Efficient IPv6 Neighbor Discovery in Wireless Environment

Dragoş Neagoe & Antonios Pateas
wir14dne@student.lu.se & wir14apa@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisors:
Jens A Andersson (EIT LTH)
Stefan Höst (EIT LTH)
Samita Chakrabarti (Ericsson AB, San Jose)
Jaume Rius I Riu (Ericsson AB, Stockholm)

Examiner:
Maria Kihl (EIT LTH)

November 20, 2016

Abstract

As the address space of IPv4 is being depleted with the development of IoT (Internet Of Things), there is an increasing need for permanent transition to the IPv6 protocol as soon as possible. Nowadays, many 3GPP (3rd Generation Partnership Project) Networks have implemented or will implement IPv6 in the near future for Internet access. These networks will also use NDP (Neighbor Discovery Protocol), which is the IPv6 tailored version of ARP (Address Resolution Protocol). The protocol is responsible for address auto-configuration, maintaining lists of all neighbors connected to a network, verifying if they are still reachable, managing prefixes and duplicate address detection.

The protocol is defined in RFC 4861 and although it works fine for wired connected devices, it has been proven highly inefficient in terms of battery lifetime saving, when wireless networks came to the market and its use increased tremendously. This thesis work is a continuation of a previous master thesis and complements the work done previously by showing how the solutions suggested in the new draft can be implemented at the router and host side and practically confirms the previous results of the theoretical analysis through simulation scenarios of sleep and wake-up of the nodes, performed in OMNeT++. Subsequently, the scalability of the system as a whole was analyzed with a simulation model containing a range of hosts from 1 to 100, and shows it can operate efficiently on a larger scale, reducing multicast messaging by almost 100%, presumably saving their battery power.

Acknowledgments

Popular Science Report

The introduction and rise of Internet of Things (IoT), and the use of more and more wireless devices in the communication between users, has depleted the available addresses of IPv4. The introduction of the new IPv6 protocol solves the address depletion problem, but on the other hand, many of the existing protocols have to be redesigned.

This thesis is based on RFC 4861's NDP (Neighbor Discovery Protocol for IPv6 Networks, the equivalent protocol of ARP (Address Resolution Protocol) for IPv4 Networks. Like ARP, NDP is used in all Networks, wired or wireless, and it's main feature is to check and update periodically the state of the Network, provide L2 addresses to hosts in the same Network and verify their reachability.

While wired devices experience no issues regarding power supply, as they are constantly hooked to a power source and rarely experience network failures, wireless devices have limited power, as they rely on battery lifetime. This is also the case of machines running NDP - the protocol relies on periodic exchange of multicast ICMPv6 (Internet Control Message Protocol version 6) control messages, creating unnecessary traffic overhead in the Network, as all hosts in a Network would receive those messages, regardless if they are meant for them or not. As a general working mode of a battery operated device, one enters predefined sleeping cycles (stand-by), which are designed by each manufacturer in different ways. Therefore, multicast signaling inside Networks disrupt those sleeping cycles, causing increased battery consumption, as a result of more required processing power and more consumed bandwidth.

RFC 6775, together with [3], propose updates to NDP, which would solve the problems mentioned above. The major update is that each host can update the router about its state, by sending unicast messages, without involving the other hosts in the Network. The router, instead of sending periodic control messages

to every host, it sends control messages to each host separately in specific time intervals. Only when a major change occurs in the Network, for instance an addition of a new host, or when a host leaves the Network, multicast messages are sent to every host to update their state. Together with the establishment of unicast signaling, a new method of address registration is introduced in the documents cited above, called Address Registration Option. This registration method is fully compatible with the two standard mechanism which provide the L3 addresses to hosts - Stateless Address Autoconfiguration (SLAAC) and Dynamic Host Configuration Protocol (DHCP).

The previous thesis work took the first steps in implementing the proposed protocol changes, by investigating functions inside RADVD - the Router Advertisement Daemon, run on all routers and responsible for sending the multicast periodic control messages to the hosts (Router Advertisements). A full implementation of the proposed changes require covering both sides of the Network, i.e Host and Router. While RADVD is handling the Router side, the implementation at the Host side needs to be done inside the Linux Kernel. In this thesis work, the RADVD implementation was completed and possible implementation methods were shown inside the Linux Kernel. Due to the overall complexity of the Linux Kernel, while the proposed code could cover most aspects from RFC 6775, it wasn't possible to test it, in order to conclude how much workload is left.

Simulations took place to compare the two protocols and verify, in what extend these proposed changes can potentially improve battery lifetime. So, sleep and wake up scenario was tested in same time intervals in order to observe Network traffic. The goal was to have a decrease in control messages in the case where the suggested changes were applied. Different number of hosts were selected to see if these changes can be applied to larger network. In both cases, the best case scenario was tested and parameters which would normally hinder network performance were neglected. This decision was made to reduce the complexity of the Network as well. The results of the simulations indicated that there could be a decrease in control messages and the Network seems stable and scalable as number of host increases.

Contents

List of Acronyms

- 3GPP - Third Generation Partnership Project
- 6CO - 6LoWPAN Context Option
- ABRO - Authoritative Border Router Option
- ARO - Address Registration Option
- CIDR - Classless Inter-Domain Routing
- DAD - Duplicate Address Detection
- DHCPv6 - Dynamic Host Configuration Protocol version 6
- EAH - Efficiency-Aware Host
- EPS - Evolved Packet System
- ICMP - Internet Control Message Protocol
- ICMPv6 - Internet Control Message Protocol version 6
- IETF - Internet Engineering Task Force
- IP - Internet Protocol
- IPsec - IP security
- IPv6 - Internet Protocol version 6
- IoT - Internet of Things
- M2M - Machine to Machine
- MAC - Media Access Control
- MTU - Maximum Transmission Unit
- NA - Neighbor Advertisement

- NCE - Neighbor Cache Entry
- ND - Neighbor Discovery
- NDP - Neighbor Discovery Protocol
- NEAR - IPv6 ND-efficiency-aware Router
- NS - Neighbor Solicitation
- NUD - Neighbor Unreachability Detection
- OSI - Open Systems Interconnection
- PRNG - Pseudo Random Number Generator
- RA - Router Advertisement
- RADVD - Routing Advertisement Daemon
- RS - Router Solicitation
- SLAAC - Stateless Address Autoconfiguration
- SLLA - Source Link Layer Address
- TID - Transaction ID
- UIID - Unique Interface Identifier

List of Figures

List of Tables

Chapter 1

Introduction

As the number of devices that are connected to the Internet rapidly grows, larger networks are needed in order to be able to connect all devices. The rise of IoT (Internet of Things) sees the interaction of all these devices, which will create smarter environments. It is expected that between 26 and 50 billion devices will be connected to the Internet by 2020 [1] and that IoT will together connect trillions of devices, as it can be seen in Figure 1.1.
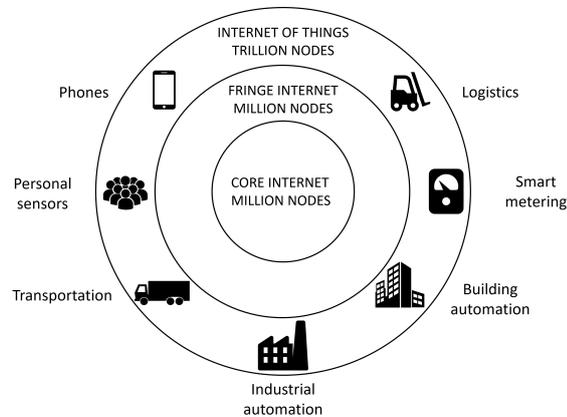
Figure 1.1: Internet of Things [4]

The deployment of IPv6 over IPv4 is a common topic nowadays, as the IPv4 address space is more or less close to depletion. The address availability in IPv6 is incredibly vast and can host billions of devices. IPv4 permits broadcasting messages only, whereas IPv6 supports multicast as well. Moreover, IPv4 requires manual configurations of addresses or stateful address configuration (Dynamic Host Configuration Protocol - DHCP), which makes it undesirable for the IoT network [2].

In contrast, IPv6 allows for both stateless (Stateless Address Autoconfiguration - SLAAC) or stateful (Dynamic Host Configuration Protocol version 6 - DHCPv6) address configuration. Table 1.1 summarizes some of the main differences between IPv4 and IPv6.

	IPv4	IPv6
Address space	2^{32} addresses	2^{128} addresses
Status of address space	close to depletion	plenty
Type of communication	broadcast / multicast	multicast
Address configuration	stateful only	stateful / stateless
Address configuration protocols	DHCP	SLAAC with ICMPv6 or DHCPv6
Encryption	optional	required - IPsec

Table 1.1: IPv4 vs. IPv6

This thesis work focuses on the Neighbor Discovery Protocol (NDP), which is the IPv6 alternative of the Address Resolution Protocol (ARP). It operates in the Link Layer of the Open Systems Interconnection (OSI) model and it represents one of the core protocols required in IPv6 networks, being used to sense other devices in a network. It uses ICMPv6 control packets to perform functions similar to the equivalent IPv4 ARP protocol.

1.1 Background

When IoT and M2M (Machine to Machine) were introduced, networks have become more complex and so requirements for NDP started to change. RFC 4861 defines NDP and obsoletes RFC 2461. It features the behavior of the node in the network, how it interacts with its neighbors on the other end of the link and how identifies them. This was done by sending ICMP messages for IPv4 protocol in the ARP case and by sending ICMPv6 messages in the NDP case. Most of those messages are sent by NDP periodically and multicast. The ICMPv6 protocol is defined in RFC 4443 and it's main purpose is to perform network diagnosis. In this thesis, only the specific ICMPv6 messages used by NDP are discussed.

The protocol operates simultaneously on two sides, one taking care of the hosts inside the network and the other regarding the operation of the routers. The differences between the two will be analyzed in Chapter 3. The protocols main functions are:

- Discover other nodes inside the network;

- Determine nodes prefixes and parameters;

- Resolve nodes link-layer address;

- Verify the reachability of other nodes through Neighbor Unreachability Detection (NUD);

- Verify duplicate addresses through Duplicate Address Detection (DAD);

- Redirect nodes with better route choices.

A more detailed description of NDP, as well as a new proposed working mechanism for NDP will be discussed in Chapter 2.

The IoT infrastructure consists of many battery operated devices, therefore the key factors which contribute to the optimization of such a device are to reduce the used resources, to decrease the required processing power and to increase the life time of the battery. Although NDP performs well in wired networks, where power supply is not a problem, in wireless networks where the connected devices operate with batteries, it has been proven to be inefficient ([3], [4], [5], [6], [7]). The ICMPv6 packets generated by NDP contain unnecessary periodic control information, which increase network traffic, makes the processors work overtime and thus depletes the battery lifetime faster. The work described in [3] suggests changes to optimize the NDP for wireless networks and reduce this unnecessary information as much as possible.

1.2   Problem definition

The use of the protocol described in [8] is best suited for wired instead of wireless networks and the reason presented in the following articles.

This inefficiency of NDP has been stated in [5] and [6]. The problem described in [5] presents the EPS (Evolved Packet System) network impact of IoT/M2M IPv6 connectivity, in which network resources are used in 3rd Generation Partnership Project (3GPP) cellular networks every time periodic multicast messages are received by mobile terminals, which increases processing loads and costs.

In [6], possible issues of using multicasting in some wireless networks including Wi-Fi are presented. One of those issues is that when wireless multicast packets are transmited in a network, the access point acts as a hub: if one host transmits, the rest have to be silent, lowering data rates tremendously.

Most ICMPv6 messages are sent multicast by NDP, meaning that some nodes or parts of networks which are not related to the communication will receive that information as well, even when the operating device is in sleep mode, if this device

is part of the multicast group. In most cases, those multicast groups in IoT and M2M networks are quite large, containing many nodes, so often the multicast messages have nothing to do with the node, but the node has to wake up from the sleep mode, check the packet and determine that it is irrelevant. Moreover, when a node enters or leaves the network, multicast messages are being sent in order for the network to be updated, but as a result, this disturbs the other devices from their sleep cycle, causing unnecessary battery consumption and therefore making RFC 4861 not optimal with today's wireless networks.

RFC 4919 investigates the behavior of NDP in the context of Low Powered Area Networks running the IPv6 Protocol (6LoWPAN). It is meant as a solution from the Internet Engineering Task Force (IETF) to solve some of the problems risen by the introduction of IoT and it targets devices operating in the IEEE 802.15.4-2003 standard. Wireless devices, especially ones operating in low powered networks, are the core of IoT. They are meant to function as "out of the box", needing minimum configuration, having limited features which target specific tasks that are required in a network, as a complement to the existing infrastructure. Some limitations of those devices would be: short range, low bit rate, limited power, memory and energy.

Concerning low powered networks, the goal of RFC 4919 is to reduce packet overhead, bandwidth consumption, decrease the required processing power of the devices and decrease battery consumption. Those can be made possible through the following strategies, without going into details:

- Fragmentation and reassembly of packets
- Header compression
- Address Auto-configuration
- Mesh Routing Protocol
- Implementation Configurations

RFC 6775 introduces later on a specific problem and solution to NDP regarding multicast signaling in 6LoWPANs. It proposes an implementation configuration of NDP which reduces multicast signaling to a minimum. In [3] this problem is researched again, showing that this trend is not limited to 6LoWPAN networks, but that multicast signaling affects all types of wireless networks.

One solution for those problems is offered in [3], which re-uses some implementation configurations presented in RFC 6775 and complements those with other features, which altogether are meant not only for 6LoWPANs, but for all types of wireless networks. A suggestion of a novel method of address registration is presented, in which the routers of the network can be used as core and also convert all the multicast ICMPv6 messages to unicast. This solves the issues in both the above mentioned Internet drafts.

This thesis work complements [11], by offering an implementation method of the Efficient NDP in the RADVD (Router Advertisement Daemon) at the router side, in the Linux Kernel at the host side and and by analyzing a practical simulation model, which generates results comparable to the theoretical analysis done in the previous work. In the next chapters, more information will follow about how ND is performed in the RFC 4861 and its issues. This thesis work's main objective is to see how those implementation changes to NDP affect the network, its functionality and its scalability.

1.3   Previous work

This work is a continuation of the thesis work with title *Analysis of IPv6 Neighbor Discovery for Mobile and Wireless Networks* [11] written by Hariharasudan Vigneswaran and Jeena Rachel John. In their work, the problem of using RFC 4861 in wireless networks was presented and a theoretical analysis of the Legacy NDP by the use of different scenarios took place, measuring the number of RS (Router Solicitation), RA (Router Advertisement), NS (Neighbor Solicitation), NA (Neighbor Advertisement). Then these results were compared theoretically with the Efficient NDP, in order to see what gain would have been accomplished if these modifications were applied to protocol. The IPv6 NDP behavior was analyzed, the scalability of the network, the transient behavior of the network and the number of multicast messages exchanged in the network. Sleep and wake up scenario and its effect was only mentioned, due to the limited time. In this work, a sleep and wake-up scenario is studied, for two test case scenarios, one based on RFC 4861 and one based on [3] with the proposed changes. OMNeT++ is used as a simulation tool, and the outcome of the simulation is to observe how these changes can improve RFC 4861 for wireless devices in terms of saving power. This is discussed further on Chapter 4 and 5. The previous work also investigated the possible implementation of the protocol in RADVD, the Routing Advertising Daemon running on all Unix devices, which is responsible for generating NDP control messages on the router side. A working RADVD implementation will be shown in this thesis work and the host side in the Linux Kernel will be explored, in order to show an implementation method of the proposed modifications as a whole. This will be described in detail in Chapter 3.

1.4   Approach

The implementation part of the protocol was performed in an Unix environment, running Ubuntu 14.04 LTS and the latest version of RADVD (2.14). Most of the investigated code was done in C Programming Language. On the router side, a full analysis of RADVD was done, together with it's functions and libraries, in

order to determine where changes were necessary. The implementation inside the Linux Kernel on the host (client) side was the biggest challenge of this work and is still only partially performed.

To prove the efficiency and scalability of the proposed protocol changes, a practical simulation scenario was created, using the open source software OMNeT++, in which, using C++, two different scenarios were tested and compared, one running the Legacy NDP and one running the Efficient NDP. In both scenarios, the same parameters were used, such as number of hosts and simulation time intervals, in order to observe how much different is the number of the exchanged messages between the router and the hosts in both cases. The number of hosts selected for testing were from 1 to 100, in order to see the behavior of the network as the number of hosts gets larger, hence test its scalability.

1.5   Organization of thesis

This thesis work is carried out by both Dragos and Antonios in collaboration with the Department of Electrical and Information technology (EIT) at Lund University and Ericsson AB, Stockholm. The authors had the goal, which was to evaluate the scalability and energy consumption according to [3], as well as potentially implementing the proposed changes in NDP. Both of the authors have taken active part in all the key steps of the project. Dragos focused mostly on coding and the changes that are required in designing the new Efficient ND protocol. Antonios focused mostly on studying the sleep and wake up scenario, simulating the models for both protocols in OMNeT++ and analyze the results. Due to the limited time, Dragos also participated in programming the simulation environment in OMNeT++ and extracting the results. Chapter 3 was written by Dragos, chapter 4 was written by Antonios, while the rest of the chapters were written by both authors.

Chapter 2

Neighbor Discovery Protocol

This chapter offers the reader some background information about NDP as described in RFC 4861 - Legacy NDP and about the Efficient NDP draft [3] which proposes changes to optimize NDP for wireless devices - Efficient NDP. The first part of this chapter describes the packet types that are exchanged between routers and hosts through NDP and their functions. Then, it describes the types of addresses used by NDP and the role of DAD when a new host wants to enter the network. The next sections refer to suggested changes as they are described in the Efficient NDP draft [3], in order to make the NDP optimal.

2.1 RFC 4861 Review - Legacy NDP

RFC 4861 was introduced to specify the NDP for IPv6. Nodes that use IPv6 are able to determine each other's link-layer addresses and see each other in the network. They are also able to detect routers and if at some point some of their nodes/neighbors leave the network or lose their connection, it is possible to keep reachability information about the paths to their neighbors that are active. The requirement for these functions is the nodes to be at the same link when they use Neighbor Discovery [8].

2.1.1 Overview

In a network, every node, whether it is a router or a host, needs to know all its neighbors on the other end of the link, so it can keep its routing table updated at all times. This process is called ND (Neighbor Discovery). Each node contains a list with all of its neighbors, from which it can distinguish if it is a router or

a host. This list is called a NEC (Neighbor Cache Entry). A node's network ID or prefix is defined while it is trying to get an IP address. RFC 4861 provides following features:

- Router Discovery
- Address Autoconfiguration
- Prefix Discovery
- Address Resolution
- Duplicate Address Detection - DAD
- Neighbor Unreachability Detection - NUD

2.1.2   Neighbor Discovery Protocol

Neighbor Discovery Protocol is a part of ICMPv6 and it contains the following packet types, as it is described in RFC 4861:

- RS - Router Solicitation
- RA - Router Advertisement
- NS - Neighbor Solicitation
- NA - Neighbor Advertisement
- Redirect message

RSs are messages packets sent from a host that enters or reconnects to a network to any router that is on the local area of this network in order to request their presence to be advertised in the network. In other words, the nodes request an RA (Router Advertisement) packet.

RAs are packets that broadcast from the router in response to RS messages. They are also sent periodically, but their time interval may vary. They contain data, such as the address configuration, network prefix, data that can determine if there is another address with the same link and info about the default router.

NSs are packets sent from a node in order to acquire information about a neighbor. They are also used for DAD (Duplicate Address Detection) and for NUD (Neighbor Unreachability Detection) to check if a neighbor can still be reached and get the MAC address of the destination.

NAs are the response packets to NSs. When a node announces link-layer address change, it may send unsolicited NA messages.

The exchange of those messages in NDP is illustrated in Figure 2.1.
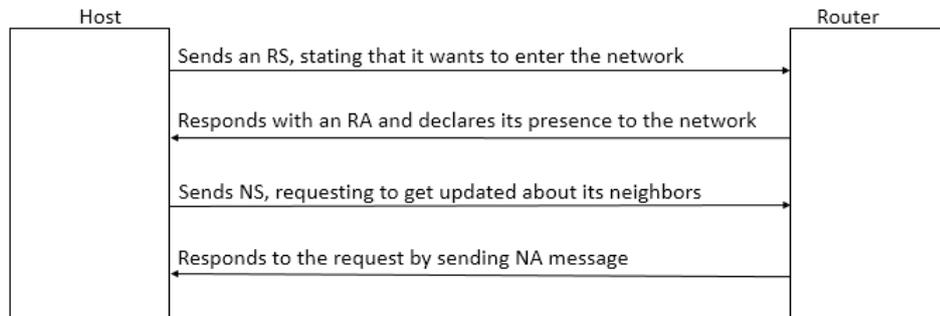
Figure 2.1: Exchange message between host and router

Redirects are messages that are simply used by routers to inform a node about a better first hop for its destination.

2.1.3   Addresses

Neighbor Discovery uses different addresses such as:

- All-nodes multicast address. It is the link-local scope address from which all nodes can be reached, FF02::1.

- All-routers multicast address. It is the link-local scope address from which all routers can be reached, FF02::2.

- Solicited-node multicast address. It is a link-local scope multicast address that is calculated as a function of the solicited address of the target. This function operation is described in detail in [26]. The selection of this function is done, so the difference between the IP addresses is only between the most significant bits.

- Link-local address. All router interfaces are required to have a link-local address. It is a unicast address that is used to reach neighbors and is link-only. Also, according to RFC 4862, host interfaces are required to have a link-local address.

- Unspecified address. It is a reserved address value that indicates that the specific address is missing or it is unknown. It can be used as a source address in some cases, but never as a destination address. Its value is 0:0:0:0:0:0:0:0.

The Neighbor Cache is a log of the recent traffic and it contains all the individual neighbors' entries. The NCE also contains information about the neighbor's reachability, meaning if a neighbor can be reached or not, a flag from which the neighbor origin can be distinguished, whether it is a host or a router and the number of probes that have not replied. The neighbors reachability can be in one of the following states:

- INCOMPLETE: Neighbor Address is still to be determined.

- REACHABLE: The neighbor has been reached very recently.

- STALE: It's not known anymore whether the neighbor can be reached or not. The reachability is not verified until traffic is sent to the neighbor.

- DELAY: As with STALE, it's not known anymore whether the neighbor can be reached or not, so instead of investigating the neighbor immediately, this investigation is being delayed in order to give chance to the upper-layer protocols to provide reachability confirmation.

- PROBE: It's not known anymore whether the neighbor can be reached or not, so NS probes are sent to verify it.

Address resolution is a very essential process of RFC 4861, through which a node can determine the link-layer address of a neighbor, given only its IP address. It is never performed on multicast addresses, but only on on-link addresses and for which the link-local address is known to the sender. The sender first checks its NCE table to see if it contains the destination address. If it is not, a new entry is created which has "INCOMPLETE" status. Before a node starts sending packets to another node, it needs to know the IP or the MAC address of the receiver. So, when a node wants to know about the receiver's address, it sends an NS message to the solicited node multicast group of the receiver. All nodes that belong in the same group, will receive the NS. The receiver in this case, will respond with a NA. If the NS had a source address, which means the sender is also part of the network, the NA will go only to the sender and then the sender and the receiver only can send NS and NA to each other. If the NS didn't have a source address, meaning that the sender wasn't part of the network, then the NA will go to the multicast address of all nodes. Once the sender gets the NA from the destination, its NCE status is updated to REACHABLE.

2.1.4   DAD (Duplicate Address Detection)

DAD is a mechanism that determines if an address that a node wishes to use when entering the network is available, or it is being used by another node. It is performed on unicast addresses only, without playing any role if the addresses were configured manually, stateless or stateful. It can't be performed on anycast

addresses though and any individual unicast address must be tested for uniqueness [16].

To explain this mechanism, an example is given which contains a network with a router and two hosts, of which one is already in the network and the second one just joined the network. The new host will send an RS, asking to enter the network and as an answer will receive RA from the router and get the network prefix. Having the prefix, the host can then acquire its IP address with the use of SLAAC. SLAAC is a feature which allows hosts to pick their IP address once they know the network prefix. So, if the router picks an address which is already taken by the other host of the network, it will send NS to the solicited node multicast group of this selected address and check if there is someone already uses this address.
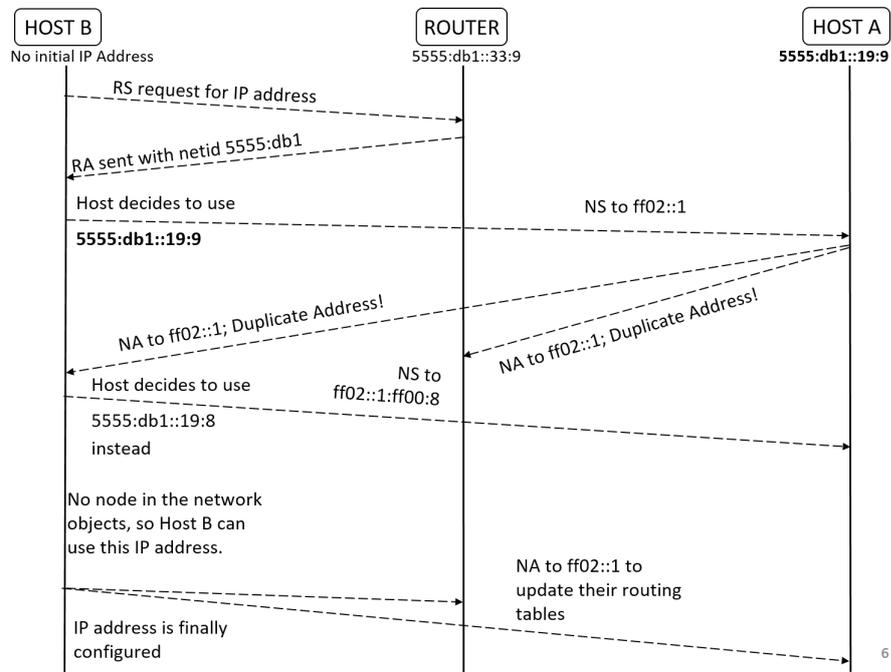
Figure 2.2: Illustration of DAD

The first host will receive the NS, but since the second host has no IP address yet, the first will reply with NA to all nodes of this multicast address group. The new host will receive the NA from the other host and it will then know that the selected address is already taken. So, then it changes the network suffix and tries again with a new address. The same process continues, but since no one uses the new address, the host configures itself with the new address and sends NA to the other hosts of the network to update them and they will update their NCE (Neighbor Cache Entries) as well. The new host has now to start address resolution process first, in order to update its NCE, before it starts exchanging packets with the

other hosts. Figure 2.2 illustrates DAD.

2.1.5   Neighbor Unreachability Detection

Neighbor Unreachability Detection can be used for all paths for communication between hosts and neighbors, such as host-to-host, host-to-router, and router-to-host. It is a mechanism that may recover the communication between neighbors and is performed only for neighbors to which unicast packets are sent. During the communication between neighbors, there is a chance that the connection between them will fail, hence communications will interrupt. If the failure occurs to the path, there is a chance to recover the communication. In this case, nodes will start checking the reachability status of their neighbors. If, however, the failure occurs to the destination, the recovery is not possible and the communication will fail.

In case of a path failure, it's important how the neighbor is being used. For example, if the neighbor is a router, an appropriate path recovery would be to skip this router and pick an alternative one. Address resolution should be redone if the neighbor is the final destination.

2.2   Optimization on RFC 4861 - Efficient NDP

NDP protocol's way of functioning was proven to be inefficient in wireless networks, due to the repetitive use of multicast signaling which disrupts sleeping cycles of mobile operated devices, consuming battery life. RFC 6775 introduces some modifications to RFC 4861, in the context of Low Power Radio Networks (6LoWPAN), which should significantly reduce the usage of multicast signaling and Efficient NDP draft [3] is a generalisation of the RFC 6775 [4], for all type of networks.

2.2.1   Goals and Requirements

The main goal of this draft is to reduce the use of multicast signaling, by removing the need of multicast NS messages and periodic RS/RA messages.

The requirements are as following:

- Add more centralized control to the routers, so that hosts register themselves to routers;

- Introduce a new working mechanism for DAD, so that the hosts do not require to defend their address all the time; This will be done with a new introduced option from RFC6775, which is called Address Registration Option and having SLLA enabled all the time;

- Have the protocol compatible with the Legacy NDP, and make it possible to function in efficient mode, legacy mode or mixed mode. The mixed mode is out of the scope of this thesis;

- The new Address Registration mechanism should function together with SLAAC (Stateless Address Auto-configuration) and DHCPv6 (Dynamic Host Configuration Protocol Version 6);

- Introduce a state loss mechanism (Router Epoch mechanism) to allow routers to rebuild their cache entries in case of power failure. This option was left out of the thesis, however it's a possible future work implementation.

2.2.2 Proposed protocol changes

The E Flag

The E Flag is a configuration knob which signals the devices in the network that NDP is working in the efficient mode. This flag is to be implemented inside the RA messages, in the flag field. As of [19], until so far, the last 2 bits of the flag field are still reserved for further use, therefore the second last bit can be used for the E Flag. This can be observed in Figure 2.3.
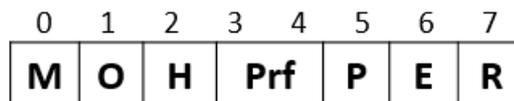
Figure 2.3: Flags field of Router Advertisements [19]

Once a RA with an enabled E Flag is received, inside that network, routers and hosts work in the Efficient NDP mode and are called NEARs (IPv6 ND-efficiency-aware Router) and EAHs (Efficient Aware Hosts). The E Flag was implemented in this thesis inside RADVD at the router side, which can be viewed in Chapter 3.2.1.

2.2.3 Host and Router initialization

During start up, the router becomes NEAR by enabling the E flag, then it sends first RA to hosts. Hosts send their first RS as multicast, in which they include

their Layer 2 address (SLLA). The SLLA will be updated in the router NCE and used for the next unicast reply to the host. Unicast messaging is now possible. This process is depicted in Figure 2.4.
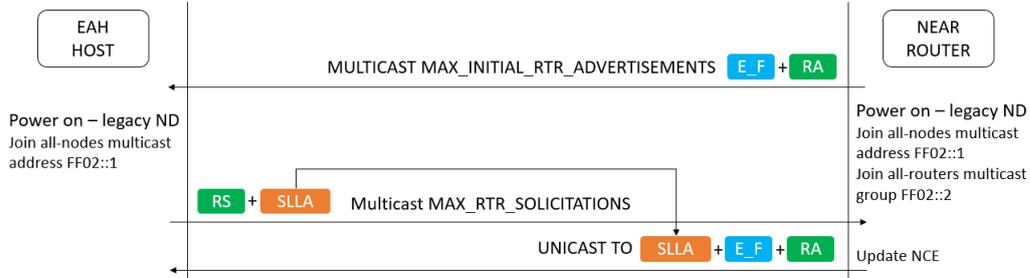
Figure 2.4: Host and Router initialization

2.2.4   Address Registration and ARO

Address Registration is supposed to work in a different manner than in the Legacy NDP. [4] introduces 3 new router discovery options - Address Registration Option (ARO - Figure 2.5), 6LoWPAN Context Option (6CO) and Authoritative Border Router Option (ABRO). The two latest, while out of scope of this thesis, they could be taken into consideration for further implementation of RFC 6775.
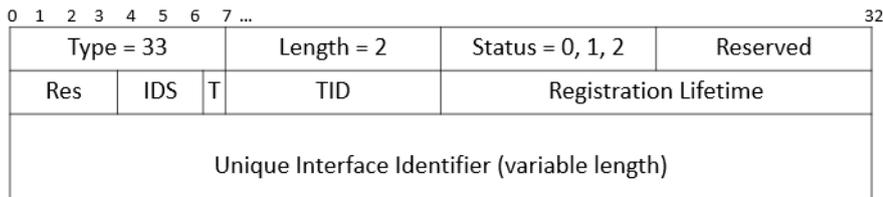
Figure 2.5: Address Registration Option [4]

Hosts would register to routers through unicast NS messages containing ARO and SLLA, instead of relying of multicast DAD messages. Hosts would update their NCE with the new Address Registrars, storing the SLLAs from the UIID field, together with the Registration Lifetime and TID fields, which are used for maintaining registrations and determining which registration is more recent. This would take place while the host is performing NUD. A NCE prototype can be seen in Table 2.1. The Registration Lifetime parameter is the same as the Default Router Lifetime parameter contained in RA messages, which is defined in RFC 4861 between `MaxRtrAdvInterval` and 9000 seconds.

UIID	TID	SLLA	Registration Lifetime

Table 2.1: NCE Prototype

Routers, receiving the NS with ARO and SLLA, update their NCEs and reply with unicast NAs, sent to the SLLA newly learned. Both the NSs and NAs contain the ARO option. When a NA is received on a host from a router, the router has filled in the status field, indicating if the registration was successful or not. This process is illustrated in Figure 2.6.
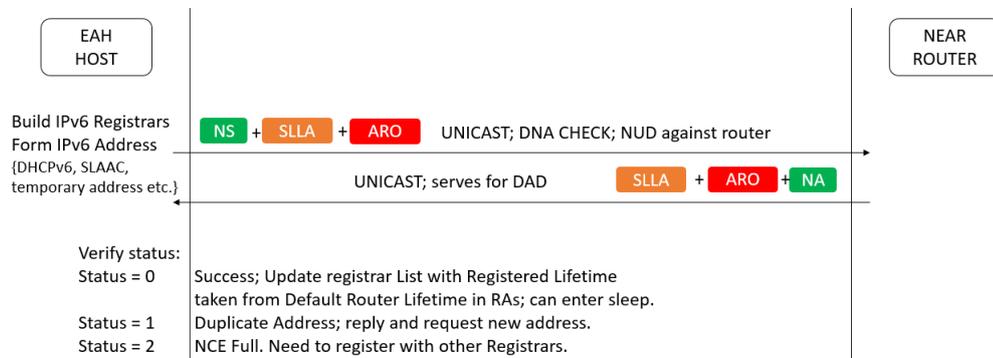
Figure 2.6: Address Registration Process

The implementation of ARO inside the Linux Kernel was researched in this thesis and an implementation method inside the Linux Kernel was proposed in Chapter 3.4.1.

2.2.5 Registration refresh and sleepy hosts

When a registration is about to expire, a host can unicast a RS to request new information. Similarly, when a host enters sleeping mode, it can wake up before it's registration lifetime expires, send an unicast NS and the router will refresh it's registration with a unicast NA. This enables the hosts to enter sleeping cycles and perform undisturbed, instead of having to defend their address all the time as it was with DAD in Legacy NDP (See Figure 2.7). This process was suggested for implementation inside the Linux Kernel, together with the ARO, in Chapter 3.4.1.
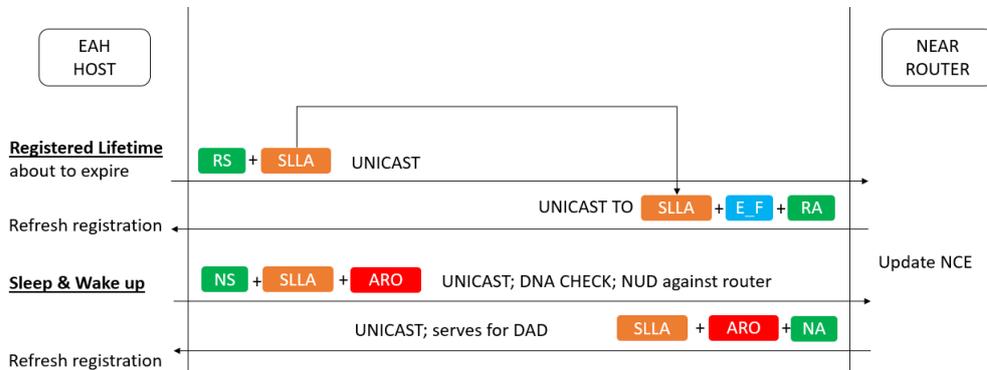
Figure 2.7: Refresh of registration

2.2.6 Router epoch and RAO

The Efficient NDP draft suggests implementing a Router Epoch mechanism, which would handle re-registrations, in case of power loss or failure. This would be done through a new option called RAO (Registrar Address Option), attached to the RA messages sent by the routers. It also covers the case when different registrars exist in the network, in which the routers would inform the hosts where to register. The process is depicted in Figure 2.8.
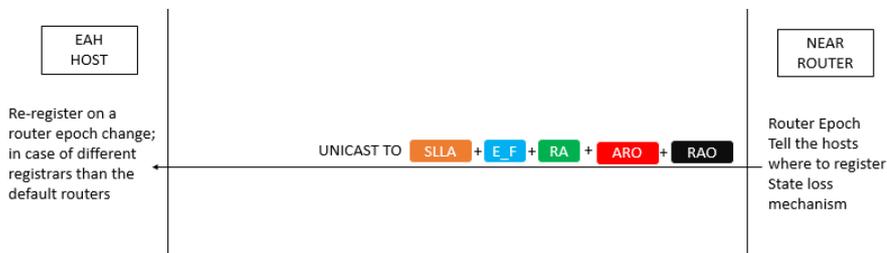
Figure 2.8: Router Epoch

This option was not investigated in this thesis due to lack of time, however it can be a topic for future work.

Chapter 3

Code Implementation

The suggested changes of NDP, mentioned in Chapter 2.2, introduce several implementation goals, in order to achieve the Efficient NDP. Those goals are depicted in Figure 3.1. 6LoWPAN support has already been added to RADVD by Varka
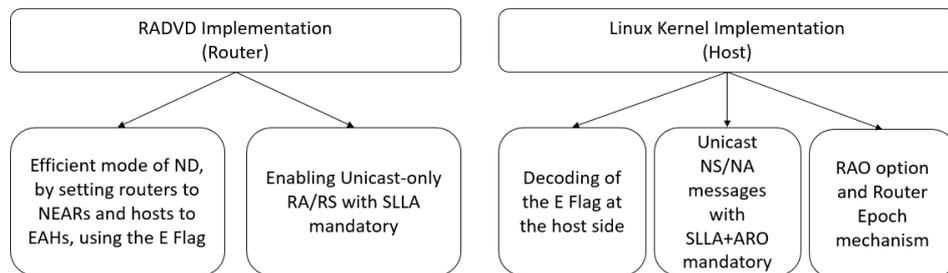
Figure 3.1: Refresh of registration

Bhadram in the Release 1.10.0 of RADVD on 18th of March 2014, which now supports 6Co (6LoWPAN Context Option) and ABRO (Authoritative Border Router Option) options, options attached to RA/RS messages. The third new option introduced in [4] is ARO, the one that this thesis is also focusing on. This has to be implemented in the Linux Kernel, as it is attached to NS and NA messages. ARO doesn't have to be implemented in RADVD, as RADVD is only handling RA and RSs.

This chapter is structured in two parts, a RADVD and a Linux Kernel part. In Section 3.1, a review of RADVD was written, which would help understanding Section 3.2, the implementation section of RADVD. Using the same principle, Section 3.3 is offering a Linux Kernel Review and Section 3.4 is describing the detailed implementation process of Efficient NDP.

17

3.1   Router Advertisement Daemon RADVD Overview

RADVD is the Router Advertisement Daemon used by Linux machines which operate as routers inside networks. RADVD is responsible of sending Router Advertisements and answering to Router Solicitations sent by hosts which require Stateless Address Auto-configuration (SLAAC). By default, RADVD multicasts RAs periodically inside the network. The source code of RADVD is open-source and can be obtained from [17].

RADVD is installed on Linux machines by running the following command:
```
#apt-get install radvd
```

When performing modifications of the RADVD source code, the code has to be build and installed on the Linux machine. This is performed by running following commands inside the RADVD folder:
```
#./configure
#make
#make install
```

A complete guide of how to configure RADVD can be found in the previous thesis work [11], in the Appendix.

Following source code and header files of RADVD are reviewed below, which are afterwards involved in the implementation of the E Flag, shown in Chapter 3.2.1:

$$
\text{RADVD} \begin{cases} \text{Defaults.h} \\ \text{Device-common.c} \\ \text{Device-linux.c} \\ \text{Gram.y} \\ \text{Interface.c} \\ \text{Process.c} \\ \text{Radvd.c} \\ \text{Radvdump.c} \\ \text{Radvd.conf.5.man} \\ \text{Scanner.l} \\ \text{Send.c} \end{cases}
$$

Defaults.h

This header file contains the definitions of key parameters of the RADVD daemon, with their default values, such as MaxRtrAdvInterval, MAX_INITIAL_RTR_ _ADVERTISEMENTS, default values of flags (AdvOtherConfigFlag , AdvManagedFlag etc). The E flag was defined in this header, in Chapter 3.2.1.

Device-common.c

The function `setup_linklocal_addr` is used to store a found SLLA (ifa_addr) from an interface, to the `iface` pointer address. The function `check_device` performs different checkups, such as if the RADVD daemon is running in the UnicastOnly mode or not.

Device_linux.c

Inside `update_defice_info()`, the type of used hardware is determined, together with the assigned prefix length for the specific hardware. For example, in 6LoW-PAN networks (`ARPHRD_IEEE802154`), the prefix length is 64 (EUI-64).

Through `setup_allrouters_membership()`, the router joins the all-routers multicast group (FF02::2). The following structure, `struct AdvPrefix *prefix = iface->AdvPrefixList`, determines the prefix length of the Source Link Layer Address. A checkup of IPv6 forwarding is done through the `check_ip6_forwarding()` method. Forwarding has to be enabled on the Linux machine, in order to have RADVD running. More information can be found about this in the Appendix of [11].

Gram.y

A .y file is a grammar input file, read and parsed by Bison, producing C language functions. This file consists of C declarations, Bison declarations, Grammar rules and additional C codes [18]. Inside this file, parameters of RADVD are defined in Chapter 3.2.1 through so called tokens and their functionality is being set up - for example, a flag would be declared as a switch which can be turned on or off, or a timer is declared as number. The parser file calls for the lexical analyzer file, Scanner.l in this case, explained in the same section, below.

Interface.c

Interface initialization is performed in this source code, and all parameters as members of the interface are defined. Example: The Home Agent flag is stored at the interface pointer, inside the RA header: `iface->ra_header_info.AdvHomeAgentFlag = DFLT_AdvHomeAgentFlag;` The parameters used by RADVD inside RAs are attached to each particular interface. The function `check_iface()` performs a series of checkups of parameters for the used interface, to see if parameters receive correct values.

Process.c

This file contain functions which process Router Advertisements and Router Solicitation messages, checks the size of the packets, comparing to the ICMPv6 header, checks if the Router Advertisements received contain or not a non-linklocal source address (`IN6_IS_ADDR_LINKLOCAL`), or if other parameters contained in the received messages are valid or not. Lines 154-170 verify when the last RA was sent (`MinDelayBetweenRAs` - defined in radvd.h, present as an adjustable variable in the configuration file of radvd). If a RA was recently sent, the next RA is rescheduled with the reschedule_iface() function. This function is defined in interface.c and schedules RAs according to `MAX_INITIAL_RTR_ADVERT_INTERVAL`. An addition for the E Flag was done in Chapter 3.2.1.

Radvd.c

In summary, this source file handles the following features of NDP:

- Handles the process id (PID) of the RADVD daemon; The PID file is located in the Linux Kernel at /var/run/radvd.pid;

- Defines different options for running the RADVD;

- Creates the RADVD daemon with system functions;

- Checks for IP forwarding, user permissions, interfaces.

Radvdump.c

This file is responsible for printing out a description of the content of RAs sent by RADVD, similar to a tcpdump. It captures the packets being sent, showing which options were configured on each interface. The E Flag option was added here, in Chapter 3.2.1.

Radvd.conf.5.man

The Radvdump file described above takes input of configured parameters or configuration knobs from this file. Every parameter or configuration knob is also described here, with details such as format, if it can take a value of true or false (enabled or disabled), or if it is true or false by default. The E Flag configuration knob was added here, in Chapter 3.2.1.

Scanner.l

This is a Flex file - lexical analyzer. Together with the grammar file Gram.y, those files are parsed by Flex and Bison respectively, and a C file is being produced. The Flex file is used as an input to the Bison file. A lexical analyzer file consists of definitions, rules and user routines, three sections delimited in the file through the '%%' symbol. More details about this type of file can be read at [27] and the additions done to Scanner.l, together with Gram.y, are to be found in Chapter 3.2.1.

Send.c

The file send.c contains functions which are used for creating the RAs. The function `add_ra_header()` creates the header of the RAs, adding all defined fileds from RFC4861 and indexing following options:

- add_prefix() - defined options for prefix are added
- add_route() - IPv6 CIDR (Classless Inter-Domain Routing) routes are advertised to clients
- add_rdnss() - Recursive DNS Server (RFC 5006)
- add_dnssl() - DNS Search List (RFC 6106) - DNS suffixes domain names
- add_sllao() - The Source Link Layer Address (SLLA) is added
- add_lowpanco() and add_abro() are functions supporting 6LowPAN - RFC 6775

The function `build_ra()` constructs the RA by adding all of the above options together. Through `send_ra()`, the RA is then sent to the all-routers multicast address FF02::2 with the variable `all_hosts_addr`. The RA header was updated with the E Flag in Chapter 3.2.1.

This concludes the general overview of RADVD's files and functions, and now the detailed implementation performed in RADVD follows.

3.2 Router Advertisement Daemon RADVD Implementation

The Efficient Aware mode of the Neighbor Discovery Protocol implies offering routers in a network a more centralized control. Introducing a new flag (E Flag,

as it is suggested in [3]) in the RAs will signal the hosts that the network is operating in the Efficient Aware mode, meaning that the hosts would be Efficient Aware Hosts - EAHs and the router would be IPv6 ND-efficiency-aware Router - NEARs.

As it was suggested in [11], in order to implement the E Flag, changes in the RADVD source code, as well as in the Linux Kernel have to be made. Both Linux Kernel and RADVD are open source and can be downloaded from GitHub.

3.2.1   The E Flag - RADVD Implementation

First, as per [11], a new parameter which defines the E Flag is introduced, named `AdvEFlag`. This parameter is to be defined in the structures `ra_header_info` and `AdvPrefix` as an integer.

In the defaults.h header file, the E flag was added as a macro. The default value of 1 was given, which means all routers are working in the Efficient Aware mode (not mandatory). This can be changed depending on the setup - we can have a mixed mode network where part of the routers can work in the efficient aware mode, part can work in the legacy mode.

```
#define DFLT_AdvEFlag 1
```

The E Flag was declared in gram.y as a token in the bison declaration:

```
%token T_AdvEFlag
```

In the Flex file, scanner.l, the flag was added as a token named `T_AdvEFlag`, which will be then returned as input to the Bison file gram.y:

```
AdvEFlag { return T_AdvEFlag; }
```

A grammar rule was added in gram.y, where the `AdvEFlag` member of the `ra_header_info` structure is accessed, which is a member of the `iface` structure. The flag is defined as a SWITCH, so that it can be enabled or disabled, depending if the network is operating in the efficient aware mode or in the legacy mode:

```
1  { | T_AdvEFlag SWITCH ';' /* E */
2          {
3                  iface->ra_header_info.AdvEFlag = $2;
4          }
5    }
```

In the process.c file, an if condition was added for the E flag, which verifies if the flag received the correct value:

```
1  if  (( radvert −>nd_ra_flags_reserved & ND_RA_FLAG_E)
2      && ! iface −>ra_header_info . AdvEFlag) {
3    (LOG_WARNING,  "our␣AdvEFlag␣on␣%s
4  ␣␣␣␣doesn't␣agree␣with␣%s" ,  iface −>props . name,  addr_str );
5  }
```

The configuration file enables the possibility to configure the RADVD daemon by simply enabling or disabling any flags or parameters. The radvd.conf5 file is responsible for this. Therefore, the addition was made here:

```
1  .TP
2  .BR AdvEFlag  "␣"  on  |  off
3
4  Default :  off
```

The radvdump file was also updated by adding in the `print_ff()` function a condition which is verifying if the E flag is enabled in the configuration file and then printing it in a dump capture, together with the rest of the configured RADVD parameters:

```
1  if  (! edefs  ||  DFLT_AdvEFlag != (ND_RA_FLAG_E ==
2      ( radvert −>nd_ra_flags_reserved & ND_RA_FLAG_E)))
3    printf ( "\tAdvEFlag␣%s;\n" ,  ( radvert −>nd_ra_flags_reserved &
4      ND_RA_FLAG_E) ?  "on"  :  "off" );
```

In send.c, the following syntax was added in the texttttadd_ra_header and `add_prefix` structures. If the bitwise "OR" is true, `nd_ra_flags_reserved` will receive the value of ND_RA_FLAG_E, which should be the value of the E Flag received from the Kernel.
In the Kernel, in the include/linux/icmpv6.h header file, the E flag is defined as `ND_RA_FLAG_E`, with a hexadecimal value of 0x02: `#define ND_RA_FLAG_E 0x02`

```
1  radvert . nd_ra_flags_reserved |=
2          ( ra_header_info −>AdvEFlag) ? ND_RA_FLAG_E :  0;
3
4  pinfo . nd_opt_pi_flags_reserved |=
5          ( prefix −>AdvEFlag) ? ND_OPT_PI_FLAG_ONLINK :  0;
```

This concludes the changes for the E Flag implementation on the RADVD router side. Having those changes done to the RADVD code, the code can now be compiled, ran and installed on a Linux machine, having the E Flag set to 'on' in the configuration file. By capturing a packet through Wireshark, it can be seen that now the second less significant bit (Reserved) of the 8 bit long Flag string is set to 1:

```
▼Flags: 0x02
  0... .... = Managed address configuration: Not set
  .0.. .... = Other configuration: Not set
  ..0. .... = Home Agent: Not set
  ...0 0... = Prf (Default Router Preference): Medium (0)
  .... .0.. = Proxy: Not set
  .... ..1. = Reserved: 1
```

Figure 3.2: Wireshark Capture of Flags

The capture can't show the flag named as the E Flag, as this is part of Wireshark implementation. As it can be seen in Figure 2.3 and according to RFC 5175 [19], the flag options of the Router Advertisements have the last 2 bits unused.
A Radvdump capture can also show that the E flag is enabled:

```
# radvd configuration generated by radvdump 2.12
# based on Router Advertisement from fe80::a00:27ff:fe67:a7b1
# received by interface eth0
#

interface eth0
{
        AdvSendAdvert on;
        # Note: {Min,Max}RtrAdvInterval cannot be obtained with radvdump
        AdvManagedFlag off;
        AdvOtherConfigFlag off;
        AdvEFlag on;
        AdvReachableTime 0;
        AdvRetransTimer 0;
```

Figure 3.3: Radvdump Capture

3.2.2   Unicast RA/RS messages

As [3] suggests, by introducing the Address Registration Option as a novel method of handling Address Registration and Duplicate Address Detection, it enables the protocol to reduce multicast signaling to a minimum. It is assumed that this would greatly improve the battery life of 6LoWPAN devices.

The L2 SLLA, which was not mandatory in RA/RS messages, now should always be included. In RADVD, this can be enabled by turning the AdvSourceLLAddress switch from the configuration file to on:

```
.BR AdvSourceLLAddress " " on | off
```

Moreover, the UnicastOnly option from the configuration file should also be turned on, to avoid periodic multicast messages:

```
.BR UnicastOnly " " on | off
```

This concludes the implementation details of Efficient NDP at the router side, inside the RADVD, covered in this thesis. The mixed operating mode of NDP was not covered, which, when fully implemented, would allow NDP to function in both Legacy and Efficient modes. Another option that was left out, due to the broadness of the work, is the RAO option, described in Chapter 2.2.6. Since this option would be attached to RA messages, it would also be implemented in RADVD.

3.3   Linux Kernel Overview

The implementation of the suggested changes involves understanding general concepts of Networking inside the Kernel. The Linux Kernel handles the Layer 2, Layer 3 and Layer 4 of the OSI model, respectively the Data Link, Network and Transport layer.

net_device

The Network Device represents the foundation of the network stack in the Linux Kernel. It is defined in `include/linux/netdevice.h`, inside the net_device structure, and consists of parameters like IRQ, MTU, MAC Address, name of device etc. The net_device consists of different categories, such as Configuration. Here, the link layer address is configured with `unsigned char dev_addr[MAX_ADDR_LEN]`, `addr_len` being the length of the address in bytes. Another category would be the Link Layer Multicast, where the multicast addresses used for sending packets are configured. Therefore, the structure `struct dev_mc_list *mc_list` is used for devices which use multicasting. More details can be found in [13] and [14].

Socket Buffer

In the Linux Kernel, a packet is represented through a structure named `sk_buf` (SKB-Socket Buffer). This structure is located in the `include/linux/skbuff.h` header. Depending on the needs, the SKB can retrieve the L2, L3 or L4 header of a packet. A thorough explanation of the Socket Buffer can be found in [14], however here are some important structures of the `sk_buf`, required for understanding the source code of the ndisc protocol:

- `struct net_device *dev`
  A SKB contains a dev member, which points to the network device (net_device), which can be incoming or outgoing, depending if the packet is received or

about to be transmitted.

- `struct sock *sk`

  A pointer to a sock data structure; used when generating data locally.

- `alloc_skb, kfree_skb, skb_reserve, skb_put, skb_push, skb_pull`

  Functions for memory allocation, memory releasing, space reservation for header, addition of a block of data in the buffer at the beginning or end (put and push), removal of a block of data from the buffer.

Netlink

In this thesis, we will refer only to IPv6 packets, since NDP is an IPv6 Protocol. When a packet is received by the Kernel, the method `ipv6_rcv()` is called and the network device driver is sending it further to the Network Layer.

Similarly, packets are created locally through Transport Layer Protocol sockets, then forwarded to lower layers. The `struct sock` represents a L3 socket and the `struct socket` represents an user-space socket. The ND Protocol determines the MAC address of a host through its IP address. Using the destination and the MAC address, the Ethernet header is constructed and an user-space socket can send a packet.

The Linux Kernel is communicating with the user-space through sockets called **netlink sockets**, a new method which replaces the `ioctl()` system calls. Netlink sockets in user-space communicate with netlink sockets in the kernel space, and the communication is bidirectional. Through this method, system parameters of the kernel can be modified. Only the neighboring system part of the netlinks are investigated, as this is the general scope of this thesis.

The Neighbor Discovery Protocol is exchanging ICMPv6 messages through rt-netlink messages (NETLINK_ROUTE). Those messages are categorized into families, such as ADDR (network addresses), NEIGH (Neighboring subsystem messages), NEIGHTBL (neighboring table). Each Netlink family can send messages for creating a new entry (e.g.: RTM_NEWNEIGH), for deleting an entry (RTM_DELNEIGH) and for retrieving an entry (RTM_GETNEIGH).

Each time an ICMPv6 message is received, the `icmpv6_rcv()` function is called, and if the message is of ND type, the `ndisc_rcv()` is further on called. This function is handling all five NDP message types: RA, RS, NA, NS and Redirect.

The Neighbor

In the Neighbor Discovery Protocol, one of the most important data structure in the Kernel is the Neighbor structure - `struct neighbour`. This structure is defined in `include/net/neighbour.h`. Inside a network, all neighbors are stored in a Neighbor Cache Entry (NCE), also called neighboring table (ARP table for IPv4), defined in `include/net/neighbour.h` as `struct neigh_table`. For NDP, `nd_tbl` is an instance of `neigh_table`. The neighboring table is created through the `neigh_table_init()` function. A constructor method, named `ndisc_constructor()`, achieves initialization, when a new neighbor object has to be introduced. `__neigh_create()` is calling for this constructor and returns 0 when a new neighbor has been added successfully. Similarly, the functions `pndisc_constructor()` and `pndisc_destructor()` handle creating and removing a neighbor proxy entry.

Before calling the constructors, however, memory has to be allocated for a new neighbor. This is done through `neigh_alloc()`, together with a process called Garbage collector. This process attempts to free up some unused memory, and if it's successful, `neigh_hash_grow()` is called to increase the size of the hash table. Figure 3.4 shows the process of adding a new neighbor to the network.
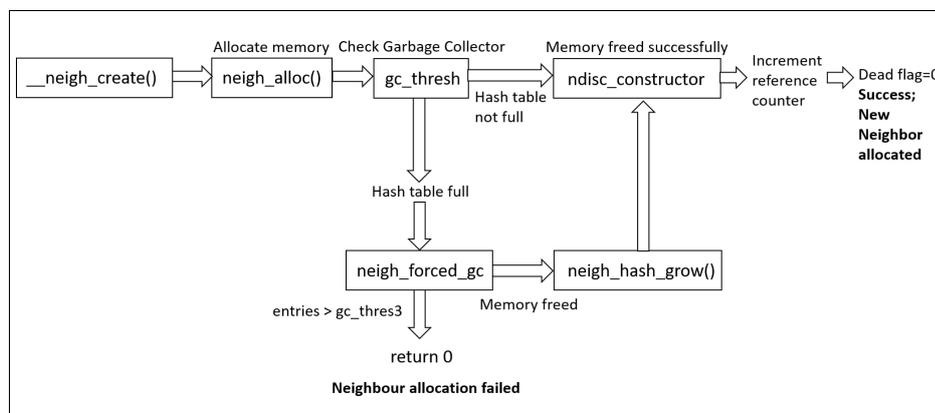
Figure 3.4: Adding a new neighbor

When a neighbor needs to be registered to the network, first a look-up in the existing NCE is done through the `neigh_lookup()` function. If the look-up fails, no new entry is added in the NCE. If the look-up succeeds, a new entry is added in the NCE through the `neigh_update()` function.

Different parameters of NDP can be modified in the `neigh_parms` object of the `neigh_table`. Those parameters are exported to the `/proc/sys/net/ipv6/neigh` directory.

3.4   Linux Kernel Implementation

The implementation of suggested protocol changes in [3] have to be done on both the client side, respectively Linux Kernel, and host side, in the RADVD code. Inside the Kernel, following source code files are mainly involved in the functionality of the Neighbor Discovery Protocol:

- `include/net/neighbour.h`

- `net/ipv6/ndisc.c`

- `include/net/ndisc.h`

- `include/net/if_inet6.h`

- `include/uapi/linux/icmpv6.h`

3.4.1   The E Flag - Kernel Implementation

On the host side, i.e. in the Linux Kernel, as it was mentioned above, the E Flag was defined in the include/linux/icmpv6.h header file, as `ND_RA_FLAG_E`, with a hexadecimal value of 0x02. Inside include/uapi/linux/icmpv6.h, following structure `icmpv6_nd_ra` can be found, containing flags of RAs, and where the E Flag has to be added, as it can be seen in Listing 3.1.

Listing 3.1: E Flag definitions

```
1    struct icmpv6_nd_ra {
2    home_agent:1,
3    other:1,
4    managed:1;
5    eflag:1; ...} u_nd_ra;
6    ...
7  #define icmp6_addrconf_managed   icmp6_dataun.u_nd_ra.managed
8  #define icmp6_addrconf_other   icmp6_dataun.u_nd_ra.other
9  #define icmp6_addrconf_eflag   icmp6_dataun.u_nd_ra.eflag
```

Next, the E Flag needs to be defined inside `inet6_dev.if_flags` in the include/net/if_inet6.h header file: `#define IF_RA_EFlag 0x08`.

In net/ipv6/ndisc.c, the main source file of the ND protocol, the value of the flags is stored in `ndisc_router_discovery()` function, where an addition for IF_RA_EFlag has been made in Listing 3.2.

Listing 3.2: E Flag definitions

```
1    in6_dev−>if_flags = (in6_dev−>if_flags & ~(IF_RA_MANAGED |
2         IF_RA_OTHERCONF | IF_RA_EFlag)) |
3         (ra_msg−>icmph.icmp6_addrconf_managed ?
4           IF_RA_MANAGED : 0) |
5         (ra_msg−>icmph.icmp6_addrconf_other ?
6           IF_RA_OTHERCONF : 0)
7         (ra_msg−>icmph.icmp6_addrconf_eflag ?
8           IF_RA_EFlag :0);
```

The values of those flags are being sent from the Kernel to the user-space by Netlink Sockets (A short review of Netlink sockets is given in the Kernel Overview Chapter 3.3). The value of the E Flag should be decoded from the received RA's, and if the flag would be enabled, the protocol should work in the efficient mode. A code found at [20], should retrieve the value of the M/O flags. This could be modified accordingly, in order to read the E Flag as well and introduce a knob which would set efficient mode on or off. This hasn't been studied in this thesis, but it can be considered a future work topic.

3.4.2   Address Registration Option

Summary of the implementation

The idea of Address Registration Option (ARO) was introduced in RFC 6775 [4] and then further suggested for implementation in [3]. The main idea of ARO is to replace the usage of multicast messages in the DAD process with a new method of handling registrations, by including an ARO header in the NS/NA messages. Inside this header, the hosts have to include their L2 Address (SLLA), which will be stored at the router side in the router's NCE. This allows the router to reply to NS's with unicast NA's and handle address registration of hosts, which would replace the multicast DAD process used by the Legacy NDP. If SLLA is missing, it would either mean the protocol is working in the Legacy mode or there is an error, as it can also be seen in Table 3.1.

ARO	SLLA	Status
✓	X	error
✓	✓	Working as Efficient NDP; Update NCEs
X	✓	Working as Legacy NDP
X	X	Working as Legacy NDP

Table 3.1: Status of the NDP Protocol

Before digging into a detailed section of code implementation for the ARO option, a summary of the implementation is depicted in Figure 3.5, to show which steps were followed and what are the key concepts that were changed or introduced. On the right side of the Figure, a list of each approached function or feature is shown, together with a reference that points to the Listing in which it was implemented, found in the detailed implementation description.
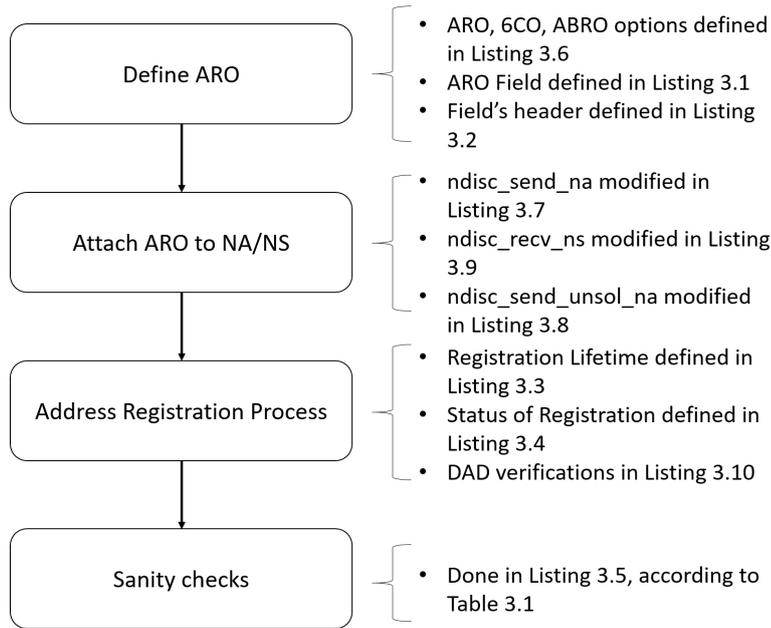
Figure 3.5: Code implementation steps

Detailed implementation description

An implementation attempt was made by Varka Bhadram in [15], however, as the targeted changes enter the Linux Kernel more deeply, the implementation wasn't finished, nor tested. This implementation was examined in detail, and code comments done by Alexander Aring were implemented. This implementation will be now shown in detail, by taking snippets of the proposed code and explain which additions were necessary and why.

In the Linux Kernel, the `ndisc.h` header contains all parameter definitions of the NDP protocol. Here, the ARO option was defined, with the value of 33, as it is the type defined by [4]. This is to be seen in Listing 3.3.

Listing 3.3: ARO Definitions [15]

```
1   enum {
2           ...
3           ND_OPT_ARO = 33,
4           __ND_OPT_MAX
5   };
6   struct ndisc_options {
7           struct nd_opt_hdr *nd_opts_aro;
8   ...
9   }
```

The header of the ARO is introduced in a new structure, seen in Listing 3.4, with its corresponding fields defined in [4] and also shown in Figure 2.5 of Chapter 2.2.4.

Listing 3.4: ARO Header [15]

```
1   struct aro_option {
2           struct nd_opt_hdr aro_opt;
3           __u8 status;
4           __u16 reserved;
5           __u16 reg_lifetime;
6           __u8 eui_64[8];
7   };
```

ARO uses a timer called registration lifetime, which serves for managing and refreshing existing neighbor registration with the routers. This registration lifetime, defined in Listing 3.5, shall be the same as the Router Lifetime of the RA messages. `jiffies` is a variable used by the Kernel, which is incremented when a used timer is expiring [14]. The timer is implemented as the Router Lifetime of RAs (gc_staletime in Listing 3.5).

Listing 3.5: Registration Lifetime [15]

```
1   %include/net/neighbour.h
2   struct neigh_params      {
3     #ifdef CONFIG_IEEE802154_6LOWPAN
4         __u16 reg_lifetime;
5     #endif
6   }
7   %net/core/neighbour.c
8   if ((atomic_read(&n->refcnt) == 1) && (state == NUD_FAILED ||
9     time_after(jiffies, n->used + n->parms->gc_staletime) ||
10      #if IS_ENABLED CONFIG_IEEE802154_6LOWPAN
11        time_after(jiffies, n->used + n->parms->reg_lifetime)
12      #endif
13      )) {
14        *np = n->next;
15        n->dead = 1;
16   write_unlock(&n->lock);
17   ...
```

A new function was created (Listing 3.6), which handles the proposed fields of the ARO. The ARO included in NS or NA messages is used to perform Address Registration, Neighbor Unreachability Detection (NUD) or to refresh an existing registration. The router receiving a NS first does a lookup in it's NCE, to check if the received SLLA is already registered or not, then it is checked against duplicates. If it's not duplicate and the SLLA is already registered, then the registration lifetime of the ARO is verified as following:

- If the registration lifetime of the ARO is 0, it means a host wants to de-register itself from the router. Then, the neighbor is declared as dead, NUD is set to NUD_FAILED and the status field of the ARO is set to 0, which means it was successfully removed from the network. (Lines 10-13)

- If the registration lifetime of the ARO is not 0, the registration is refreshed and then the status field of the ARO is set to 0, which means the registration was successful (Lines 14-16).

If the address is a duplicate, then the status of the ARO field is set to 1. Otherwise, if the SLLA of the ARO is not found in the router's NCE, Address Registration is performed, a new entry of a neighbor is added to the NCE and the status of ARO is set to 0. If the NCE is full, the registration failed and the status field is set to 2. (Lines 21-29) The code of this function can be seen in the Listing 3.6.

Listing 3.6: Function handling ARO options [15]

```
1   static struct neighbour *ndisc_lowpan_options_handle(
2                               struct net_device *dev,
3                               const struct in6_addr *saddr,
4                       struct aro_option *aro)
5   {
6       struct neighbour *neigh = NULL;
7       neigh = neigh_lookup(&nd_tbl, saddr, dev);
8       if (neigh) {
9           if (!memcmp(neigh->ha, aro->eui_64, 8)) {
10              if (aro->reg_lifetime == 0) {
11                  neigh->dead = 1;
12                  neigh->nud_state = NUD_FAILED;
13                  aro->status = 0;
14              } else {
15                  neigh->parms->reg_lifetime = aro->reg_lifetime;
16                  aro->status = 0;
17              }
18          } else
19              aro->status = 1;
20      } else {
21          neigh = neigh_create(&nd_tbl, saddr, dev);
22          if (neigh) {
23              neigh_update(neigh, aro->eui_64, NUD_STALE, 0);
24              neigh->parms->reg_lifetime = aro->reg_lifetime;
25              aro->status = 0;
```

```
26            } else
27                aro->status = 2;
28        }
29        return neigh;
30    }
```

As [3] suggests, the SLLA has to be included in the ARO option, in order to enable the usage of unicast messaging. However, the protocol should be able to work in mixed mode, so when Legacy NDP is in use, SLLA can be sometimes not used. When a NS is processed, some sanity checks should be performed, which verify if the received NS does include an ARO option (Efficient NDP), if the ARO option does include a SLLA, or if no ARO option is present. Those sanity checks were done in the `ndisc_lowpan_ns` function, whose code can be analyzed in Listing 3.7. The 'if' statements were slightly modified from Varka's code, by analyzing the code comments in [15] done by Alexander Aring.

Listing 3.7: Sanity check of ARO [15]

```
1    static struct neighbour *ndisc_lowpan_ns(struct sk_buff *skb,
2                          struct net_device *dev,
3                          struct aro_option *aro_opt,
4                          u8 *lladdr)
5    {
6      const struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
7      struct neighbour *neigh = NULL;
8
9      if (!lladdr && aro_opt) {
10        ND_PRINTK(2, warn, "NS packet is not having the SLLAO\n");
11        return NULL;
12      }
13
14      if (lladdr && aro_opt) {
15        neigh = ndisc_lowpan_options_handle(dev, saddr, aro_opt);
16        if (!neigh) {
17          ND_PRINTK(2, warn,"NS : Error in lowpan option
18    handling\n");
19          return NULL;
20        }
21      }
22
23      if (lladdr && !aro_opt) {
24        neigh = __neigh_lookup(&nd_tbl, saddr, dev, 1);
25        if (neigh)
26          neigh_update(neigh, lladdr, NUD_STALE, 0);
27      }
28      return neigh;
29    }
```

The sanity checks from Listing 3.7 perform as following:

- If a NS is received, which contains the ARO option, but the SLLA is missing, a warning will occur. The ARO option in this case will be ignored, as per [4].

- If a NS is received, which contains both ARO and SLLA, the `ndisc_lowpan_options_handle()` from Listing 3.1 is called. This will analyze the received ARO, respectively it's registration lifetime, to check if refresh of registration is in order. NUD is also performed and DAD through the status field of the ARO. Should no neighbor be found, an error message will be generated.

- If the SLLA is present, but not the ARO, it can mean that the protocol is working in the legacy mode, so the NCE tables will be updated with the new SLLA, if not already present.

The 6LoWPAN options introduced in [4] - ARO, 6CO, ABRO, were added in `ndisc_parse_options()` function, as it can be seen in Listing 3.8. This function is called and verified by each of the corresponding function of sending / receiving NA, NS, RA, RS, or Redirect. 6CO and ABRO are not used in this thesis, but they could be investigated in the future, as they are part of the RFC 6775 implementation.

Listing 3.8: ARO options [15]

```
1   struct ndisc_options *ndisc_parse_options(u8 *opt, int
2   opt_len , ..)
3   {
4   ...
5                    case ND_OPT_ARO:
6                        if (!ndopts->nd_opts_aro)
7                            ndopts->nd_opts_aro = nd_opt;
8                        break;
9                    case ND_OPT_6CO:
10                       if (!ndopts->nd_opts_6co)
11                           ndopts->nd_opts_6co = nd_opt;
12                       break;
13                   case ND_OPT_ABRO:
14                       if (!ndopts->nd_opts_abro)
15                           ndopts->nd_opts_abro = nd_opt;
16                       break;
17  ...
18  }
```

The function `ndisc_send_na()`, which is the one used for sending Neighbor Advertisements, was modified by Varka in order to include the ARO. Therefore, the lines 5-8 of Listing 3.9 add the ARO space to the length of the options field of the NA, if it's being used. `skb_put()` of line 16 adds ND_OPT_ARO_SPACE bytes to the Socket Buffer, so that it is ready for transmission.

Listing 3.9: Sending NA's function [15]

```
1   static void ndisc_send_na(struct net_device *dev, ... ,
2                             struct aro_option *aro)
3   {
4   ...
5           if (inc_opt) {
6                   optlen += ndisc_opt_addr_space(dev);
7                   if (aro)
8                           optlen += ND_OPT_ARO_SPACE;
9           }
10  ...
11          if (inc_opt) {
12                  if (dev->type == ARPHRD_IEEE802154) {
13                      ndisc_fill_addr_option(skb,
14                        ND_OPT_TARGET_LL_ADDR, dev->dev_addr);
15                      memcpy((struct aro_option *)
16                              skb_put(skb, ND_OPT_ARO_SPACE),
17                              aro, sizeof(*aro));
18                  } else
19          ndisc_fill_addr_option(skb, ND_OPT_TARGET_LL_ADDR,
20                  dev->dev_addr);
21          }
22
23          ndisc_send_skb(skb, daddr, src_addr);
24  }
```

The most important outcome of the Efficient NDP is that multicast NS's are replaced by unicast NS's. This also means that the protocol has to stop sending unsolicited NA's. In the `ndisc_send_unsol_na()` function from Listing 3.10, the aro_option parameter was declared NULL (line 8), which means the ARO option is not to be included in unsolicited NA's, as they will not be used by Efficient NDP in the first place.

Listing 3.10: Unsolicited NA's [15]

```
1   static void ndisc_send_unsol_na(...)
2   {
3   ...
4           list_for_each_entry(ifa, &idev->addr_list, if_list) {
5                   ndisc_send_na(dev, NULL,
6                       &in6addr_linklocal_allnodes, &ifa->addr,
7               ...
8                       inc_opt=*/ true, NULL);
9           }
10  ...
11  }
```

Listings 3.11 and 4.1 describe the additions that were done by Varka to the `ndisc_recv_ns()` function, additions explained in theory in Chapter 2.2.4. When

a NS is received, a checkup has to be done to see if it contains an ARO. If so, the status and length of the ARO have to be verified. The status of the ARO has to be 0, otherwise it means the address is duplicate or the NCE is full. The length of the ARO has to be 2, as it is defined in [4]. A call to `ndisc_lowpan_ns()` from Listing 3.2 was done, to verify if the NS is containing the ARO and/or the SLLA. When this is verified, the node receiving the NS will update it's NCE with the received SLLA, perform DAD if necessary and reply to the NS with a corresponding NA.

Listing 3.11: Receiving NS's function [15]

```
1   static void ndisc_recv_ns(struct sk_buff *skb)
2   {
3   ...
4     if (ndopts.nd_opts_aro) {
5         memcpy(aro, (struct aro_option *)ndopts.nd_opts_aro,
6             sizeof(*aro));
7         if (aro->status != 0 || aro->aro_opt.nd_opt_len != 2) {
8             ND_PRINTK(2, warn, "ARO: invalid status and
9                             length, must discard the packet\n");
10            return;
11        }
12    }
13
14    if (dev->type == ARPHRD_IEEE802154) {
15        neigh = ndisc_lowpan_ns(skb, dev, aro, lladdr);
16        if (neigh) {
17            ndisc_send_na(dev, neigh, daddr, &msg->target,
18                idev->cnf.forwarding,
19                true, false, true, aro);
20        return;
21        } else {
22            ND_PRINTK(2, warn,
23                "NS : error in handling Lowpan packet\n");
24            return;
25        }
26    }
27  ...
28
29  }
```

Lines 3-7 of Listing 4.1 treat the following case: a NS was received, stating a duplicate address. This means that the protocol is working in the Legacy mode and has a conflict of address with another host, so an NA reply has to be sent back to resolve the conflict. The ARO field receives a NULL value here (line 6), because the protocol is in Legacy mode. The second part of the code handles the normal situation, in which when a NS is received, a NA reply has to be sent back.

Listing 3.12: Receiving NS's function [15]

```
1   static void ndisc_recv_ns(struct sk_buff *skb)
2   {
3   if (dad) {
4       ndisc_send_na(dev, NULL, &in6addr_linklocal_allnodes,
5         &msg->target, !!is_router, false, (ifp != NULL), true,
6                   NULL);
7       goto out;
8   }
9
10  if (neigh || !dev->header_ops) {
11      ndisc_send_na(dev, neigh, saddr, &msg->target, !!is_router,
12                  true, (ifp != NULL && inc), inc, NULL);
13      if (neigh)
14                  neigh_release(neigh);
15  }
```

This concludes the review of the code implementations proposed in [15], leaving the testing and evaluation of possible additional functions to the future work. The following chapter investigates a simulation environment, in which the Legacy NDP as well as the Efficient NDP are configured, in order to observe how the proposed changes of the protocol affect a network, in terms of performance and scalability.

Chapter 4

Simulation model

This chapter presents a simulation model that was implemented with two different test case scenarios, in which a Legacy NDP and a Efficient NDP network were configured. The simulations analyze a sleep and wake up scenario, in which the number of messages arriving at the hosts in both cases will be then compared in Chapter 5. Through this comparison, assumptions will be then made if hosts will wake up less frequent from sleeping cycles, hence consume less battery, as it was introduced in [3].

The software chosen for this simulation is OMNEeT++ v5.0. In order to describe the simulation environment, first an introduction to OMNeT++ is given in Chapter 4.1, and some of its main features that were essential for the simulation are presented. An example of those features is INET, which allows the user to create more complex simulations models.

In the next section, in Chapter 4.2, follows a description of the simulation model that was used for the sleep and wake up experiment, its components and goals. The section describes the implementation of the two scenarios, their parameters and in the end what were the limitations.

4.1 Simulator description

In this section refers to OMNeT++, its compound structure and how a network design can be represented. It also refers to INET, an extra tool of OMNeT++, used to create and simulate more complex simulation models.

4.1.1 OMNeT++

OMNeT++ is an open source discrete event simulator environment written in C++. It operates under the Academic Public License, which makes the software free for non-profit use. OMNeT++ has been recognized as reliable platform for creating and simulating networks and distribution systems between academics, so more and more researchers use it [24]. OMNeT++ has been designed to create and run large scale simulations, with hierarchical and customizable modules. In this thesis work, OMNeT++ v5.0 was used.

OMNeT++ uses a model language named NEtwork Description (NED), which allows the user to model many simple modules together. The simple modules can be combined and thus create more complex models. Figure 4.1 illustrates the structure of a compound module.
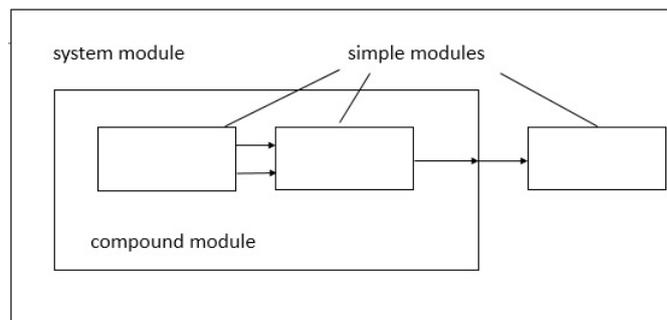
Figure 4.1: Module Structure

Once different modules are built, the network model can be defined with the use of Network Description (NED) language. NED language is used to define the topology of the network, to describe which modules are used, to define the interconnections between the modules, and the parameters of the compounds. Parameters can be set by using the user interface of OMNeT++, which translates automatically the different modules into code. Figure 4.2 and Listing 4.1 show an example of how two host are connected together, using the graphical interface and the same structure is shown in the source code of the NED language.
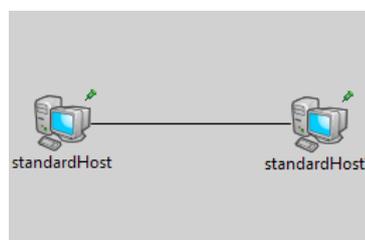
Figure 4.2: NED-based Network description

Depending on the experiment, parameters can also be defined on an INI file, which overrides the parameters set on the NED language. In this thesis work, parameters such as simulation time or number of hosts are defined on an INI file, while others such as router lifetime can be set with the NED language.

Listing 4.1: Receiving NS's function [15]

```
 1   network Network
 2   {
 3   submodules:
 4          standardHost: StandardHost6 {
 5          @display("p=38,82");
 6          }
 7       standardHost: StandardHost6 {
 8          @display("p=228,83");
 9          }
10   connections:
11          standardHost.ethg++ <——> standardHost1.ethg++;
12   }
```

4.1.2   INET

The INET framework was built as an extension library of the OMNeT++. The INET framework for OMNeT++ is the keystone of the simulator. OMNeT++ implements only generic and simpler modules, but the INET framework adds up a very large library of standards used in Internet networking such as TCP, UDP, IPv4, IPv6, ICMPv6, BGP, etc. Since NDP uses ICMPv6 control message, the use of INET was a necessity. On top of protocols issued from the OSI layers, INET also implements several applications models and routing protocols and is responsible for the configuration of the IP network (IPv4/IPv6). It also has wireless radio communication and distribution models along with implementations with MAC protocols.

Therefore, this framework is really important in order to have a more realistic simulation, but adds to complexity. Figure 4.3 shows for example the difference between a regular OMNeT++ host and one using the INET Framework.

In order for the simulation to run properly, each node must have an IP address. Configuration can be done either automatically, or manually. In this thesis work, it was done automatically, as it is mentioned on Chapter 4.2. Moreover, in order to implement network traffic, or assign IPv6 protocol in the simulation model, INET was necessary and was used as well.
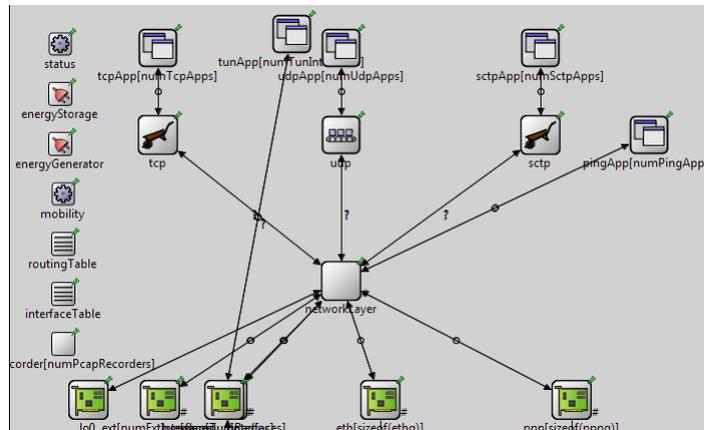
Figure 4.3: Example of host in INET

4.2   Sleep and wake up Scenario

As it was mentioned in the Introduction chapter (Chapter 1), the NDP protocol was proven to be inefficient in the wireless networks domain, creating unnecessary traffic and use of resources, such as increased processing power and creates consequences such as battery drainage. In the previous work, [11], a theoretical analysis was performed for Legacy and Efficient NDP, which has shown that the proposed protocol optimization is efficient, by offering a significant reduction of exchanged messages in the network. A sleep and wake up scenario was also briefly discussed, but not analyzed, which assumed to produce 100% saving of multicast messages.

In the current work, the sleep and wake up scenario was chosen for analysis, for both Efficient NDP and Legacy NDP. It was considered essential to show how both protocols operate under the same conditions, in a simulated environment. That is, in order to relate results in terms of messages savings to a practical approach, by assuming that battery lifetime of wireless devices running the Efficient NDP could be improved.

The purpose of this simulation is to determine how many messages arrive at the host side, in the two cases of Legacy NDP and Efficient NDP, and the outcome is to determine how many messages can be saved in the Efficient NDP. The message savings affect the number of times a host wakes up from sleeping cycles, which also impacts battery consumption. The results are discussed in Chapter 5.

4.2.1   Simulation parameters

In this simulation, two different scenarios were chosen. The first case analyzes Legacy NDP as per [8] and the second case analyzes the Efficient NDP as it is described in [3]. The previous analysis, [11], evaluated the behavior of the protocol in time intervals between 5 and 20 minutes. In the current work, in order to see how the exchange of multicast messages propagates in a longer period of time in a network, simulations were run for 1 hour, 4 hours, 8 hours and 24 hours. Moreover, to evaluate the scalability of the model, the number of hosts selected for each time interval were 1, 5, 20 and 100 hosts and one IPv6 router was used. Those parameters were selected for both Legacy and Efficient NDP. The Appendix contains information on how those parameters were adjusted.

Hosts in this experiment have been selected to join the network in a random manner, by using a uniform distribution. This was done in order to make sure that all hosts can enter the network during their respective simulation time. Table 4.1 shows the time intervals for each case.

Time / Hosts	1h	4h	8h	24h
1	(0, 3600)	(0, 14400)	(0, 28800)	(0, 86400)
5	(0, 720)	(0, 2880)	(0, 5760)	(0, 17280)
20	(0, 180)	(0, 720)	(0, 1440)	(0, 4320)
100	(0,36)	(0, 144)	(0, 288)	(0, 864)

Table 4.1: Uniformly distributed time intervals for different number of hosts and experiment time in seconds

Figure 4.4 gives an example of those chosen time intervals, for the case of having 5 hosts, joining the network, for a 1 hour simulation time. The simulation time is divided with the numbers of hosts, then the result is a equal time threshold, in which each host joins the network. For example, Host 1 joins the network somewhere between 0 and 720 seconds.
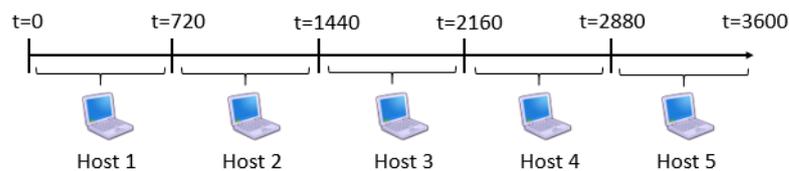

Figure 4.4: Time intervals in which hosts join the network

OMNeT++ parameters

In order to perform the simulation in OMNeT++, several parameters were chosen, some of them leading to model limitations:

- Wireless hosts were considered to be still during the whole duration of the experiment. While this option is available for implementation in OMNeT++, it was left out in order to reduce the complexity of the model. If implemented, it can be assumed that some delay and packet loss would be introduced in the network, which could increase the number of exchanged multicast messages, due to re-transmissions.

- UDP traffic was inserted into the network. Traffic is necessary when designing a simulation. TCP traffic could also have been used, but it wouldn't have made a difference. More information about how traffic was inserted into the network can be found in the Appendix.

- It was assumed that, during the exchange of messages, there was no packet loss, so there was no re-transmission. This can also be implemented in OMNeT++, by introducing a packet drop rate, according to a packet loss probability which can be calculated. Re-transmissions could also increase the number of exchanged multicast messages.

- The model does not cover a case of hosts leaving the network (or failing), while other hosts already joined the network. The hosts join the network progressively, according to Chapter 4.2.1. This impacts the number of exchanged NS/NA messages, sent when verifying the neighbor reachability, in the Legacy NDP case - the number would increase, making Legacy NDP less favorable compared to Efficient NDP. This does not affect Efficient NDP, as by implementing the proposed protocol changes, Efficient NDP hosts would perform NUD when they wake up ([3], Chapter 8.10), to refresh their registrations. As a result, they wouldn't have to send separate messages for NUD and for registration refreshing.

4.2.2   Legacy NDP

To set up the environment, an existing example model from the OMNeT++ library was used (`inet/examples/wireless/wiredandwirelesshostswithap/`) as a starting point, which had already Legacy NDP implemented and was designed to operate with wired and wireless hosts. This model was changed according to the selected simulation parameters, mentioned in Chapter 4.2.1, as well as according to the setup that follows in the current chapter. The modified code can be found in the Section 1 of the Appendix,. Hence simulations were run for a number of 1-100 hosts, for time intervals varying from 1 to 24 hours.

The network in the simulation consists of an IPv6 router module, an access point and a wireless host module that works with IPv6 protocol. The wired host was removed from the original model, since it was not needed. The number of hosts and the simulation can be changed accordingly. Those settings were be done in the configuration file of the project (`omnetpp.ini`). A radioMedium module was used for allowing wireless traffic in the simulation. Finally, the Configurator module allows the user to implement the IPv6 protocol. OMNeT++ contains the NDP protocol in the IPv6 router module and the time intervals between multicast messages are set there. This simulation model is depicted in Figure 4.5. As a note, in this figure, `wirelessHost6` stands for 1 to 100 hosts.
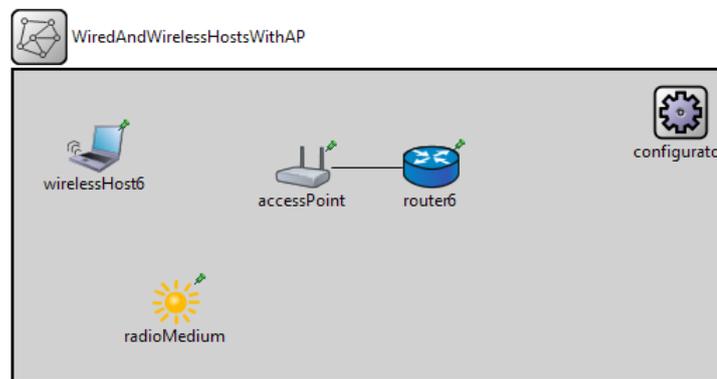
Figure 4.5: Simulation model design

Legacy NDP is sending out unsolicited multicast RAs from the router to the hosts, to update the network status. As it is of interest to count the number of messages arrived at each host, parameters defined in RFC 4861 have to be configured for this simulation model, in order to select how frequent those unsolicited RAs are sent. Those parameters are `MaxRtrAdvInterval` and `MinRtrAdvInterval`. They can be adjusted inside the NDP source code of OMNeT++, where they are defined as: `minIntervalBetweenRAs` and `maxIntervalBetweenRAs`.

`MaxRtrAdvInterval` can take values from the interval $4s < $ MaxRtrAdvInterval $< 1800s$, with a default value of 600s.

`MinRtrAdvInterval` can take values from the interval $3s < $ MinRtrAdvInterval $< 0.75 * $ `MaxRtrAdvInterval`, with a default value of $0.33 * $ `MaxRtrAdvInterval` if `MaxRtrAdvInterval` $\geq 9s$; otherwise, `MinRtrAdvInterval` $=$ `MaxRtrAdvInterval`.

In this simulation, the time intervals between RAs were selected as

$$\texttt{MinRtrAdvInterval} = 0.75 * 1800 = 1350s$$

and

$$\texttt{MaxRtrAdvInterval} = 1800s$$

Those values were selected, so that a minimum number of unsolicited multicast messages are being sent. This means that the Legacy NDP is performing in it's best case scenario, which can be then compared with Efficient NDP. At this point, it is expected that by using a worse case scenario (larger number of unsolicited multicast messages), when comparing Legacy NDP with Efficient NDP, the latter would turn out to show more message savings than the first. This is verified in Chapter 5.

Once all the mentioned aspects are implemented, the simulation can be started. It follows the standard working mode of NDP, as described in Chapter 2.1. The messages sent and received from the router go through the access point and reach the hosts. Simulation results are then extracted from the generated vector file, and imported in MS. Excel. The number of received messages at the host are ready to be counted. The results follow in Chapter 5.

4.2.3   Efficient NDP

The simulation model of Efficient NDP was created by starting with a basic Tic-Toc example from the OMNeT++ library, in which packets are just traveling from point A to B. On top of this, a simplified model of the NDP protocol was designed, with relevant requirements of Efficient NDP, in order to be able to have the exchange of RS/RAs and NS/NAs as it is described in Chapter 2.2. A detailed description about how this model was coded can be found in the Efficient NDP OMNeT++ Implementation details subsection of the Appendix. Having this model implemented, messages arriving at the hosts can be counted, compared with the Legacy NDP case from Chapter 4.2.2 and conclusions can be drawn on how much message savings are feasible in Efficient NDP.

Main characteristics of the model

The created model needs to have the following basic features, in order to be able to perform the exchange of messages as Efficient NDP requires:

- Hosts initially send RS messages when they join the network, to perform Network Discovery and receive RA's from the router;

- Hosts send unicast NS every time their Registration Lifetime is about to expire, and receive NA's as response, to refresh their registration, as it was explained in Chapter 2.2.4;

- Hosts join the network randomly, as in a real-life scenario, by using a Uniform Distribution. The reason for this was explained in Chapter 4.2.1.

A detailed explanation of the operating mode of the model will be described later
in this chapter, after the model limitations are presented.

Model Limitations

- Hosts and Routers do not have actual IP Addresses configured. This process
  is done when a host initializes his interfaces and joins a network, through
  a multicast NS message to the all-routers multicast address. Since this
  message is sent at initialization, and only to routers, it wouldn't reach other
  hosts in order to disrupt their sleeping cycles, so it can be left out.

- Only the Registered Lifetime field of ARO was implemented. In efficient
  ND, DAD is performed through ARO. It is achieved through the NS/NA
  messages mentioned above, which can signal a duplicate address trough the
  status field of ARO. This process was explained in detail in 2.2.4. The Reg-
  istered Lifetime is the most relevant parameter of ARO in this simulation,
  as it allows the protocol to exchange unicast NS/NAs which would perform
  the address registration.

- DAD, as mentioned above, was not implemented. In order to have it imple-
  mented, the already embedded Legacy NDP module of OMNeT++ would
  have to be modified, according to all aspects mentioned in the Efficient
  NDP implementation requirements, similar to the work that was done in
  Chapter 3. In the current simulation, this might not have a big impact on
  the results, as it is assumed that chances of having duplicate addresses in a
  real-life network, containing only 100 hosts, are quite low.

Detailed description of the model

In the efficient mode, multicast messages are removed and instead Registration
Lifetime from the Address Registration Option described in Chapter 2.2.4 is used
and set by default to 9000 seconds. A lower value could also have been used, but the
maximum value was selected, so that there would be a minimum number of NS/NA
messages exchanged, that would refresh hosts registrations. This translates to
hosts waking up less frequent from sleeping cycles. As per [4], hosts will rely
on the Address Registration Option annexed to NS/NAs in order to refresh their
registration, and this will be done through the Registration Lifetime field, which
should be the same as the Router Lifetime inside the RA/RSs.

The hosts can wake up at 2/3 of the Registration Lifetime, to refresh their regis-
tration, so in the simulation measurements, 6000 seconds was used as time interval
for NS/NAs between router and each host separately. At this point, it must be
noted that the update can also be done at 1/3 of the Router Lifetime, or another
alternative approach would be to wait until the end of the Registration Lifetime
before the update, as suggested in [3].

In order to perform this simulation in OMNeT++, a simplified model of NDP was designed through block definitions and classes. Each host, when joining the network, would immediately send a multicast RS and receive an RA to perform registration with the router. Then it would send an NS/NA to set up it's Registration Lifetime and configure it's IP address. Afterwards, it can enter sleep mode. The timer of 9000 seconds starts right away and every 6000 seconds, the host wakes up from sleep, sends an NS and receives an NA with the refreshed Router Lifetime. This process is depicted in Figure 4.6.

With this model implemented, simulations were run for 1, 5, 20 and 100 hosts, for a period of 1, 4, 8 and 24 hours. The number of messages arrived at the hosts were counted, in order to determine, for the sleep and wake up scenario, how many messages can be saved in the Efficient NDP, compared to Legacy NDP. Through this comparison, assumptions can be then made on if the amount of sleeping cycles being disrupted on battery operated devices running Efficient NDP is decreased. The results and this analysis follow in Chapter 5.
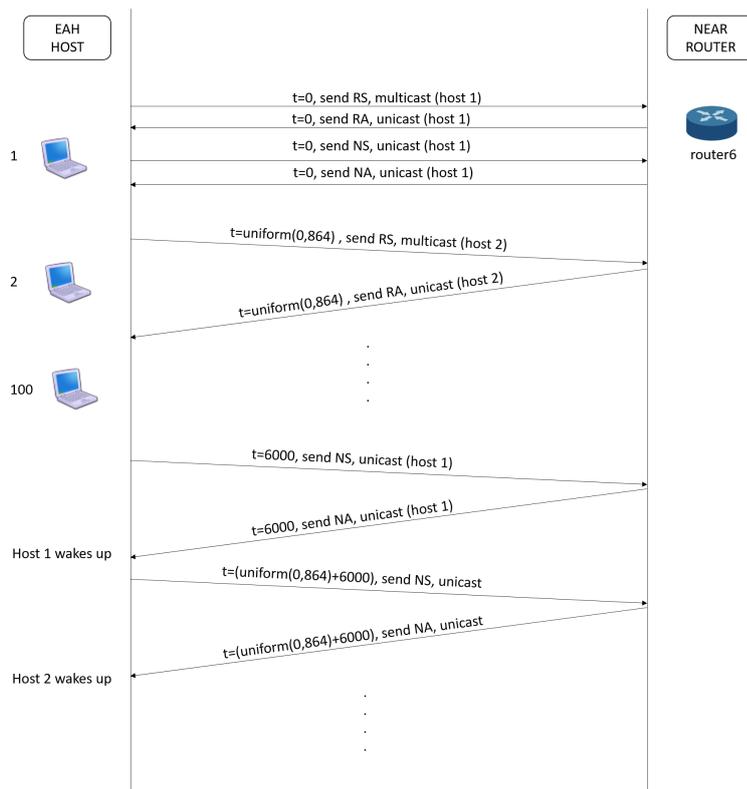
Figure 4.6: Network Setup of Efficient NDP

Chapter 5

Results

In this chapter, the results of the two test case scenarios investigated in Chapter 4 are described and analyzed. The objective is to show if the changes proposed in the Efficient NDP would reduce the number of exchanged messages in the sleep and wake up scenario between hosts and router and to what extend. Sleep and wake up scenario was selected, because battery operated devices enter into sleep mode when not in use to save power and the mechanism of Legacy NDP with the periodic messages disrupts their cycle. So, Efficient NDP should allow hosts to complete their sleeping cycles with no interruptions and thus save energy and power and reduce network traffic. Moreover, this protocol should be also applicable in networks with larger number of hosts.

In the first section, the reader can find the results for the Legacy NDP presented and explained, in the second section the results for the Efficient NDP and in the last section, a summary of the results is given and both test case scenarios are being compared.

5.1 Legacy NDP simulation results

In this section, the simulation results obtained from running the Legacy NDP test case scenario from Chapter 4 are analyzed. Tables 5.1 - 5.4 show the number of multicast exchanged messages in the Legacy NDP case, for 1, 5, 20 and 100 hosts, during a period of time of 1, 4, 8 and 24 hours. In the Legacy NDP, the router sends those periodic control messages to check and update their network status.

49

Time frame [h]	RA	NS	Total messages
1	6	2	8
4	12	2	14
8	22	2	24
24	59	2	61

Table 5.1: Number of messages received by 1 host (Legacy NDP)

Time frame [h]	RA	NS	Total messages
1	50	41	91
4	85	41	126
8	130	41	171
24	310	41	351

Table 5.2: Number of messages received by 5 hosts (Legacy NDP)

Time frame [h]	RA	NS	Total messages
1	500	660	1160
4	620	660	1280
8	800	660	1460
24	1560	660	2220

Table 5.3: Number of messages received by 20 hosts (Legacy NDP)

Time frame [h]	RA	NS	Total messages
1	10500	9700	20200
4	11100	9700	20800
8	12000	9700	21700
24	15600	9700	25300

Table 5.4: Total messages exchanged by 100 hosts (Legacy NDP)

The number of exchanged messages in the case of Legacy NDP can be explained by following the working mode of NDP, as per RFC 4861. As it can be observed in Table 5.1, for the case of 1 wireless host in the network, simulated in 1 hour, 6 RAs are being sent. This number can be justified through the following steps:

- When the simulation starts, first the router powers up and the host joins the network. The router is then initialized and multicasts up to 3 initial RAs to the host, according to the parameter described in RFC 4861, `MAX_INITIAL_RTR_ADVERTISEMENTS`.

- Next, the host is also initialized and sends a control message (RA) to the Router to get an IP address. The host answers the call and sends a RS as a reply.

- The simulation time is set to 3600 seconds (1 hour) and `MaxRtrAdvInterval` and `MinRtrAdvInterval` are set to 1350 and 1800 seconds respectively, as described in Chapter 4.2.2. Between this time interval, the router multicasts unsolicited RAs to check if the host is present and update its routing table, hence RA messages are not entirely periodic. Therefore, in 3600 seconds, 2 more RAs can be sent from the router which makes the total equal to 6.

After the first periodic RA/RS is sent from the router when powering up, the host initializes and multicasts a NS to the available router. The number of sent NSs is stable for every time interval and seems to increase with the number of hosts. The NS messages in the Legacy NDP can be either multicast or unicast [8]. They are unicast in the case where the node seeks to verify the reachability of a neighbor and multicast when the node needs to resolve an address. As it was mentioned in the OMNeT++ parameters section of Chapter 4.2.1, this model does not cover the case when hosts would leave the network or fail, while other hosts are joining. Therefore, NUD was implemented such that the unreachability detection is only done once, when a new hosts joins the network, which translates to having the same number of NSs for a given amount of hosts, for any period of time simulated. This was achieved by letting the `ReachableTime` parameter of NUD take it's default value, of 0 seconds. This is explained in details in RFC 4861, in Chapter 7.3.3. This can be investigated in more depth in a future work.

The number of exchanged NSs is added to the number of multicast RAs, to show the total number of messages that arrive at each host for each simulation. This means, except for the initial phase when the host powers up, it is assumed that each time a host receives a message, it wakes up from sleep to send a reply.

5.2   Efficient NDP simulation results

In this section, the simulation results obtained from running the Efficient NDP test case scenario from Chapter 4 are analyzed. Tables 5.5 - 5.8 display the obtained results for simulations ran in case of 1, 5, 20 and 100 hosts, for a period of time of 1, 4, 8, and 24 hours. In the Efficient NDP, besides the initial RA/RS, which are multicast, the host sends and receives NS/NA unicast control messages to/from the router and the router responds respectively. The setup was explained in Chapter 4.2.3.

To clarify the values obtained in Tables 5.5 - 5.8, an example was chosen: in the case where 1 host was used for simulation, in a 1 hour time period, it can be seen that 1 RA and 1 NS have been exchanged. The RA is sent in the initial start up

phase, when the router powers up, and is multicast. Afterwards, an NS is sent from the host, that is requesting to register in the network, receive an IP address set up its Registration Lifetime, so it can enter sleep. As Registration Lifetime was set to 9000 seconds and as it was assumed that hosts would refresh their registration every 6000 seconds, no further additional NS/NAs are exchanged in a 1 hour simulation time.

The reason that in higher time intervals there is still 1 message exchanged is that the time where the host enter the network each time varies according to the uniform distribution (Table 4.1), hence at 4 hours, the host will send an RS asking to join between 0 and 14400 seconds. Moreover, OMNeT++ seems to make hosts join a little before simulation time expires, so there is no further message exchange. This happens due to the algorithm it runs as it is mentioned in the second paragraph of this chapter. This is why, for the case of 1 host, there is no more than 1 multicast message.

For the cases with more than 1 hosts, it can be observed that the number of RAs is the same with the number of hosts. This is expected, because the RAs are sent from the router when a new host joins the network. Afterwards, no additional RAs are required, as the hosts update their neighboring tables and Registration Lifetime through NS/NA messages, then are ready to enter sleep mode.

After the 2/3 of the Registration Lifetime, hence 6000 seconds, NS are sent from the host to the router and the router replies with NAs. Those messages are unicast, but they are taken into consideration when counting because they are sent when the host wakes up from sleep mode, making it relevant to the sleep and wake up scenario chosen, so that it can be compared with the case of Legacy NDP.

After a host receives the NA, it goes back to sleep mode. The number of the NS messages depend only on when the hosts will enter the network for the first time. As it was mentioned in Chapter 4.2.1, they enter randomly, using a uniform distribution such that each host gets the chance to join the network, in the simulated time.

Time frame (h)	RA	NS	Total messages
1	1	1	2
4	1	3	4
8	1	5	6
24	1	15	16

Table 5.5: Number of messages received by 1 hosts (Efficient NDP)

Time frame (h)	RA	NS	Total messages
1	5	5	10
4	5	12	17
8	5	20	25
24	5	55	60

Table 5.6: Number of messages received by 5 hosts (Efficient NDP)

Time frame (h)	RA	NS	Total messages
1	20	20	40
4	20	45	65
8	20	82	102
24	20	227	247

Table 5.7: Number of messages received by 20 hosts (Efficient NDP)

Time frame (h)	RA	NS	Total messages
1	100	100	200
4	100	236	336
8	100	414	514
24	100	1137	1237

Table 5.8: Number of messages received by 100 hosts (Efficient NDP)

5.3   Summary of Results

It is worth mentioning, at this point, that the uniform distribution used for the hosts joining the network, as it was described in Chapter 4.2.1, generated the same results every time the same simulation was run. The reason for this is how OMNeT++ implements a deterministic algorithm, called the Mersenne Twister PRNG (Pseudorandom Number Generator), as it is mentioned in [25].

The reason for using this uniform distribution was also described in Chapter 4.2.1, and the objective was to observe the behavior of devices connecting to the network. After joining the network, hosts would exchange the initial RS/RA messages, followed by NS/NA messages to perform address registration, then they start their

Registration Lifetime timer. At this point, they are ready to enter sleep mode, meaning they won't be woken up by periodic multicast messages, as per Legacy NDP. However, hosts even when they are in sleep mode, are still connected to the network. The case of hosts disconnecting or failing in the network was not studied, due to lack of time. The impact of this case was explained in the OMNeT++ parameters section of Chapter 4.2.1.

Figures 5.1 - 5.4 display graphs which compare the obtained number of messages received by the hosts in Legacy NDP and Efficient NDP, after running the simulations for 1, 4, 8 and 24 hours, for 1, 5, 20 and 100 hosts respectively. The objective of running those two test case scenarios was to create a simulation environment in which a network would have many end nodes, to observe the exchanged control messages flow.
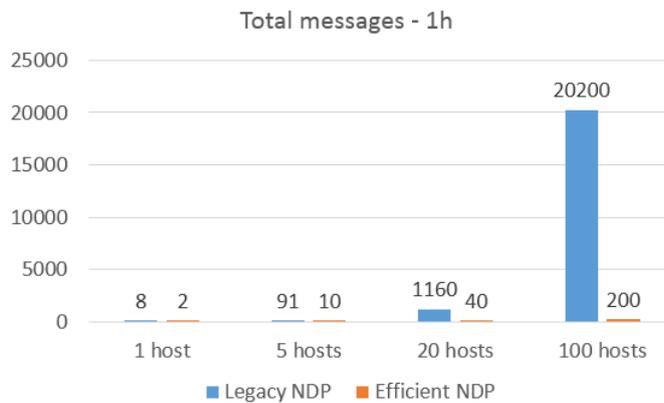

Figure 5.1: Comparison of total messages for 1 hour
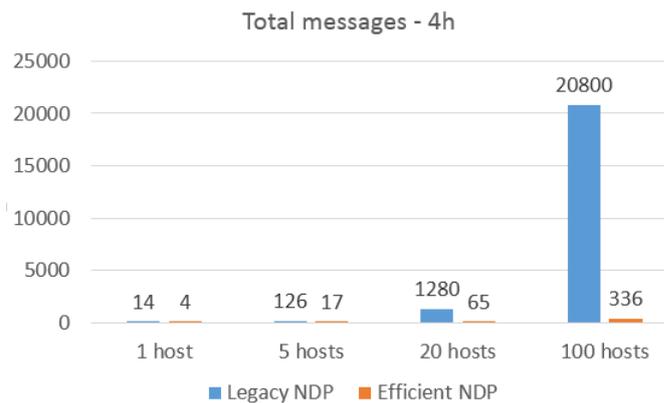

Figure 5.2: Comparison of total messages for 4 hours

It can be observed that in every case, the number of unsolicited messages in the Efficient NDP is significantly smaller than the ones from the respective Legacy

NDP case, where its function of sending multicast periodic RAs increases the total number of messages. This reduction in exchanged messages occurs, due to the mechanism of Efficient NDP which enables the hosts to exchange unicast messages with the router in order to keep their place in the network and discards the periodic multicast exchange of messages.
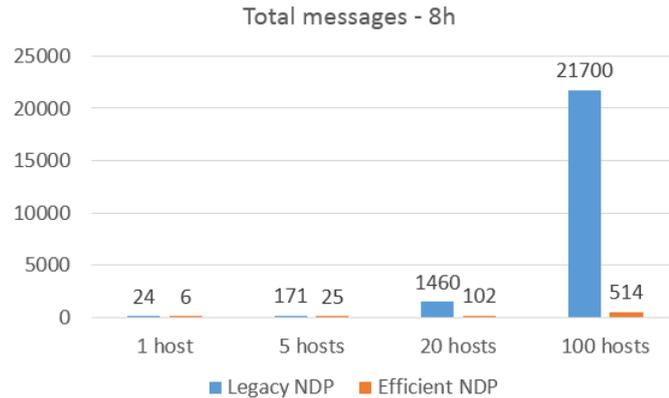
Figure 5.3: Comparison of total messages for 8 hours
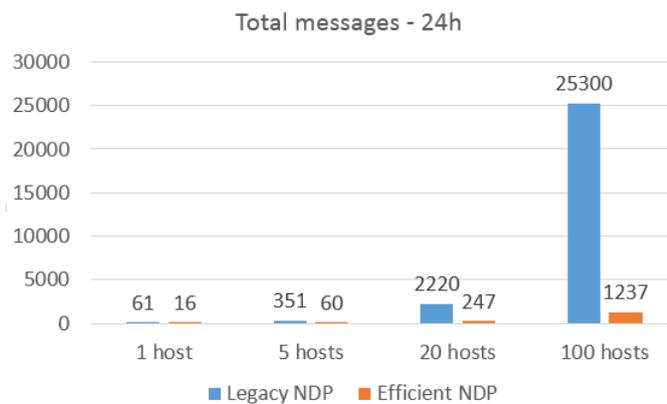
Figure 5.4: Comparison of total messages for 24 hours

As [3] suggests implementing a centralized system, the impact the changes have on the central node and on the whole system had to be analyzed and evaluated, in terms of scalability, as well as complexity, workload and system failure.

The central node of Efficient NDP has now a more centralized role, as it has to keep track of all the hosts Router Lifetime to perform a refresh of address registration if needed. It's workload would also be reduced, as it doesn't send periodic multicast RS/RAs anymore. Moreover, the router in Efficient NDP doesn't need to remind the hosts when to refresh registrations, as the hosts would wake up themselves

when their Registration Lifetime is about to expire.

As it can be seen from the Figures 5.1 - 5.4, more messages are being saved, as the number of hosts being used increase, making this model scalable. Percentages of total message savings are shown in Chapter 6, in Tables 6.1 - 6.4.

With this reduction of total messages, it can be assumed that the total network traffic produced is also reduced, hence less workload. Also, by having less messages being exchanged, less resources would be consumed by the operating devices, which together with less processing power being used, the complexity of the system as a whole entity would also be reduced. This could lead to an improve of battery lifetime. The difference in total exchanged messages in both cases, becomes more obvious as the number of hosts grow and naturally traffic in network grow as well.

The last objective of the thesis was to investigate the impact of the protocol changes on Stateless Address Auto Configuration (SLAAC). As per [3], the new Address Registration Option described in Chapter 2 and proposed for implementation in Chapter 3, is fully compatible with SLAAC and DHCPv6 and replaces the DAD procedure of RFC 4862. RFC 4862 was not studied in great detail in this thesis, as it is another standalone protocol, which while working closely together with RFC 4861, would require more time for analyzing it.

Chapter 6

Conclusions

The mechanisms proposed in the Efficient NDP, mainly aim at energy efficiency for mobile hosts that are battery powered. It aims at having them unaffected from any unnecessary control messages when they are in sleep mode, hence inactive and saving power. The results presented in Chapter 5, when the two cases were compared, showed that there is a significantly larger number of unnecessary control messages in the Legacy NDP, which is greatly reduced with the implementation of Efficient NDP. Those results also confirm the theoretical results obtained in [11]. By having the exchanged messages reduced, network traffic is also expected to decrease. It is also assumed that with the reduction of these messages, the energy of the battery could be saved for the mobile devices. However this is just an assumption as there is no clear technique on measuring the savings of energy power and it goes beyond the scope of this thesis. However, together with the reduction of these messages, power saving techniques would also have to be investigated, as manufacturers implement their own techniques [9].

Nevertheless, this outcome could have a very positive effect to both the manufacturers and the users of the mobile devices if Efficient NDP is actually deployed. Tables 6.1 - 6.4 show a summary of the estimated savings in the sent control messages for all the cases simulated and discussed in Chapter 5.

A hosts address remains unchanged even during periods, as it is already registered to the router. Its status is updated when it wakes up and sends the unicast NS to the router. Therefore, the implementation of Efficient NDP is able to save the number of total exchanged messages that are sent to the hosts. The hosts don't need to be awake to defend their address, which allows them to finish their sleeping cycle uninterrupted. This could improve their battery life.

Time (h)	Legacy NDP	Efficient NDP	Message Saving
1	8	2	75%
4	14	4	71.42%
8	24	6	75%
24	61	16	73.77%

Table 6.1: Total messages save by hosts - 1 host

Time (h)	Legacy NDP	Efficient NDP	Message Saving
1	91	10	89.01%
4	126	17	86.5%
8	171	25	85.31%
24	351	60	82.9%

Table 6.2: Total messages save by hosts - 5 host

Time (h)	Legacy NDP	Efficient NDP	Message Saving
1	1160	40	96.55%
4	1280	65	94.92%
8	1460	102	93.01%
24	2220	247	88.87%

Table 6.3: Total messages save by hosts - 20 hosts

Time (h)	Legacy NDP	Efficient NDP	Message Saving
1	20200	200	99%
4	20800	336	98.38%
8	21700	514	94.38%
24	25300	1237	95.11%

Table 6.4: Total messages save by hosts - 100 hosts

It can be assumed from the results that by the use of the Efficient NDP, there would be a reasonable gain in battery life, judging by the smaller number of the exchanged messages. Moreover, judging from the percentage values in the tables, it seems that there is a higher reduction of total messages exchanged, as the number of hosts increases, making this model scalable for a large number of hosts.

The Legacy NDP is generating periodic multicast messages, which increase the traffic in the network tremendously in proportion with the number of hosts. In Efficient NDP, multicast messages are being sent only at host initialization, while the rest of the exchanged messages are unicast.

Finally, it is crucial to compare the findings from the simulations run with the theoretical results in [11]. In the theoretical analysis of the previous work, in the sleep and wake up scenario, it was concluded that there is a 100% expectation of saving multicast messages and when not considering sleep and wake up, savings between 81% and 96%.

In this thesis, total message savings vary between 71% and 99%, for a sleep and wake up scenario, the latter being for the highest number of hosts assigned. This confirms that the theoretical results of the previous work are valid, especially when the number of hosts gets larger. The variation of the percentage can be explained through the usage of the uniform distribution when joining a network, which was chosen to simulate a real-life environment.

As of the coding of Efficient NDP, the thesis report concludes with a partial implementation on the Linux Kernel side and a working/tested implementation on the RADVD side. Due to the complexity of the Linux Kernel, the implementation couldn't be tested. In order to complete this implementation, deep knowledge and experience of Kernel programming is required, i.e system calls, device drivers, netlink sockets, together with a great understanding of operating systems. A list of future possible steps for implementation is shown in Chapter 7.

Chapter 7

Future Work

The coding part of Efficient NDP was only partially implemented in this thesis. Figure 7.1 displays the status of the implementation, on both RADVD and Linux Kernel side. The dashed lined boxes represent the future implementation steps.
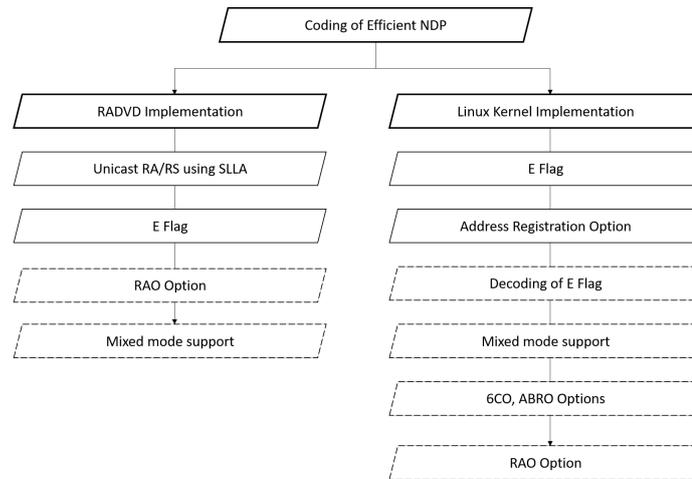

Figure 7.1: Implementation steps

The Registrar Address Option (RAO) has been left out, but can be implemented in a similar way with the Address Registration Option (ARO) inside the Linux Kernel. For implementing it in RADVD, one can use as a starting point the already implemented functions of 6CO and ABRO, which are the two other new options, besides ARO, which RFC 6775 introduced. 6CO and ABRO were out of the scope of this thesis, as this thesis mainly aimed to implement the features described in [3], but they should also be implemented, together with ARO in the Linux Kernel, to complete the implementation of RFC 6775. The RAO Option would cover the

case of device failure or the case of requiring different registrars than the default routers.

The E Flag was implemented in RADVD, but has to be decoded at the host side in the Linux Kernel, so that the protocol can switch to the efficient mode - this is mentioned in the end of Chapter 3.2.2. Moreover, the case of mixed mode was not studied, in which NDP would be able to function in both Legacy and Efficient modes.

To conclude on the coding part, once those mentioned steps are completed, the protocol can be tested in a virtual environment and further in a real environment, to draw the final conclusion regarding the Efficient NDP protocol.

When it comes to simulations, the analysis could be extended, if one could run the same simulations, including some assumptions or limitations that were taken in the present one to make the simulation model less complex. One logical step for continuation of the simulation work is to have the following factors included:

- Adding a probability of packet drop, which then as a result would require a re-transmission of the packet.

- Having some or all the host of the network be on the move. This could make simulation environment more realistic, considering that usually mobile hosts move and don't stand still.

- Hosts that leave the network and rejoin, so there are more updates in their routing tables and thus, more exchanged messages.

- Implementing DAD, so there is a check when a host entering the network and its IP is the same with another host of the same network. If this is the case, then the IP address request should be rejected and a new request should be made in order to acquire and IP address that is not occupied by any other host in the network.

- Simulating a network with more than 100 hosts to test its stability and scalability.

All these factors are expected to make the simulation models more complex when implemented, however they could make the outcoming results more realistic.

Bibliography

[1] R. James and Associates *"The Internet of Things - A Study in Hype, Reality, Disruption and Growth."*, St Petersburg, Florida, 800-248-8863 Jan 24, 2014

[2] H. Shah, R. Shrimali, V. Parikh *"Header Compression and Neighbor Discovery in 6LoWPAN based IoT - A Survey"*, IEEE WiSPNET 2016 Conference

[3] S. Chakrabarti. E. Nordmark, P. Thubert, M. Wasserman, *"IPv6 Neighbor Discovery Optimizations for Wired and Wireless Networks"*, February 27, 2015

[4] Z. Shelby, Ed., S. Chakrabarti, E. Nordmark, C. Bormann, *"Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)"*, November 2012

[5] F. Garneij, S. Chakrabarti, S.Krishnan„ *"Impact of IPv6 Neighbor Discovery on Cellular M2M Networks"*, July 2014

[6] E. Vyncke Ed., P. Thubert, E. Levy- Abengnoli, A. Yourtchenko, *"Why Network-Layer Multicast is Not Always Efficient At Data link Layer"*, February 2014

[7] E. Nordmark, A. Yourtchenko, S. Krishnan *"IPv6 Neighbor Discovery Optional Unicast RS/RA Refresh"*, March 2016

[8] E. Nordmark, W. Simson, H. Soliman, *"Neighbor Discovery for IP version 6 (IPv6)"*, September 2007

[9] C. Betancourt, K. Shin *"A white paper on Power-saving techniques lead to ultra-low-power processors for battery-operated devices"*, April 2013

[10] N. Kushalnagar, G. Montenegro, C. Schumacher *"IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals"*, August 2007

[11] Hariharasudan Vigneswaran, Jeena Rachel John, *"Analysis of IPv6 Neighbor Discovery for Mobile and Wireless Networks"*,[Faculty of Engineering, LTH, Lund University, Lund, Sweden], October 2015

[12] Z. Shelby, C. Bormann *"6LoWPAN the Wireless Embedded Internet"*, 1st ed, John Wiley and Sons Ltd, UK, 2009

[13] R. Rosen, *"Linux Kernel Networking: Implementation and Theory"*, [A press Berkeley CA, USA], 2013

[14] C. Benvenuti, O'Reilly, *"Understanding Linux Network Internals"*, 2006

[15] Bhadram V., Aring A., *"IEEE 802.15.4 stack for Linux."* October 3, 2013. Retrieved May 10, 2016, from https://sourceforge.net/p/linux-zigbee/mailman/message/31477539/

[16] S. Thomson, T. Narten, T. Jinmei, *"IPv6 Stateless Address Autoconfiguration"*, September 2007

[17] Retrieved July 11, 2016, from http://www.litech.org/radvd/

[18] Retrieved July 11, 2016, from http://dinosaur.compilertools.net/bison/bison_6.html#SEC49

[19] B. Haberman, Ed., R. Hinden, *"IPv6 Router Advertisement Flags Option"*, March 2008

[20] https://github.com/tomaszmrugalski/dibbler/blob/master/scripts/mo-flags.c and http://kumaran127.blogspot.ro/2013/05/get-m-and-o-flag-of-most-recently.html Retrieved July, 21, 2016

[21] https://OMNeT++pp.org/intro

[22] OMNeT++ https://inet.OMNeT++pp.org/Introduction.html

[23] OMNeT++ framework manual, https://OMNeT++pp.org/doc/OMNeT++pp/manual/

[24] INET framework manual, https://OMNeT++pp.org/doc/inet/api-current/inet-manual-draft.pdf

[25] https://omnetpp.org/doc/omnetpp/tictoc-tutorial/part2.html

[26] R. Hinden, S. Deering, *"IPv6 Addressing Architecture"*, February 2006

[27] http://docs.oracle.com/cd/E19620-01/805-4035/6j3r2rqmk/index.html, Retrieved June 24, 2016

Appendix

Legacy NDP OMNeT++ Implementation details

The model used for simulating the Legacy NDP sleep and wake up scenario is a modified version of **inet/examples/wireless/wiredandwirelesshostswithap/**, according to the setup mentioned in Chapter 4.2.2. The source code of the model is shown in Listing 7.1, followed by its configuration file, **omnetpp.ini**, shown in Listing 7.2.

Listing 7.1: OMNeT++ source code file of Legacy NDP

```
1   package inet.examples.wireless.wiredandwirelesshostswithap;
2   import inet.networklayer.configurator.ipv6
3           .FlatNetworkConfigurator6;
4   import inet.networklayer.icmpv6.IPv6NeighbourDiscovery;
5   import inet.node.ethernet.Eth100M;
6   import inet.node.ipv6.Router6;
7   import inet.node.xmipv6.WirelessHost6;
8   import inet.node.wireless.AccessPoint;
9   import inet.physicallayer.ieee80211.packetlevel
10          .Ieee80211ScalarRadioMedium;
11
12  network WiredAndWirelessHostsWithAP
13  {
14      parameters:
15          int n;
16          @display("bgb=503,434");
17      submodules:
18          wirelessHost[n]: WirelessHost6 {
19              @display("p=58,88");
20          }
21          router6: Router6 {
```

```
22              @display("p=412,88");
23          }
24          accessPoint: AccessPoint {
25              @display("p=323,87");
26          }
27          configurator: FlatNetworkConfigurator6 {
28              @display("p=323,165");
29          }
30          radioMedium: Ieee80211ScalarRadioMedium {
31              @display("p=98,392");
32          }
33      connections:
34
35          accessPoint.ethg++ <--> Eth100M <--> router6.ethg++;
36
37  }
```

Listing 7.2 shows the OMNeT++ configuration file of Legacy NDP, `omnetpp.ini`. Inside this file, the simulation time and the number of hosts can be adjusted. UDP traffic is also inserted into the network here, as it was mentioned in Chapter 4.2.1, in the OMNeT++ parameters section.

Listing 7.2: OMNeT++ configuration file of Legacy NDP

```
1  [General]
2  network = WiredAndWirelessHostsWithAP
3  sim-time-limit = 1h
4  tkenv-plugin-path = ../../../etc/plugins
5
6  # number of client computers
7  *.n = 5
8  **.*Host*.numUdpApps = 3
9  **.*Host*.udpApp[0].typename = "UDPEchoApp"
10 **.*Host*.udpApp[0].localPort = 1000
11 **.*Host*.udpApp[*].typename = "UDPBasicApp"
12 **.*Host*.udpApp[1..].destPort = 1000
13 **.*Host*.udpApp[1..].messageLength = 100B
14 **.*Host*.udpApp[1..].sendInterval = 1s
15 **.*Host*.udpApp[1..].stopTime = 300s
```

Efficient NDP OMNeT++ Implementation details

The simulation model for Efficient NDP was created by using a simple TicToc example from OMNeT++ as a starting point. Afterwards, the model was developed by adding two types of classes, the Hub (Router) and the Station (Host). To each class, functions were defined, as it can be seen in Figure 7.2.
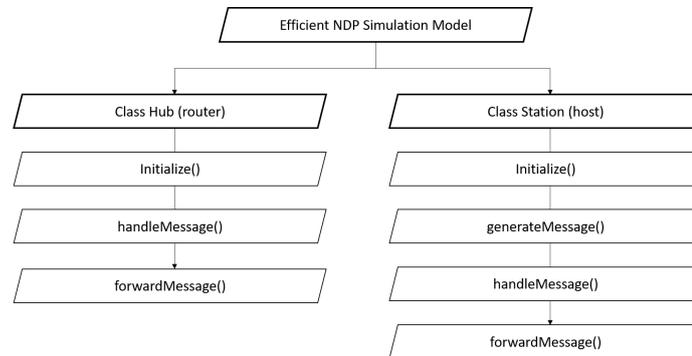
Figure 7.2: Efficient NDP Simulation model

In the Hub class, the function forwardMessage() is responsible for replying to the hosts with RAs, when RSs are received, as well as with NAs when NSs are received. In the Station class, in the initialize() function, the uniform distribution explained in 4.2.1 is configured, so that all hosts get the chance to join the network in the predetermined simulation time. The generateMessage() function takes care of sending out RS messages, where as the handleMessage() takes care of sending NS messages.

Listing 7.3: OMNeT++ source code file of Efficient NDP

```cpp
1  #include <string.h>
2  #include <omnetpp.h>
3
4  using namespace omnetpp;
5
6  class Hub : public cSimpleModule
7  {
8    protected:
9       virtual void initialize() override;
10      virtual void handleMessage(cMessage *msg) override;
11      virtual void forwardMessage(cMessage *msg);
12  };
13
14  Define_Module(Hub);
15
16  void Hub::initialize() {}
17  void Hub::handleMessage(cMessage *msg){
18      {
19              forwardMessage(msg);
20              }
21  }
22  void Hub::forwardMessage(cMessage *msg){
23      cModule *t = msg->getSenderModule();
24              int sn = t->getIndex();
25              EV << "Sender Module Index = " << sn << "\n";
26  const char *tt = msg->getName();
```

```
27  EV<<"MSG␣RECEIVED␣IS␣A␣"<<tt;
28
29  if (strcmp(tt, "RS")==0) {
30      cMessage *copy = msg->dup();
31      cMessage *msg2 = new cMessage("RA");
32      copy=msg2;
33      send(copy, "outhub", sn);
34      }
35  else
36  {
37      cMessage *copy = msg->dup();
38      cMessage *msg2 = new cMessage("NA");
39      copy=msg2;
40      send(copy, "outhubNA", sn);
41      }
42
43
44  }
45
46  class Station : public cSimpleModule
47  {
48    protected:
49      virtual cMessage *generateMessage();
50      virtual void initialize(int n) override;
51      virtual void handleMessage(cMessage *msg) override;
52      virtual void forwardMessage(cMessage *msg);
53  };
54
55  Define_Module(Station);
56
57  void Station::initialize(int n)
58  {
59
60      for (int i=0; i<100; i++)
61      {
62
63      if (getIndex()== i) {
64      cMessage *msg = generateMessage();
65      scheduleAt(i*uniform(0, 720), msg);
66      }
67      }
68
69  }
70
71  cMessage *Station::generateMessage()
72  {
73      cMessage *msg = new cMessage("RS");
74      return msg;
75      }
76  void Station::handleMessage(cMessage *msg)
77  {
```

```
78        const char *tt = msg->getName();
79        if (strcmp(tt, "RS")==0) {
80            send(msg, "outs");
81            cMessage *NS = new cMessage ("NS");
82            send(NS, "outNA");
83        }
84        else if (strcmp(tt, "NA")==0) {
85            cMessage *NS = new cMessage ("NS");
86            send(NS, "outNA");
87        }
88        else{
89            delete msg;
90            }
91  }
92
93  void Station::forwardMessage(cMessage *msg)
94  {
95  }
```

Listing 7.4 defines the gates used for the two classes, the Hub and the Station. Different gates were used at the hub, for incoming and outgoing traffic, one pair for sending and receiving RAs (outhub/inhub) and one pair for sending and receiving NAs (outhubNA, inhubNA). The same stands for the Station - the pair outs/ins was used for RA/RS traffic, and the pair outNA/inNA were used for NS/NA traffic. The third pair (outSt, inSt) was initially implemented for trafic between the hosts, but in the end it was not used, as the hosts didn't require to communicate between each other.

Lines 30-41 of Listing 7.4 connect the defined gates together, as in incoming/outgoing traffic, and a loop was written in which, by taking all the used hosts (1 to 100), a delay of 6000s was introduced when sending NA messages, which is the wake up timer of the hosts, when they have to refresh their Registration Lifetime, as it was explained in the Detailed description of the model section of Chapter 4.2.3.

Listing 7.4: NED file source code of Efficient NDP

```
 1  simple Hub
 2  {
 3      gates:
 4          output outhub[100];
 5          input inhub[100];
 6          output outhubNA[100];
 7          input inhubNA[100];
 8  }
 9
10  simple Station
11  {
12      gates:
13          output outs;
```

```
14          input ins;
15                  output outNA;
16                  input inNA;
17                  output outSt@loose;
18                  input inSt@loose;
19  }
20
21
22  network Tictoc1
23  {        parameters:
24      int n=5;
25      submodules:
26          hub: Hub;
27          Station[n]: Station {
28              @display("p=169,30");
29          }
30      connections allowunconnected:
31          for i=0..n-1 {
32              hub.outhub[i] --> Station[i].ins;
33              Station[i].outs --> hub.inhub[i];
34
35                      hub.outhubNA[i] --> Station[i].inNA;
36              Station[i].outNA --> {   delay = 6000s; }
37                  --> hub.inhubNA[i];
38          }
39          for j=0..n-2 {
40              Station[j+1].outSt --> Station[j].inSt;
41          }
42  }
```