonly used Contiki version for CC2530 is compiled and debugged using the IAR Embedded Workbench for 8051[26]. Therefore, this work recommends to choose the IAR workbench as the development tool.

There are many advantages of IAR workbench. IAR Embedded Workbench with its optimizing C/C++ compiler provides extensive support for a wide range of 8051 devices. There is an integrated development environment with project management tools and editor for the IAR. The optimizing compiler is able to generate very compact and efficient code. A variety of configuration files for 8051 devices from different manufacturers make the workbench meet multiple products requirements. In addition, ready-made examples and code templates are included in the workbench.

Appendix D

SmartRF05EB

As shown in Figure D.1, the SmartRF05 Evaluation Board (SmartRF05EB)[27] is the motherboard in several development kits for the Low Power RF device from Texas Instruments. SmartRF05EB is the platform for the evaluation module, such as CC2530. It can be connected to a PC via USB and controls the evaluation module. The board has several user interfaces and connections to external interfaces, which allow fast prototyping and testing of both software and hardware. This work suggests to choose SmartRF05EB as the motherboard for the CC2530 evaluation module.

Figure D.1: SmartRF05EB

There are two main components on the board: the USB controller and the evaluation module. The USB controller and the evaluation module are connected with each other by using SPI, UART or the Debug Interface. The USB controller can access the UART RS232 interface, LCD, one LED, joystick and one button (USB button). The evaluation module can access the LCD, serial flash, four LEDs, 2 buttons, joystick and UART RS232 interface.

Appendix E

Test Setup

In order to test the mDNS design, this work provides a test environment. A simple one hop 3-node example is recommended to be used. It means there should be three CC2530EM devices and the corresponding equipment included in the test case. All the nodes should be within one hop distance from each other. Routers are not considered in the test.

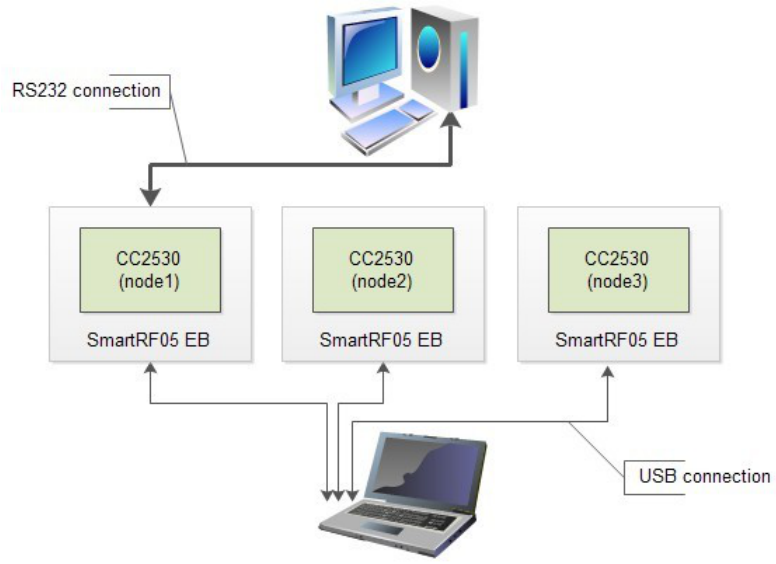


Figure E.1: Test for evaluation setup for the mDNS design

As shown in Figure E.1, a RS-232 cable is used to connect the evaluation board with the PC during the test. All the information which is needed to be tested can be shown on the PC monitor. The PC is used to monitor the incoming and outgoing information of the node1. The laptop is used to supply

the power for the SmartRF05EB and CC2530EM devices. As an alternative option, the SmartRF Protocol Packet Sniffer[37] can also be used to display RF packets captured with a listening RF device, such as node1.

As introduced in Chapter 5, the mDNS API can be called by the user applications. This work recommends to call the mDNS API in the user section which is shown in subsection 5.9.3.2 for the test cases. With the monitoring of incoming and outgoing mDNS messages, the mDNS design can be tested.

This work does not provide the detailed test cases for the designed mDNS Library. It is recommended to test the design from two aspects: (1) the startup and (2) the name resolution. During the startup procedure, whether the mDNS service starts successfully could be tested. As indicated in section 4.4, the Probing and Announcing steps should be tested properly, and Figure 4.4 could be used as the test sequence. The name conflict detection should be involved in the startup test case. During the name resolution procedure, whether the mDNS service works properly could be tested. Figure 5.2, 5.3, 5.5, 5.6 could be used as the test sequence for the whole name resolution process. Notice, in a full test, the interoperability of the mDNS Library and a library like Bonjour or Avahi should be tested as well.

Nomenclature

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) is a set of standards defined by Internet Engineering Task Force (IETF), which creates and maintains all core Internet standards and the architecture work. It was developed to enable the wireless network of embedded devices using simplified IPv6 functionalities, which take into account solve the limitation of embedded devices such as low power, small memory and limited bandwidth.
Ad-hoc	It is a decentralized wireless network. Each node of an ad-hoc network behaves as an end-point as well as a router.
CC2530	CC2530 is a system-on-chip solution tailored for the IEEE 802.15.4, Zig-Bee, ZigBee RF4CE and Smart Energy applications.
Contiki OS	Contiki is an open source, highly portable, multi-tasking operating system for memory-efficient network embedded systems and wireless sensor networks.
DNS	The Domain Name System (DNS) is a hierarchical naming system built on distributed databases for computers, services, or any resource connected to the Internet or a private network.
DNS SRV	It is one kind of resource records of DNS which is used for locating specific services in a specific domain.

Fully Qualified Domain Name (FQDN)	A fully qualified domain name (FQDN), sometimes also referred as an absolute domain name, is a domain name that specifies its exact location in the tree hierarchy of the Domain Name System (DNS).To be specific, it is the complete domain name for a specific computer, or host, on the Internet
IEEE 802.15.4 standard	The IEEE 802.15.4 standard specifies the physical layer (PHY) and the media access control (MAC) for low-rate wireless personal area networks.It is a low data-rate standard especially for low power embedded devices with a limited resource budget.
IPv6	IPv6 is a version of Internet Protocol (IP). It is responsible for routing packets and it is developed to resolve limitations of IPv4, such as the limited address space.
Maximum Transmission Unit (MTU)	In computer networking, the maximum transmission unit (MTU) of a communications protocol of a layer is the size (in bytes) of the largest protocol data unit that the layer can pass onwards.
mDNS Library	This work combines the name server with the resolver into a single library, named mDNS Library.
Multicast DNS	Multicast DNS is a technology using the traditional DNS programming interface, packet formats and operating semantics, to replace the function of the conventional DNS server, especially in small area networks. It is a promising technology chosen to implement service discovery for wireless ad-hoc networks.
Name Server	A name server is a server program which hold information about the domain tree's structure.

Personal Area Network (PAN)	The PAN is used for interconnecting devices around an individual person's workspace.
Protothreads	Protothreads are lightweight stackless threads designed especially for the memory constrained system.
Querier	The functional part of the DNS/mDNS generates the DNS/mDNS query.
Resolver	The user programs access name servers through standard programs called resolvers.
Resource Records (RRs)	RRs hold the information for the corresponding domain and all the information stored in the zone files of the DNS.
Responder	The functional part of the DNS/mDNS generates the DNS/mDNS response.
Service discovery	It is an important addition to ad-hoc networks since it enables sharing of functionality and resources and makes it possible to run distributed applications without the need for manual installation and maintenance of a service database.

Bibliography

[1] S. Cheshire and M. Krochmal. Internet-Draft: Multicast DNS. `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt`, Feb 14, 2011.

[2] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, November 1987.

[3] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, November 1987.

[4] S. Cheshire and M. Krochmal. Internet-Draft: DNS-Based Service Discovery. `http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt`, Feb 27, 2011.

[5] A. Gulbrandsen, P. Vixie and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Febrary, 2000.

[6] S. Cheshire. Zero Configuration Networking (Zeroconf). `http://www.zeroconf.org/`, September, 2011.

[7] Olafur Gudmundsson, Andrew Sullivan. DNS Extensions (dnsext). `http://datatracker.ietf.org/wg/dnsext/charter/`, November, 2011.

[8] S. Cheshire. Multicast DNS. `http://www.multicastdns.org/`, September, 2011.

[9] Internet Systems Consortium. BIND 9 Administrator Reference Manual. `http://www.bind9.net/arm97.pdf`, 2010.

[10] Zach Shelby and Carsten Bormann. 6LoWPAN: The Wireless Embedded Internet. A John Wiley and Sons, Ltd, Publication, 2009.

[11] Atmel. IEEE 802.15.4 MAC - User Guide. `http://www.atmel.com/dyn/resources/prod_documents/doc5182.pdf`. Sep, 2006.

[12] Günter Obiltschnig. Automatic Configuration and Service Discovery for Networked Smart Devices. Electronica Embedded Conference Munich, 2006.

[13] S. Thomson, T. Narten and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862, September 2007.

[14] G. Montenegro, N. Kushalnagar, J. Hui and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, September 2007.

[15] Adam Dunkels. The Contiki OS: The Operating System for the Internet of Things. `http://www.sics.se/contiki/`, May, 2011

[16] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O'Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, Adam Dunkels. Poster Abstract: Making Sensor Networks IPv6 Ready. Nov, 2008.

[17] TI. A True System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee Applications. `http://focus.ti.com/lit/ds/symlink/cc2530.pdf`. Feb, 2011.

[18] P. Thubert. Internet-Draft: Compression Format for IPv6 Datagrams in Low Power and Lossy Networks. `http://tools.ietf.org/pdf/draft-ietf-6lowpan-hc-15.pdf`, February 24, 2011.

[19] S. Deering, R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December, 1998.

[20] S. C. Ergen. IEEE 802.15.4 Summary, Technical Report, Advanced Technology Lab of National Semiconductor, August 2004.

[21] B. Patil, F. Xia and B. Sarikaya. Transmission of IPv6 via the IPv6 Convergence Sublayer over IEEE 802.16 Networks. RFC 5121, February 2008.

[22] J. Postel. User Datagram Protocol. RFC 768, August, 1980.

[23] Apple Inc. Introduction to Bonjour Overview. `http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Introduction.html#//apple_ref/doc/uid/10000119i`, March, 2011.

[24] Lennart Poettering, Trent Lloyd, Sjoerd Simons. Avahi. `http://avahi.org/`, April, 2011.

[25] Adam Dunkels. Contiki - Crash Course. `http://www.ee.kth.se/~mikaelj/wsn_course/contiki-course-kth-9oct2008-draft.pdf`, October, 2008.

[26] IAR Systems AB. IAR Embedded Workbench ® IDE User Guide. `http://supp.iar.com/FilesPublic/UPDINFO/004919/EWSH_UserGuide.pdf`, Febrary, 2010.

[27] TI. SmartRF05 Evaluation Board User's Guide. `http://www.ti.com/lit/ug/swru210a/swru210a.pdf`, 2010.

[28] John D. Day and Hubert Zimmermann. The OS1 Reference Model. Proceedings of the IEEE, 1983.

[29] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, March, 1997.

[30] Jonathan B. Postel. Simple Mail Transfer Protocol. RFC 821, August, 1982.

[31] R. Fielding, J. Gettys, J. Mogul, L. Masinter, P. Leach, T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June, 1999.

[32] Jonathan Hui, David Culler, Berkeley Samita Chakrabarti. 6LoWPAN: Incorporating IEEE 802.15.4 into the IP architecture. Internet Protocol for Smart Objects (IPSO) Alliance White Paper, January, 2009.

[33] Altera Corporation. Direct Sequence Spread Spectrum (DSSS) Modem Reference Design. Altera Application Note 289, April, 2003.

[34] Jin-Shyan Lee. An Experiment on Performance Study of IEEE 802.15.4 Wireless Networks. IEEE International Conference on Emerging Technologies and Factory Automation, September, 2005.

[35] A. Conta, Lucent, S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 2463, December, 1998.

[36] Tanır Özçelebi. Computer Networks 2IC15Application Layer. TU/e Computer Science, System Architecture and Networking Slides, April, 2011.

[37] TI. SmartRF™ Packet Sniffer User Manual. `http://www.ti.com/lit/ug/swru187f/swru187f.pdf`, 2010.