Muticast DNS on Resource Constrained Devices in Low-power Lossy IPv6 Networks

Aleksandar Siljanovski

Computer Science
Jacobs University Bremen
Campus Ring 1
28759 Bremen
Germany

Type: *Guided Research*
Date: *May 8, 2011*
Supervisor: *Prof. Dr. Jürgen Schönwälder*

Abstract

With an increase of the possibilities and areas of applicability for simple low power wireless devices the number of these devices increases daily. Having more and more devices connected to the network introduces the need for larger address space; namely the IPv6 address space. This furthermore results in the need for efficient and simple device and service discovery solution. Taking into account the existing technologies and standards that are already and extensively in use, the suitable and compatible solution, for device and service discovery in IPv6 low power wireless networks, is Multicast DNS combined with DNS Service Discovery. Developing such a solution for these devices and combining that with existing software implementations on the client side we get a powerful tool for device and service discovery. Having such a tool, results in further development possibilities that are beyond the scope of this paper.

1 Introduction

Internet of things, also known as internet of objects, represents self-configuring wireless networks of sensors with a purpose of interconnecting everyday objects. The idea is to have embeddable and practical networking devices that will enable the objects, which are equipped with such a device, to connect to an information sharing wireless network. The applications of such an idea are immense, especially when considering the amount of objects that an average person comes in contact or is surrounded with in the daily life. Nowadays billions of devices worldwide communicate with hosts over the network and billions more are added each year. This explosion of networked machines benefits companies through efficiency and data sharing.[21] Another possible future realization using the concept internet of things is smart grid. Smart grid overlays the ordinary electrical grid but in addition it uses two-way digital communications to control and monitor appliances at the consumers' homes for power consumption. This potentially would result in reducing the costs, saving energy and benefit in many environmental aspects.[22]

Due to the ability to integrate with IP networks and taking into account that the public IPv4 address space is exhausted, IPv6 is the principal chosen datagram relaying protocol. Considering the extremely large address space of IPv6, the next generation of Internet applications would be able to communicate with devices attached to virtually all man-made objects.

To interconnect the low-power devices, which enable the Internet of things, using wireless communication the IEEE 802.15.4 radio standard was chosen. IEEE 802.15.4 standard was designed to address the need for a low-cost and low-power wireless solution, allowing the targeted device to operate with multi-month to multi-year battery life. The standard defines the protocol and interconnection for devices using low data rate, and low complexity short range radio frequency transmissions in a wireless personal area network (WPAN). The basic framework of low-rate WPAN includes $\approx$10 meter communications range with data rates of 250kb/s, 100kb/s, 40kb/s and 20kb/s (the different data rates result in different power consumptions). The range can be increased by mesh routing using full function devices (FFD) and mesh headers that contain the original source and the final destination.

Even though the IEEE 802.15.4 radio provides a link-layer wireless communication solution, IPv6 networking over this standard is not natively possible since IPv6 requires 1280 octets as the minimum MTU to prevent IP fragmentation and much lesser is available in IEEE 802.15.4 radio. As such, the 6LoWPAN standard, which stands for IPv6 over low power wireless personal area network, was designed to bring IPv6 connectivity over the IEEE 802.15.4 radio. It defines encapsulation and header compression mechanism that allows IPv6 packets to be sent and received over IEEE 802.15.4 based networks. The compression mechanism shrinks the IPv6 headers from 40 down to 2 bytes and the UDP header from 8 down to 4 bytes. IPv6

requires the MTU to be 1280 octets but on the other hand the IEEE 802.15.4 standard packet size is 127 octets with a maximum of 102 octets of payload which implies that an adaptation layer is needed that will carry on the fragmentation or reassembly of IPv6 packets. The 6LoWPAN adaptation layer is placed between the IEEE 802.15.4 link layer and the IPv6 network layer. The IPv6 nodes are assigned 128 bit IP addresses but IEEE 802.15.4 may use either 64-bit extended addresses or 16-bit addresses that are unique within a PAN. [19, 25, 6]

While the 6LoWPAN standard is implemented by multiple embedded operating systems such as Contiki, TinyOS and Archock, we chose to work with Contiki since it is quite actively developed and provides a 6LoWPAN implementation which closely follows the evolving standards. Contiki is an open source, highly portable, multitasking operating system for memory-efficient networked embedded systems and wireless sensor networks. It is designed for microcontrollers with little memory and computing capabilities. Contiki is written in the C programming language and consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime. Its processes use lightweight protothreads that provide a linear, thread-like programming style on top of the event-driven kernel. Contiki also supports per-process optional multithreading and inter-process communication using message passing.[15, 11, 14, 10]

Even though Contiki provides a stable 6LoWPAN implementation, there is currently no method for discovering the services and nodes that exist on the network. Since the Internet of things can involve a multitude of devices providing different services, it can quickly become impossible to keep a track of devices and services manually. As such, a service discovery mechanism that works with Contiki and 6LoWPAN is necessary. Service discovery protocols allow automatic detection of devices and services that are offered by these devices on a computer network. Service discovery is an important part of the Semantic Web and also the Internet of things, since the future Internet must allow software agents to make use of one another's services without the need for continuous user intervention. Services are characterized by both static and dynamic attributes. DNS based Service Discovery can be performed in different ways; allowing machines to automatically advertise their services or manually adding DNS records describing the services to be advertised for clients to discover. Service discovery could also be integrated into the IEEE 802.15.4 beacon frames, 6LoWPAN neighbor discovery, or the relatively new CoAP protocol, a specialized transfer protocol for constrained networks using HTTP for integration, can also be used for service discovery. [14, 8, 19, 11]

However, unlike Multicast DNS (mDNS) and DNS Service Discovery (DNS-SD), all the other methods either require adaptations to existing standards or design of new approaches. mDNS and DNS-SD are already designed to work with IP based networks and leverage existing technologies which allows for the emerging Internet of Things to be integrated into existing networks and services without the need for any middleware. As such, the aim of this work is to develop, test and evaluate an

implementation of mDNS and DNS-SD for resource constrained devices, using the Contiki operating system as the platform of choice.

The rest of this document is organized as follows. Section two discusses related work to service discovery in embedded networks; then in section three the planned development environment is discussed together with the compatible software i.e. Contiki OS. In section four present details are given for mDNS and in section five details for service discovery using mDNS are given. Section four and five serve as background for section six which describes in detail how mDNS can be implemented and how step by step a working mDNS application can be created for the targeted platform. Section seven describes the tools and procedure used for testing the implementation, including the results from each test. Section eight introduces the evaluation criteria and discusses how each criterion is tested and evaluated. Section nine describes possible usage scenarios of mDNS in embedded networks. Finally the last section describes the conclusion of the document.

2 Related Work

Service discovery protocols allow devices on the network and services offered by those devices to be automatically detected. Service and device discovery can be done in different ways, either by making use of some existing technology and standards or unconventionally by some alternative methods.

2.1 IEEE 802.15.4 beaconing

Frames are the basic unit of data transport and one of the four frame types used in IEEE 802.15.4 networks are beacon frames. There are two types of IEEE 802.15.4 networks according to beacon frames i.e. beacon-enabled networks and non-beacon networks. In beacon-enabled networks beacon frames are transmitted periodically to announce the presence of a network as well as help devices connected to such network to synchronize with each other by the usage of the beacon frames as synchronization signal. Intuitively one would think that, since these frames are used for synchronization and announcing the presence of a network, they can also be used for discovering devices and services offered by those devices by including the necessary information in the payload of the beacon frame. However beacon-enabled device is hard to implement and it requires highly processing power to meet the constrained timing events and processing of the beacon packets. Modifying the intended purpose of these frames by including records that contain data required for service discovery would make the frames more difficult to process and add unnecessary complexity. Additionally using the beacon frames when forwarding information for a service, announced by a device, further in the network would change the whole concept of beaconing. Thus it can be concluded that

adopting some other alternative service discovery technology by modifying existing one, would require every site in the world to install, learn, configure, operate and maintain some entirely new and unfamiliar server software. [7]

2.2 CoAP

CoAP follows the Representation State Transfer (REST) architecture which consists of clients initiating requests and servers processing the requests and returning appropriate responses. Requests and responses are built around the transfer of representations of resources which capture the state of that resource. Transactions between end points identified by Transaction IDs are initialized using request or notify messages. CoAP request resembles HTTP request, it is sent by a peer to request an action on a resource identified by a uniform resource identifier (URI) on the other peer. Notify message is the inverse of request i.e. a peer sends notify message to the other peer about a resource on the peer that sent the notify message identified by URI. CoAP end-point can be considered as an application process using only one port for CoAP message exchange which implies that a host may run any number of CoAP end-points.

In a CoAP request message the Version field and the Type flag indicating request are set to 0. For every transaction the TRANSACTION_ID variable is increased by one and the value is set in the ID field. The payload follows after the ID and magic byte header must be included in case multiple messages are packed into a single UDP datagram or over TCP. Magic byte indicates the length of the message in octets. In case the A bit is set in the request, a response message must be sent to the source IPv6 address and port of the request with the same ID. In a CoAP notify message the fields are set similarly to the request message the difference is that the Type flag is set to 2. In a CoAP response message the Version field is set to 0, the Type flag is set to 1 and the code field must contain a valid code, the codes are subset of HTTP response codes. The ID must be set to that of the corresponding request and additionally options and payload may be included. The length of the payload is taken from the options or it is calculated from the datagram length. CoAP Methods include GET, POST, PUT and DELETE which are easily mapped to HTTP methods. The GET method retrieves information of an identified resource, POST method creates a new resource under the requested URI, PUT updates the resource identified by an URI and DELETE deletes the resource identified by an URI.[25]

CoAP resource discovery offered by an end-point supports the constrained RESTful environment (CoRE) link format. The discovery mechanism provides URIs (links) for the resources offered and a collection of those links is carried as a resource itself. The link format is extended with specific machine to machine link parameters and a default interface is defined to discover resources described by these links. Links are carried as a message payload where each link conveys one

target URI as a URI-reference inside angle (“<>”) brackets. Context URI of a link is by default the URI of the resource which returns the link-format representation which implies that each link describes target resource hosted by server. Additionally there are CoRE link extensions in which specific target attributes are defined. These attributes describe information useful in accessing the target link such as resource name, interface description, content type, maximum size of the link indicated by the target URI and etcetera.[24]

2.3 Bonjour

Zero Configuration Networking (zeroconf), or Bonjour as Apple calls it, is a networking technology that allows devices and services to be automatically discovered. With Bonjour electronic devices can be easily added to an existing network, or instant networks of multiple devices can be created without multiple levels of configuration. Bonjour configures each device's IP settings and then makes the services available on each device easily discoverable by all the devices on the network. Bonjour is designed for networks that do not have central DNS server; it uses Multicast DNS Service Discovery (mDNS-SD) which is a combination of mDNS and DNS-SD standards. The mDNS-SD is query driven and retrieves the type of service, the name of the service, IP and port addresses and other optional information. Each device on the network receives these information and applications running on that device can use the information to create a list of offered services. The multicast protocol, which Bonjour makes use of, is designed to reduce the traffic on the network in a way that whenever a device queries the network for information and other device on the network responds, all the devices receive that response. Since each device receives the response and caches it, the device has no need for querying for that response which results in reduction in networking traffic. Bonjour is based on mDNS and DNS-SD which means that it is based on already well defined and widely used networking protocols. Because of that it inspires new ideas for device to device interaction. [1]

3 The Development Environment

Embedded devices are designed to perform one or few dedicated functions in the most efficient way. They are controlled by one or more main processing cores called microcontrollers. Contiki is an open source, highly portable, multitasking operating system for memory-efficient networked embedded systems and wireless sensor networks. It is designed for microcontrollers with small amounts of memory. The Contiki OS is compatible with different hardware platforms such as Crossbow TelosB, RedBee EconTAG and Atmel AVR Raven. The development platform of interest in our case is the AVR Raven, since development and testing

will be performed on this platform.

3.1 Contiki OS

Contiki is written in the C programming language and consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime. Contiki processes use light-weight protothreads that provide a linear, thread-like programming style on top of the event-driven kernel. Contiki also supports per-process optional multithreading and inter-process communication using message passing. It provides three types of memory management: regular `malloc()`, memory block allocation, and a managed memory allocator. Contiki provides both full IP networking and low-power radio communication mechanisms. Regarding the power-efficiency in order to provide a long sensor network lifetime, it is crucial to control and reduce the power consumption of each sensor node. Contiki provides a software-based power profiling mechanism that keeps track of the energy expenditure of each sensor node.

The Contiki implementation of 6LoWPAN does not run as separate process, it is called by the MAC process when a 6LoWPAN packet is received and by the TCP/IP process when an IPv6 packet needs to be sent. It is initialized from the MAC process, which calls `sicslowpan_init()` passing as an argument a pointer to the MAC driver structure. The main 6LoWPAN functions are implemented in the `sicslowpan.h` and `sicslowpan.c` files. When an IP packet is too big to fit in an 802.15.4 frame (after header compression), it is fragmented in to several packets which are then sent successively. These packets are formatted as defined in RFC 4944. The packet reassembly process in Contiki assumes that when packets are received they are not in order. As such, during reassembly of packets, any non-fragmented packets are discarded; the same applies to fragments from another packet.

Each program in Contiki must contain an appropriate Makefile. In the simple Makefile given below the `CONTIKI_PROJECT` variable is always set to the name of the C code file, everything else usually doesn't require any change.

```
CONTIKI_PROJECT = <source filename>
all: $(CONTIKI_PROJECT)
CONTIKI = ../..
include $(CONTIKI)/Makefile.include
```

Once the code is compiled using the command `make TARGET=<platform>` it can be uploaded to the targeted device using the appropriate method. The skeleton of a Contiki program is given as:

```
#include "contiki.h"
PROCESS(processName,"Text Description");
```

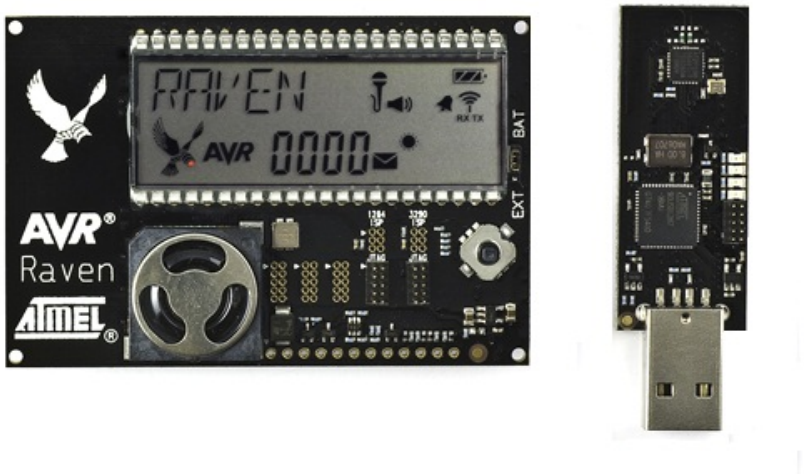

Figure 1: The AVR Raven development board and the RZUSB stick which can be used to access IEEE 802.15.4 based networks.

```
AUTOSTART_PROCESSES(&processName);
PROCESS_THREAD(processName, ev, data) {
    PROCESS_BEGIN();
    /* All thread code goes in here */
    PROCESS_END();
}
```

where the process is declared using `PROCESS()` function.

The `AUTOSTART_PROCESSES()` function takes pointers to processes and starts the protothreads once the Contiki operating system has finished loading. Each process is declared using `PROCESS_THREAD()` and the definition of thread itself goes between `PROCESS_BEGIN()` and `PROCESS_END()`. Since contiki has an event driven kernel, timers can be used to trigger events at regular intervals. Timers can be declared using`static struct etimer et` and set with `etimer_set(&et, CLOCK_SECOND)`. Additionaly other functions may depend on the hardware platform that is in use.[16, 6, 17]

3.2 AVR Raven

The Atmel AVR Raven development kit consists of a RZUSB stick and the AVR Raven itself. The RZUSB stick is an adapter which acts as Ethernet interface and can be used to access a IEEE 802.15.4 network from an IP network. The AVR Raven has a 2.4 GHz transceiver, two microcontrollers and LCD display. The ATMega3290P microcontroller is used to control the onboard LCD while the other

Figure 2: The Atmel JTAG ICE mkII programmer used to download machine code to the AVR Raven and perform in-system debugging via JTAG.

one ATMega1284P handles the high sensitivity transceiver AT86RF230. Communication between the two microcontrollers is possible whenever needed by using serial communication. The joystick and the temperature sensor are interfaced to the ATMega3290P microcontroller; however, if data is needed from these interfaces by the other ATMega1284P microcontroller, the first one can be programmed to collect and send this data to the second one using a serial line.

The ATMega3290P microcontroller offers 32 KB flash memory and 2KB internal RAM, on the other hand the ATMega1284P microcontroller has 128 KB flash memory and 16KB RAM. These microcontrollers can be programmed using the JTAGICE mkII programmer. The programmer is connected to a host machine using USB cable and at the same tame to the Raven platform using flat cable with an appropriate jack. This connection enables compiled binaries on the host machine to be transferred to the corresponding microcontroller. The AVR Raven board can either be powered by batteries or, for practical and development purposes, externally by connecting it to a 5V power source. In our case, USB cables have been made to power the AVR Raven boards.[2, 3]

3.3 mDNS and DNS-Service Discovery

DNS Service Discovery is a way of using standard DNS programming interfaces servers and packets to browse the network for services. DNS-Based Service Discovery is only peripherally related to Multicast DNS, in that the standard unicast DNS queries used by DNS-SD may also be performed using multicast when ap-

propriate, which is particularly beneficial in Zeroconf environments. [4, 5]

4 Multicast-DNS (mDNS)

Multicast DNS (mDNS) refers to clients performing DNS-like queries for DNS-like resource records by sending DNS-like UDP query and response packets over IP Multicast to UDP port 5353. Multicast DNS is a way of using familiar DNS programming interfaces, packet formats and operating semantics, in a small network where no conventional DNS server has been installed. The benefits of mDNS names are that they require little or no administration or configuration to set them up and they work when no infrastructure is present. [5]

DNS queries are fundamental part of the DNS protocol and are used by clients for requesting specific resource records. DNS query ending with ".local." or the ones for which no other DNS server is available are sent to the mDNS address 224.0.0.251 for IPv4 and FF02::FB for IPv6. The Multicast DNS host names are of the form "single-dns-label.local." and are limited to a length of 255 bytes. Multicast DNS uses UTF-8 to encode resource record names, additionally any text being represented internally in some other representation must be converted to canonical precomposed UTF-8 before being placed in any Multicast DNS packet. The reverse address mapping works in such a way that the DNS queries ending with "254.169.in-addr.arpa." are sent to the mDNS multicast address 224.0.0.251 for IPv4; similarly in the case of IPv6 addresses queries ending with "8.e.f.ip6.arpa.", "9.e.f.ip6.arpa.", "a.e.f.ip6.arpa.", and "b.e.f.ip6.arpa." are sent to FF02::FB.[5]

There are three kinds of Multicast DNS Queries. In the case of One-Shot Multicast DNS Queries the client directly sends the queries to 224.0.0.251:5353 for IPv4 or [FF02::FB]:5353 for IPv6 if the name that is queried falls within one of the reserved mDNS domains. However the client takes just the first response on the UDP port and because of that port 5353 should not be used. One-Shot Queries, Accumulating Multiple Responses are useful when more than one response is taken into consideration. In this case the client must use port 5353, wait in order to collect multiple responses and if needed retransmit the query if it seeks more responses. The last kind Continuous Multicast DNS Querying allows continuous monitoring for queries. The Querier sends the first two queries with an interval of at least one second in between and the time is increased afterwards by a factor of two until sixty minutes are reached. [5]

Multicast DNS has the ability for sending multiple questions in a single Multicast DNS query packet. However a newly connected device on the network having many questions to initially be answered may result in flooding the network with the responses it seeks. In order to avoid such situations Multicast DNS defines the top bit in the class field of a DNS question as the unicast response bit which indicates that the Querier is willing to accept unicast responses, such questions are referred

to as QU questions in contrast with QM questions requesting multicast responses. In the case of QU question the Responder should response with unicast response unless the last multicast announcement of that record is more than a quarter of its TTL. When Multicast DNS Querier wants to target a specific machine it sends the query via unicast, in such case the Responder should respond as it would for QU query.

The resource record sections of a Multicast DNS Responder message contain only records for which the Responder is authoritative. Moreover the responder must not place records from its cache, which have been learned from other Responders, in the resource records of a response packet. Resource record answers a question if the resource name matches the question name, if the record Type matches the question Type unless the question Type is ANY or the record Type is CNAME and also if the record Class matches the question Class unless the question Class is ANY. Moreover if the qtype is ANY the responder must respond with all of its records that match the query.The responder must respond only if it has non-null response to send or authoritatively knows that the record does not exist. The Multicast DNS responses must not contain questions or if they do the questions will be silently ignored. The responder should have a capability of delaying its responses up to 500ms which benefits in reducing collisions, however if the responder has the answer to every question from the query packet it should not delay any answers.[5]

When a Multicast DNS Responder starts up, wakes up or receives notification of link change it must perform Probing and Announcing. In the process of probing the responder first sends questions to test whether it has unique records such as host's addresses records. In order the number of questions to be reduced the probe query qtype should be set to ANY, also the probe query should make use of the capability of multiple questions in a single query. The probing queries are sent with 250ms delay after each of the first three, if after the third probing query no conflict responses are received we move to announcing. In the announcing process the responder sends gratuitous responses containing in the answer section all its newly registered shared or unique resource records. However in the case of shared records such as PTR records in service discovery they are simply put into the answer section and if they are unique records the most significant bit of the rrclass is set to one. When a Multicast DNS Responder, that owns unique records for which it is authoritative, receives response packet that contains the same name, rrtype and rrclass but inconsistent rdata, a conflict occurs. In the case of conflict the Multicast DNS Responder must reset its conflicted unique record to probing state in which the probing protocol decides a winner and loser that must stop using the name and reconfigure. The act of failing when probing results in a procedure that must be followed in order to clear any conflict. Namely the resource record name is being changed, by adding additional identification info, and probed again until unique name is found. In situations when after a minute of probing uniqueness is not obtained, the user should be informed.[5]

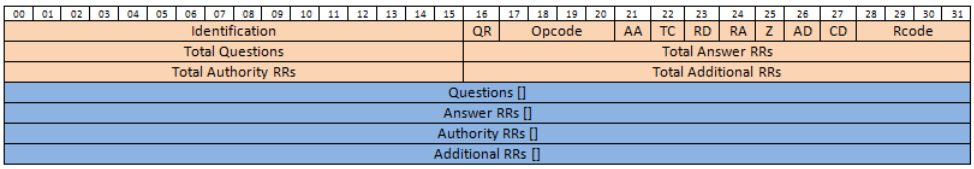

Figure 3: Multicast DNS message format

Multicast DNS Messages that are carried by UDP have size up to the IP MTU of the physical interface, the space needed for the IP header is 20 bytes for IPv4, 40 bytes for IPv6 and the UDP header is 8bytes. In case the resource record is too large to fit into single MTU sized packet, the Multicast DNS Responder should send the resource record in a single IP datagram using multiple IP fragments. However even if fragmentation is used the Multicast DNS packet must not exceed 9000 bytes. [5]

4.1 Multicast DNS message format

Multicast DNS messages just like DNS messages are composed of DNS header followed by the data indicated in the header. The format of the message can be seen on the picture (Figure 3) [12]. The reddish color indicates the message header and the blue color the message contents, the data. In multicast queries, responses and also gratuitous responses the Identification field (ID) should be set to zero on transmission; however on reception of any kind of response the ID should be ignored. This ignoring is due to hosts waking up or joining the network since their responses may contain useful data that is more important than the ID. However in the cases of unicast responses the ID must match the one from the query message. The Query/Response (QR) bit in MDNS messages must be set to zero for queries and in the case of responses to one. The OPCODE is always zero; the Authoritative Answer (AA) bit in query messages must be zero on transmission, however in response messages for Multicast Domains the AA bit is set to one, setting it to zero means that there is other place where better information can be found. Setting the Truncated (TC) bit to one in query messages indicates that additional known answer records may follow shortly and in such case the responder should record this fact and wait for those records before deciding to respond. In contrast if the TC bit is set to zero this means no additional known answers. RD stands for the recursion desired bit and RA for the recursion available bit. These bits are set to zero on transmission in both multicast query and response messages and are ignored on reception. The same applies to zero (Z) bit, authentic data (AD) bit and checking disabled (CD) bit. Namely these bits are all set to zero for queries and responses on transmission and are ignored on reception. Response code (RCODE) in both multicast queries and responses is set to zero on transmission. Messages which have non-zero RCODE are silently ignored. The next four fields indicate the number of: questions, answer resource records, authority resource records and

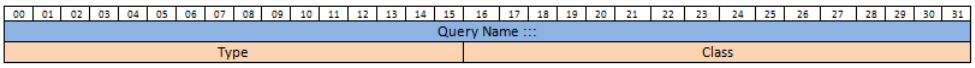

Figure 4: query format

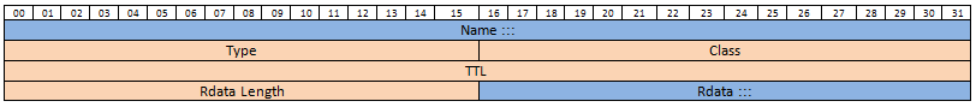

Figure 5: answer format

additional resource records. This concludes the header description; the next part of the message contains the data and it has variable length. The Questions[] field contains a list of zero or more Query structures, the Answer RRs[] field contains a list of zero or more Answer Resource Record structures, the Authority RRs[] field contains a list of zero or more Authority Resource Record structures and the Additional RRs[] a list of zero or more Additional Resource Record structures.[12, 5]

The query structure as shown in Figure 4 is composed of a query name which has variable length followed by two bytes indicating the Type of the resource record we are querying for[12]. Some of the types of interest to us are 12 for PTR record, 16 for TXT record, 28 for AAAA record and 33 for SRV record. The Type field is followed by two bytes indicating the Class which is usually set to one and indicates Class IN i.e. internet. Additionally the most significant bit of the Class field, if it is set to one indicates that a unicast response is preferred for the query, otherwise if it is set to zero a multicast response is expected.[5]

The answer, authority and additional resource records use the format shown in Figure 5, it is very similar to the query format.[12] It starts with the name of the resource record which has variable length, followed by the Type and Class field. However in the answer records the most significant bit is used to indicate if that record is a member of a unique resource record set and the entire resource record set has been sent together. A unique resource record set is a set where all the records with that name, type and class are in the ownership of a single responder i.e. at most one responder should answer to a query for that name, type and class.[5] The following fields are the time to live (TTL) which occupies four bytes, followed by two bytes indicating the length of the Rdata and the last field has a variable length and contains the data itself.

The format of the Rdata depends on the Type indicated in the resource record. For example in PTR type records the Rdata contains a name, which in service discovery is used to indicate the name of an instance of the service. In SRV type records the Rdata is composed of two bytes indicating priority, zero being used for the highest priority. This field is useful when multiple SRV records have the same Name and

only the record with the highest priority is taken into consideration. Furthermore for multiple records having the same priority the next two bytes that are indicating the weight are taken into consideration; however in most cases both of these fields are set to zero. This is followed by two bytes for the Port number and a Target which is a label with a variable length that indicates the hostname. TXT type record's Rdata contains additional information and AAAA type records contain 16 bytes Rdata that indicates the ipv6 address. . These example formats for the Rdata are for the types which are being used most of the time; the rest of the types follow similar principle, differing from each other in the Rdata by the information they carry.[5, 12]

4.1.1 Label encoding

All the names either in the query name field, the name field in the answer resource record or the names in the Rdata section follow a general encoding pattern. Namely if the user sees for example the name "www.google.com" this name is encoded as "3www6google3com0" in the packet. However one must be careful not to confuse the sizes that indicate the length of each name segment by their respective ascii characters. The zero indicates that the end of the name was reached.

In order to save space in the packets name compression mechanism is used when generating Multicast DNS packets. It works in a way that it replaces a segment or the complete name with a compact two-byte reference to the appearance of the same data somewhere earlier in the packet. The first byte gets a value of 192 or c0 in hex which indicates that the following part of the label is to be found on the location indicated by the second byte. Additionally names that appear within the rdata should be compressed if that rdata is in some of the following rrtypes: NS, CNAME, PTR, DNAME, SOA, MX, AFSDB, RT, KX, RP, PX and SRV. This introduces a specific difference with unicast DNS where name compression in the SRV record is not allowed. [5]

5 Service discovery using mDNS

Given a type of service and a domain name in which the client is looking for a service, mDNS service discovery allows the client to discover a list of named instances of that desired service using standard DNS queries.

DNS service discovery (DNS-SD) helps in registering a service, browsing for services and resolving service names to host names. Service instances are described using DNS SRV and DNS TXT record. SRV record has name of the form "<Instance>.<Service>.<Domain>" it contains the target host and port where the service instance can be reached. The TXT record having the same name introduces

additional information about the instance, in a structured form using key/value pairs. This is always true for DNS-SD services, they must have SRV and TXT records with the same name even if this means TXT record having single zero byte.[4]

Typical DNS SD TXT record has total size of 200 bytes but if needed the size can be extended to 400 bytes which fits in single 512 byte DNS message or further extended to 1300 bytes which will fit in a single 1500 byte Ethernet packet. DNS-SD uses the TXT records to store key/value pairs in the form “key=value”, everyone who is implementing DNS SD for some service should define the base set of key/value valid for that service otherwise any unknown keys will be ignored. The target host and port number given by the SRV record must not be duplicated in the TXT record. When a client discovers many instances of a service the TXT records give the client additional information so there is no need for opening TCP connection. The key should be unique and between one and nine characters long due to efficiency, because each string in the TXT record is limited to 255 bytes. If client receives TXT record having some key more than once it should take into consideration only the first occurrence of that key. Examining TXT record for a given key may result in four categories: attribute not present, attribute present with no value, attribute present with empty value and attribute present with non-empty value. TXT records are not case significant, however one must be careful with spaces since they are taken into consideration. [4]

Clients can discover service instances using query for DNS PTR record with a name in the form <Service><Domain>. The query returns list of zero or more SRV/TXT record pairs having the same name as mentioned before. The DNS PTR is a pointer from one name to another in the DNS namespace, PTR lookup for the name "<Service>.<Domain>" results in a list of zero or more PTR records providing service instance names in the form <Instance> . <Service> . <Domain> , this is called service instance enumeration. For example if we need to discover all service instances on the network that offer iTunes audio library, we send a query for a PTR with the name “_daap._tcp.local” (daap stands for digital audio access protocol). Another common example is when a web browser wants to discover all advertised web pages on the local network, it sends a query for “_http._tcp.local”. So in case more than one server offers the same set of services it doesn’t matter to the client which server it uses and it selects one according to the weight and priority rules. The names resulting from the PTR lookup are presented to the user in a list for the user to select one or possibly more. Typically only the first label is shown , the user friendly <Instance> portion of the name. In the common case, the <Service> and <Domain> are already known to the client software. This whole mechanism for DNS-SD combined with multicast DNS is capable of providing zero-configuration operation.

The <Instance> part of the name is a single user friendly DNS label the only restriction for this label is using ASCII control characters, regarding the length of

the label there is a limitation between 1 and 63 bytes, including the length byte 64 bytes. UTF-8 characters require up to four bytes for encoding which sums up to a worst case label composed of fifteen Unicode characters.

The <Service> part of the instance name consists of a pair of DNS labels where the first label is the application protocol name and the second label indicates the transport protocol used by the application namely either "_tcp" for application protocols running over tcp or "_udp" for all others. The application protocol name <app> size is limited to 15 characters not counting the mandatory underscore and only lower case letters, digits and hyphens can be used.

The <Domain> portion specifies the DNS subdomain in which the service names are registered. For Multicast DNS the <parentdomain> part of the domain is set to be “local.”. Moreover service names are not host names and thus rich-text service subdomains are encouraged. Domain names can have size of up to 255 bytes plus one byte for the terminating root label at the end. [4]

5.1 Service name resolution

The service name resolution works in a way that if a client needs to contact a particular service name from the list obtained via service instance enumeration, the client sends a query for the SRV and TXT records of that name. However there are some additional rules that reduce the traffic on a network by including all necessary resource records in the same packet, more on this later. As mentioned earlier in this paper the SRV gives the port number and target host name where the service may be found. Using SRV records allows each host to allocate its available port numbers dynamically to the services actually running on that host that need them and then advertise them via SRV records. In the common case each service instance is described by exactly one SRV record where both the priority and weight fields are set to zero.

DNS-SD uses domain names in the following forms:

<app>._tcp.<servicedomain>.<parentdomain>.
<Instance>.<app>._tcp.<servicedomain>.<parentdomain>.
<sub>._sub.<app>._tcp.<servicedomain>.<parentdomain>.

Where the first form is used for PTR queries i.e. for service instance enumeration. Query having name in the second form is a query for the SRV and the TXT record for that service instance name. The third form shows subtype browsing name, this form of a name is used for subtype instance enumeration. For example some devices such as printers use web-based user interfaces. These web interfaces are subtype of “_http._tcp”, so when we open the interface of a device the web browser issues a query for “_devicetype._sub._http._tcp.local”, in the case of a printer the “_devicetype” becomes “_printer”. The subtype identifier is allowed to be 63 bytes,

including "_sub" and the length bit it sums up to 69 bytes.

In order for a client to find what type of services are being advertised on the network, it sends a meta query for a PTR record with the name "_services.dns-sd._udp.local". This kind of a query usually is used for problem diagnoses and by network management tools. The meta query results in a set of PTR records where the record data is the two-label <Service> name followed by the same domain i.e. local for mDNS. These two-label service types can later be used in the queries for service instance enumeration. [4]

5.2 mDNS additional record generation

In order to reduce the traffic on the network and reduce the number of packets being exchanged between server and a client i.e. service provider and service seeker the DNS-SD RFC proposes service providers to include all the necessary records in the additional resource record section of the packet. The rules for the additional record generation are the following:

1. For every PTR record in the answer section of the packet the sender should also include, in the additional resource record section, the SRV and TXT records named in the PTR Rdata and also the A and AAAA records named in the SRV Rdata.

2. Analogously to the previous one, for every SRV record included in a response packet the server/responder should also include all address records i.e. A and AAAA named in the SRV Rdata.

However clients must be capable of functioning correctly even if the server fails to generate the needed additional records. This is solved by sending queries for the missing, but needed records.[4, 5]

6 Implementation

Considering some of the currently offered methods for service discovery and taking into consideration the ubiquitous deployment that DNS already enjoys it is obvious that the ideal and prospective way for device and service discovery in 6LoWPAN is by implementing mDNS with DNS-SD. The following sections from this paper show how to combine everything that was described so far for mDNS service discovery and implement it on the targeted AVR Raven platform which is running Contiki OS. This way motes running Contiki can use the existing IPv6 network and be integrated with the tools that are already using mDNS and DNS-SD.

Figure 6 is an overview visualization of the whole implementation. We have applications running on the mote that need to discover and use a service offered on the network or applications that want to offer i.e. advertise service on the network.

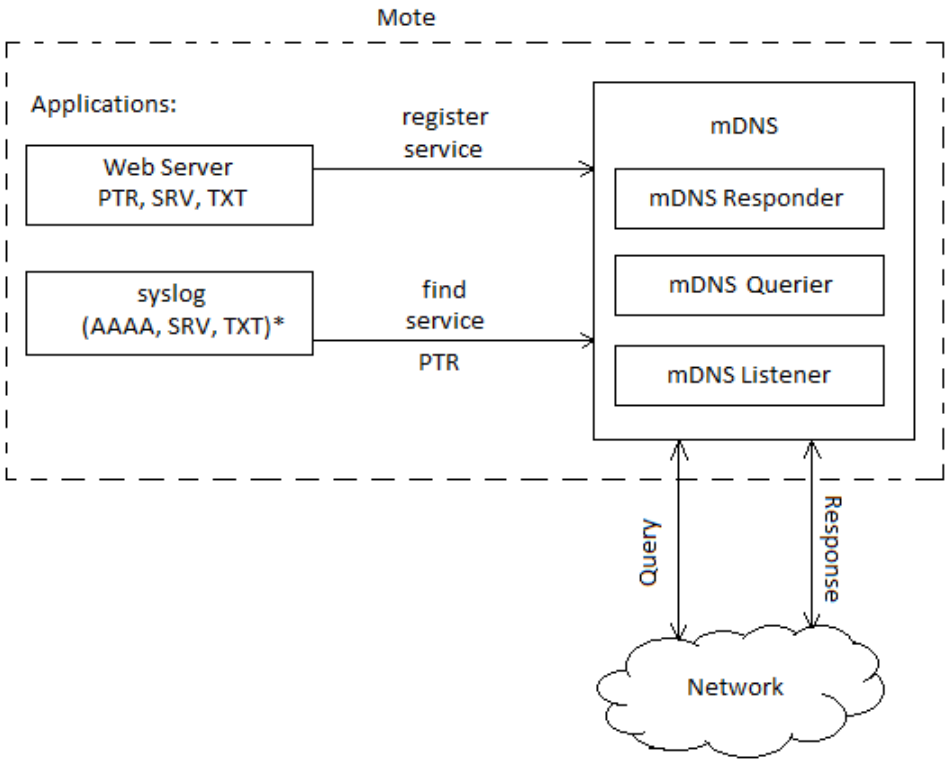

Figure 6: Implementation overview

In this particular case the web server offers a service and the syslog seeks for a service. The service discovery and service advertisement is done by using mDNS application that is composed of listener, querier and responder. The mDNS application takes the necessary parameters from the service offerer or service seeker applications running on the mote and advertises the service. It combines all the information from the service offerer in a DNS packet or tries to find out and give the information to the service seeker that it needs. Additionally when communicating with the network, in the DNS packets that it sends, it includes information about the mote itself. For example the AAAA record when sending a service advertisement or the PTR record that introduces the service type i.e. one that answers the meta query for "_services.dns-sd._udp.local" described in section 5.1.

The mDNS Listener listens for DNS packets that are coming from the network on port 5353. When it receives a DNS packet the mDNS listener can do one of two things, depending on whether the DNS packet contains queries or answers. When it receives a DNS packet that contains one or more queries it processes the packet and all questions in it. When the questions are extracted, the mDNS application can respond to those questions for which it knows the answers. However since the mote has limited resources, in this implementation the mDNS application will respond only to queries concerning applications that are running on the mote and not to queries for which it has previously learned the answers from the network. The second case is when the mDNS listener receives DNS packet that contains answers. In this case it should process the packet content, for which the applications running on the mote are interested, and store the data in data structures. Later the applications running on the mote that are expecting this data can start using it by looking into these data structures that store the information.

The mDNS Querier sends out DNS packets that contain queries. The queries are sent either when the mDNS application wants to discover a service on the network, by sending query for a PTR record with the name of the service type, or when the application needs more information for a particular service instance that was previously offered to the mote. The later is done by sending queries for the type of resource records that are missing for the service instance. The query can either contain the instance name, when some resource records describing the instance are missing, or the host name when resource records for the host offering the service are missing, such as the AAAA record.

The mDNS Responder is used when the mDNS application wants to announce a service on the network, by combining all the necessary resource records for that service and sending them in a DNS packet. The other case for which the responder is used, is when the listener receives a query concerning some service hosted on the mote. Usually this is the case when someone on the network is looking for a service of the same type that is being offered on the mote or if there is a query that is asking for more information on a service instance that was previously offered by the mote.

We shall now examine how the mDNS application works in detail, we start with how applications on the mote announce a service on the network.

An application running on the mote that wants to advertise a service, gives a structure composed of PTR, SRV and TXT record to the mDNS application. These three types of resource records are the necessary ones and specific to the service provider application. Just a brief clarification, the PTR record that is included in this structure is a pointer to an instance of the service type offered. Then the mDNS application takes these resource records and constructs the DNS packet that needs to be send. However the three records that were given previously to the mDNS application are not enough for the service to be advertised successfully. In addition to those three the mDNS application includes a PTR record with the name "_services.dns-sd._udp.local" and in the Rdata, the service type. In the case for web service the Rdata would contain "_http._tcp.local", this is explained in detail in section 5.1 from this paper. Moreover it also includes an AAAA record that has the IPv6 address of the mote and possibly it can include NSEC records to indicate completeness. More information on the NSEC records and why are they used, follows in section 6.3. After the mDNS application has a complete list of all the records needed, it initializes a DNS header structure. The data with which the header needs to be initialized is explained in detail in section 4.1, but in simple words all the header fields contain the same data for any kind of mDNS packet, except for the QR field which indicates if the packet contains queries or responses and off course the four fields that indicate the number of queries and resource records. How the resource records are distributed among the three possible sections of the DNS packet that hold resource records is explained in section 5.2. Worth to mention is that, if the sender decides to include NSEC records, the testing phase described in section 7 has shown that, they should be put in the additional RRs section. Finally when the mDNS application has finished filling the header of the DNS packet with the necessary information, the mDNS application combines all of the data in a single buffer in the order defined i.e. header followed by two PTR records in the answer resource record section, followed by all the rest resource records in the additional answer RRs section. After the buffer is filled it is being send, the sending is done as described in the following section (Section 6.1).

An application running on the mote that wants to discover and use a certain type of service that is being advertised on the network, gives a structure to the mDNS application that holds a PTR record with the name of the type of service that the application needs. In addition to the PTR record it also keeps empty records for SRV, TXT and AAAA; which need to be filled by the mDNS application. As soon as they have data in them i.e. they are not NULL, the application that looks for a service will have all the data that it needs and can start using the service on the network. So in the example case shown in Figure 6 the syslog application will ask for a service of the type "_syslog._tcp.local" by filling the PTR record and giving it to the mDNS application which then will create a query for PTR record whose name field contains the service type name i.e. the query will ask for an instance

of that type of service on the network. As described in section 5 of this paper, anyone listening on the network and offering a service of the same type as in the query, should respond to this query and provide all the needed resource records i.e. PTR, SRV, TXT and AAAA. Similarly as the application on the mote offering a service advertises an instance of that service. However in the case when not all of the needed resource records are included in the packet which was received from the network, the mDNS application should send a query or queries for the ones that are missing but are needed. As soon as all needed information for a service has been discovered, the application on the mote that needs it can start using it.[4]

6.1 Connection

As defined in section 4 from this paper mDNS refers to clients performing DNS-like queries for DNS like resource records by sending DNS-like UDP query and response packets over IP Multicast to UDP port 5353. Moreover the mDNS specification states that all packets in mDNS need to be sent to the IPv6 address FF02::FB, unless in exceptional cases the most significant bit of the Class field in the query is set to one to indicate that unicast response is required. For simplicity in the implementation these cases will be ignored. The starting point of the implementation is obviously creating UDP connection over port 5353 that will send packets to the specified IPv6 address. In the Contiki OS this is done in the following way:

```
struct uip_udp_conn *client_conn;
uip_ipaddr_t ipaddr;
uip_ip6addr(ipaddr,0xFF02,0,0,0,0,0,0,0x00FB);
client_conn=udp_new(&ipaddr, UIP_HTONS(5353), NULL);
udp_bind (client_conn, UIP_HTONS(5353));
```

After the lines above are executed, the program is ready to send UDP packets using the function bellow, where buffer contains the data that we want to put in the UDP packet; in our case the DNS packet.

```
uip_udp_packet_send(client_conn, buffer, bufferSize);
```

6.2 DNS packet

What we need for communication between a server and a client are structures that will hold the data that is being transferred. In this case we are using DNS packet structure which, as defined in section 4.1 from this paper, is composed of DNS header followed by queries and answers. The number of queries and different answers is indicated in the header itself. Since Contiki uses the C programming language the basic idea is to define equivalent data structures that will hold and represent the data for each field in the header, query or answer section from the

DNS packet. This allows us to work with the fields of the packet in our program and easily fill in the data in order at the end to create a proper DNS packet that will hold all the data that we want. The following subsections describe data structures that are capable of holding the data for DNS header, query and answer.

6.2.1 DNS header

As mentioned earlier for service advertisements the headers of the DNS packets are all identical since for successfull service advertisement from the mote all DNS packets contain same type and number of resource records. This fact allows a static header to be used whenever the application on the mote wants to send service advertisement. However in order to provide complete functionality as described in the implementation section, the application should use the generic header structure which is about to be defined, and initialize it on demand. The C code bellow defines the structure mdns_header_t which is capable of storing data for all fields for a DNS header, this is a C code representation for the DNS header format. The format and how each field should be set is described in detail in section 4.1 from this paper.

```
typedef struct {
  uint16_t Identification;
  union {
    struct {
      unsigned char Rcode :  4;
      unsigned char CD :     1;
      unsigned char AD :     1;
      unsigned char Z :      1;
      unsigned char RA :     1;
      unsigned char RD :     1;
      unsigned char TC :     1;
      unsigned char AA :     1;
      unsigned char Opcode : 4;
      unsigned char QR :     1;
    };
    uint16_t Flags;
  };
  uint16_t TotQuestions;
  uint16_t TotAnsRR;
  uint16_t TotAuthRR;
  uint16_t TotAddRR;
} mdns_header_t;
```

6.2.2 DNS body (queries and answers)

After the header in a DNS packet comes the body which is composed of the queries and/or the answers. The first structure defined bellow represents the format for DNS queries. Namely a query contains a variable length name and two byte type and class fields. You may have noticed from section 4.1 that the most significant bit from the class field is used to indicate unicast or multicast preference for an answer. Normally we deal with multicast which means this bit should be set to zero; however if we need to set it to one we can simply use the bitwise shift operator (<<).

```
struct query{
  char *qName;
  uint16_t qType;
  uint16_t qClass;
};
```

The next structure represents the format for the resource records. The first three fields are identical as in the query, however in the resource records the most significant bit of the class is used to indicate unique resource record set, more details are given in section 4.1. Additionally these fields are followed by four byte time to live (defined in seconds), two byte data length which indicates the length of the last field and the last field which has variable length and has the resource record data. The rrData format depends on the type of the resource record and the different formats for PTR, SRV and AAA are given in section 4.1, TXT is explained in detail in section 5 and the NSEC format is about to be described in the following section. Since rrData is a simple byte pointer the specific contents for each type of resource records can be combined in simple buffer to which this pointer will point to.

```
struct rr{
  char *rrName;
  uint16_t rrType;
  uint16_t rrClass;
  uint32_t ttl;
  uint16_t rrDataLength;
  uint8_t *rrData;
};
```

6.3 Advertisements

What is needed for a successful advertisement of a service are two PTR records and one TXT, SRV and AAAA record. Figure 7 shows a Wireshark capture of a packet used to advertise a web service. This packet was captured during the testing

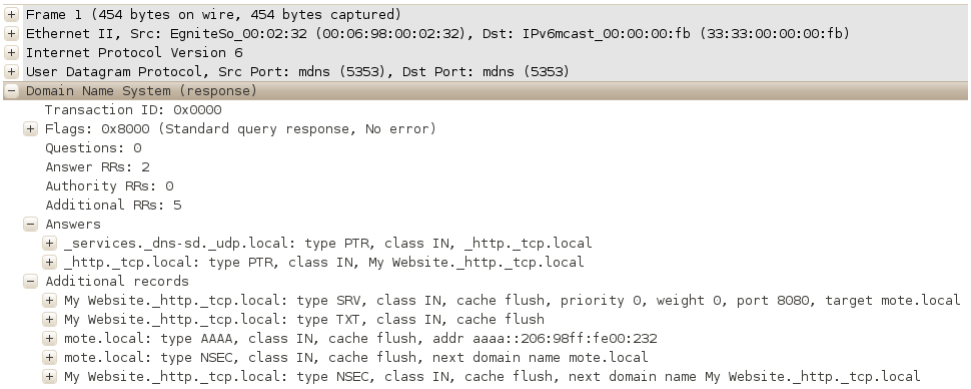

Figure 7: DNS packet screenshot from successful service advertisement

phase, more details on the testing and on Wireshark are given in section 7. The first PTR record has the name "_services._dns-sd._udp.local" which means that this is an answer to the meta query described in section 5.1 that is used for finding out the types of the offered services. So the first PTR record in the Rdata section has to contain the name of the type of the service in case this is the first advertisement of a service from that type on the network. In the example the Rdata of the first PTR record is set to "_http._tcp.local", which is the type for web service. The second PTR's name is the type of the service that was in the Rdata of the first PTR i.e. "_http._tcp.local" and the Rdata of this PTR record has to contain the instance name of the offered service, in the example "My Website._http._tcp.local". At this point the receiver of these records would have found out that there is a web service type on the network and somewhere someone offers an instance of this type of service. So next would be to answer the who and where questions and for this purpose we have the SRV,TXT and AAAA records that give information about the host of the service, the service and the location of the service. The SRV record's name is the instance name from the second PTR and the SRV record in its Rdata gives information about the host of that instance i.e. the port where this instance can be accessed on the host and the host name indicated as target. In the example the target is "mote.local". Additionally the TXT record again having the name of the instance introduces possible additional information, in its Rdata, for the service instance. Finally there is the AAAA record that contains the location i.e. the IPv6 address of the target from the SRV record.

You may have noticed the transitional property i.e. how the PTR and SRV records introduce some new data in their Rdata fields and then we get another resource record that gives additional information about the previous data. This is no coincidence and in order to indicate completeness i.e. close the transition loop two additional NSEC records are included, one for the service instance branch and one for the target. However the testing phase has shown that the service is being registered by service discovery applications with or without the NSEC records.

NSEC resource record contains a list of resource record types that have the same name as the NSEC resource record and by implication any resource record type not in the list does not exist which is a security feature. However, it does not matter how many resource records of some type we have with the same name since we need to mention the type of those resource records exactly once in an NSEC resource record that has the same name. The Rdata field of the NSEC records is composed of a label which is the name of the NSEC record itself and is followed by variable length encoding that indicates the types of the resource records with the same name.

6.3.1 NSEC bitmap encoding

The resource record type field is two bytes long which implies that there can be $2^{15} + 2^{14} ... + 2^{0}$different types and this number is split into blocks of 256. For example the type AAAA which is 28 belongs in the first 256 block and is indicated by 0 which means that the first byte from the encoding is 0. The second byte indicates the length of the bitmap that is the length of the bitmap for the resource record type number. So first we need to look into the encoding in order to determine the length. The bitmap is composed in such a way that we indicate with 1 the bit which is on the type position and not to forget we need to count from left to right since this is in network byte order. For example if the type is 28 we have bitmap 00000000000000000000000000001 (27 zeros and the 28th is indicated by 1) however we need to encode the bitmap using bytes, so for this example there are 28 bits which is rounded to 32 bits i.e. 4 bytes and thus the bitmap for this type will look like: 00000000 00000000 00000000 00010000. That is in network bit order, the reversed is 00001000 and is equal to 8 in hexadecimal, that is why 8 was shown for AAAA in Wireshark during the testing phase. In short the formula for the encoding in the Rdata of NSEC records is Type Bit Maps Field = (Window Block # | Bitmap Length | Bitmap), where | means concatenation. [13]

6.4 Message processing

In a similar way as a buffer is filled in with all the data from the DNS packet structures before an answer or query is sent, every time a new packet arrives it is received byte by byte and those bytes are filled in a buffer that needs to be processed in order to compose a meaningful information from all the bytes in this buffer. What is needed for composing a meaningful data from the stream of received bytes is a parser that will process the header and all the queries and answers.

Processing of the header is essential since the data that the header holds is used when processing the queries and answers. The format of the header is given in section 4.1, shown in Figure 3, and the corresponding C code structure representation is shown in section 6.2.1 . Since the smallest piece of data that can be represented

in C code is a single byte, we proceed by analyzing the buffer, that contains the received DNS packet, byte by byte. The idea is to use a pointer to each byte in the buffer starting from the beginning and moving the pointer towards the end of the buffer. But on the way as we move the pointer the information that the bytes contain is combined correspondingly to the format of the DNS packet into their equivalent C data structure representations. However the third and fourth byte from the header contain pieces of information that are shorter than a single byte. This is not a problem since we can analyze and extract the corresponding segments from the two bytes using bitwise operators in C.

The queries and answers always start at a fixed location in a DNS packet which is at the twelfth byte. By that point we would have processed the entire header, and since the header shows how many queries and answers the packet contains, we can proceed by iterating through the same functions that we define for analyzing each query or answer. The functions for processing queries and answers are based on the format shown in figure 4 and 5 respectively, and their equivalent C code structures are described in section 6.2.2 . The processing of a query or answer is done in a same way as the header i.e. byte by byte. What answers and queries have in common is the processing of their labels i.e. the names and parts of the Rdata section for some types of resource records that also contain label in this part. The label processing is done using a function which composes an entire label by combining all the bytes which are part of that label. How labels are encoded is described in section 4.1.1, in case part or the whole label is compressed, a function is used that receives as a parameter the location in the packet where the rest of the label can be found. This function processes the compressed part of the label and if at the location where the compressed part of the label can be located, the function comes across another compressed part (pointer to another location), it recursively calls itself with the new location. This action is repeated until the end of the label is reached, which is a byte that has a zero value.

The two paragraphs above described a way of processing the received DNS packets by obtaining the values for each field from the packet and filling in their equivalent C code data structures. After these structures are filled with all the data, this data can be easily used by the mDNS application.

7 Testing

The testing phase has proven very helpful in many situations during the development of the implementation of mDNS. The tools that were used for testing are Bonjour SDK, iTunes, Wireshark, avahi-daemon and avahi-browse.

Bonjour represents Apple's Zeroconf management tool, it provides framework for using DNS-SD. Bonjour offers a version for Windows and also a Windows SDK. Figure 8 shows a running console application that is offered as an example in the

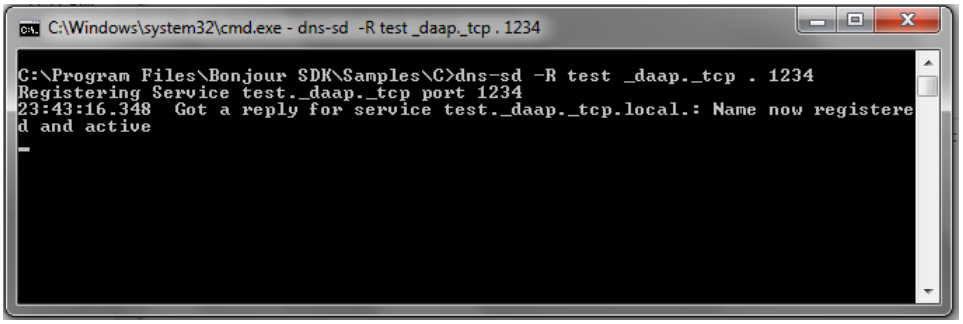

Figure 8: Bonjour SDK example service advertisement on IPv4

Bonjour SDK for Windows and is used for advertising a service on the network. The purpose of testing and running this example is to get the idea of how service advertisement works in practice. The example shows that we are announcing the instance "test._daap._tcp.local" which is an instance of a DAAP type of service which stands for digital audio access protocol. This means that we are advertising a shared music library, and the result can be seen if we run iTunes i.e. a shared library will appear with the name "test". This is interesting but what we are actually after is the DNS packet that is used to advertise this service. [18]

In order to see packets that are being transferred on the network we use the Wireshark tool which is a free and open-source packet analyzer.[23] This allows us to see the content of the packet so that we can get the overall idea of what is happening on the network. As it can be seen on Figure 9 the packet for the test service contains one answer of type PTR which is a pointer to the instance of that type of service and then in the additional records we have the SRV and TXT records just as expected; additionally there are A and AAAA type of records which are associated with the host name and for completeness the two NSEC records. This capture gives a good starting point for the implementation of mDNS in 6LoWPAN.

Avahi is the Bonjour equivalent for Linux operating systems and since the Contiki OS comes in a bundle with Ubuntu, for testing the implementation during the development Avahi was used together with Wireshark. Avahi is a system which facilitates service discovery, the avahi-daemon registers local IP addresses and static services using mDNS and DNS-SD principles.[20, 9] In order to see messages from the avahi-daemon for the packets on the network one must first stop the running avahi-daemon process with the command "sudo stop avahi-daemon" and then re-run it and leave the terminal open. This has proven useful during testing and debugging when the structure of the DNS packet is correct but some of the data inside the packet is wrong i.e. when Wireshark shows correctly formed packets but the service is not registered. In this case the terminal on which avahi-daemon is running shows the message "Packet too short or invalid while reading response record". Browsing and displaying the found services is done using avahi-browse,

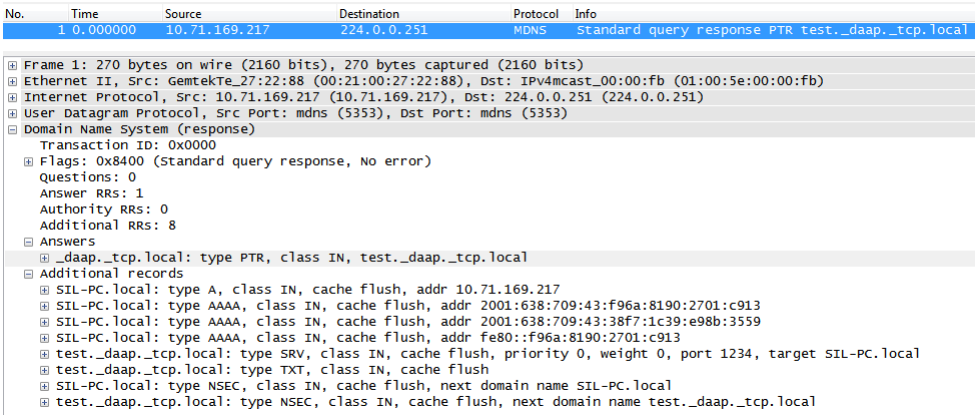

Figure 9: Wireshark capture of the packet used for advertising the example daap type service with Bonjour SDK

in order to see all services that are being advertised on the network the -a flag is used. To check the information on the host or the service itself i.e. whether the contents of the SRV,TXT and any address records are correct, the -r flag should be used which stands for resolve. So for example the command “sudo avahi-browse -a -r” will give detailed information for all services offered on the network and if we are interested in a particular type of service the -a flag should be replaced with the type of the service that we are loking for.

Testing was done thoroughly through the development of the implementation for mDNS and DNS-SD. The development started by implementing a service advertisement application, the goal was to successfully advertise some type of service on an IPv6 network, and to see the service using avahi-browse. After completion of this task the next step was to implement a listener that will process the received DNS packets, this was tested by cross checking the data that the listener parses with the results for the same packet in Wireshark. The results from the implementation and testing are described in the following two sections.

7.1 minimal-net

Compiling and uploading the code to the mote using the Atmel JTAG ICE mkII programmer each time a change is made to the code in order to test that change is a very slow way of development. This is the reason why during the development minimal-net was used as the TARGET i.e. minimal-net is simulating as close as possible the avr-raven platform and allows us to test the application on our computer. In order to use the avahi tool some modifications need to be made to the configuration file for the avahi-daemon. Namely what needs to be edited in this file are the network interfaces which avahi allows, minimal-net creates a virtual interface named tap0 and this name needs to be added in the allow-interfaces

list in the conf file (“allow-interfaces=tap0, eth9”). The conf file is located at “/etc/avahi/avahi-daemon.conf”, when editing the file the use of IPv4 can be excluded (“use-ipv4=no”) since we are interested only in IPv6 (“use-ipv6=yes”).

Creating the application for service advertisement was not straight forward and a lot of testing and debugging using the described tools had to be made. Composing an empty DNS packet, only with a header that has zero queries and answers was simple; however adding the answers to the packet caused some problems. First of all in the early implementation an entire structure holding the header and all the answers was given to the uip_udp_packet_send(3) function this way of sending worked only when a single structure was given as a parameter to the function i.e. just the header structure. When multiple structures were included and then the structure wrapping all the packets was given as a parameter to the function used for sending, the result was unsuccessful. Using Wireshark has shown that certain bytes from the packet were constantly changing and malformed packets were detected, thus it was concluded that a single buffer needs to be used as a parameter to the function and in this buffer all of the bytes from the structures were copied in the respective order. This solved the problem with the malformed packets but the service was still not showing up in avahi-browse. Then came the question of whether avahi is listening at all on the interface for minimal-net i.e. on tap0. This was the point when it was realized that the configuration file for the avahi daemon had to be edited as described earlier, to include tap0 interface. This partially solved the problem but at this point avahi-daemon was giving the error message “Packet too short or invalid while reading response record. (Maybe a UTF-8 problem?)”. After a long debugging session and adding some content in the TXT record, the service finally showed up in avahi-browse; however when running avahi-browse with the flag -r for resolving in the details the information for the IPv6 address was not included and the avahi-daemon was still producing the same error message. After closely investigating the TXT record and the AAAA record since they were one after the other, a length miscalculation was discovered. This was the final fix before the service was successfully advertised. The final results from running “avahi-browse -a -r” are shown in Figure 10 and the Wireshark capture for this packet was shown earlier in Figure 7.

The second most important part of the implementation was processing the received DNS packets, which is the reversed and very similar way of composing and sending out packets. The testing of this section was done by simple printf statements in the code and by comparing the output on the terminal with the Wireshark captures for the same packets. An example result from a processed DNS packet can be seen in Figure 11. Since we are interested only in a certain types of resource records, when a record which has a type that is not of interest, is received , the Rdata section of that record is ignored.

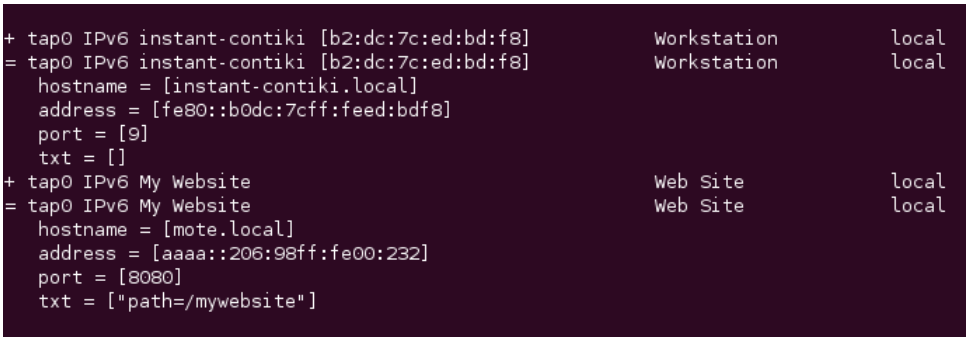

Figure 10: Successful advertisement of _http._tcp type of service on tap0

7.2 avr-raven

The code was compiled and uploaded to the avr-raven only at a point when the testing for the minimal-net was successful. Since the minimal-net simulates the avr-raven platform as closely as possible but not entirely, when compiling for the TARGET avr-raven some of the things may be different and thus the result needs to be tested using the same tools as previously i.e Wireshark and Avahi.

When the code for the successful service advertisement was compiled for the avr-raven platform and uploaded to the mote the only difference in the result was in the AAAA record i.e. the way the IPv6 address is obtained on the mote is different and thus this resulted in all zero fields for the address in this record. This problem was fixed by using the following code for obtaining the IPv6 address.

```
uip_ds6_addr_t *myaddr;
myaddr = uip_ds6_get_global(ADDR_PREFERRED);
if(myaddr != NULL){
  PRINTF("Global address: ");
  PRINT6ADDR(&myaddr->ipaddr);
  PRINTF("\n");
}else{
  PRINTF("No Global Address\n");
}
```

The application for processing received packets was not tested on the mote due to the way this application was tested earlier which is described in the previous section. Namely the output produced by the printf statements is too long to be redirected and displayed on the mote's screen.

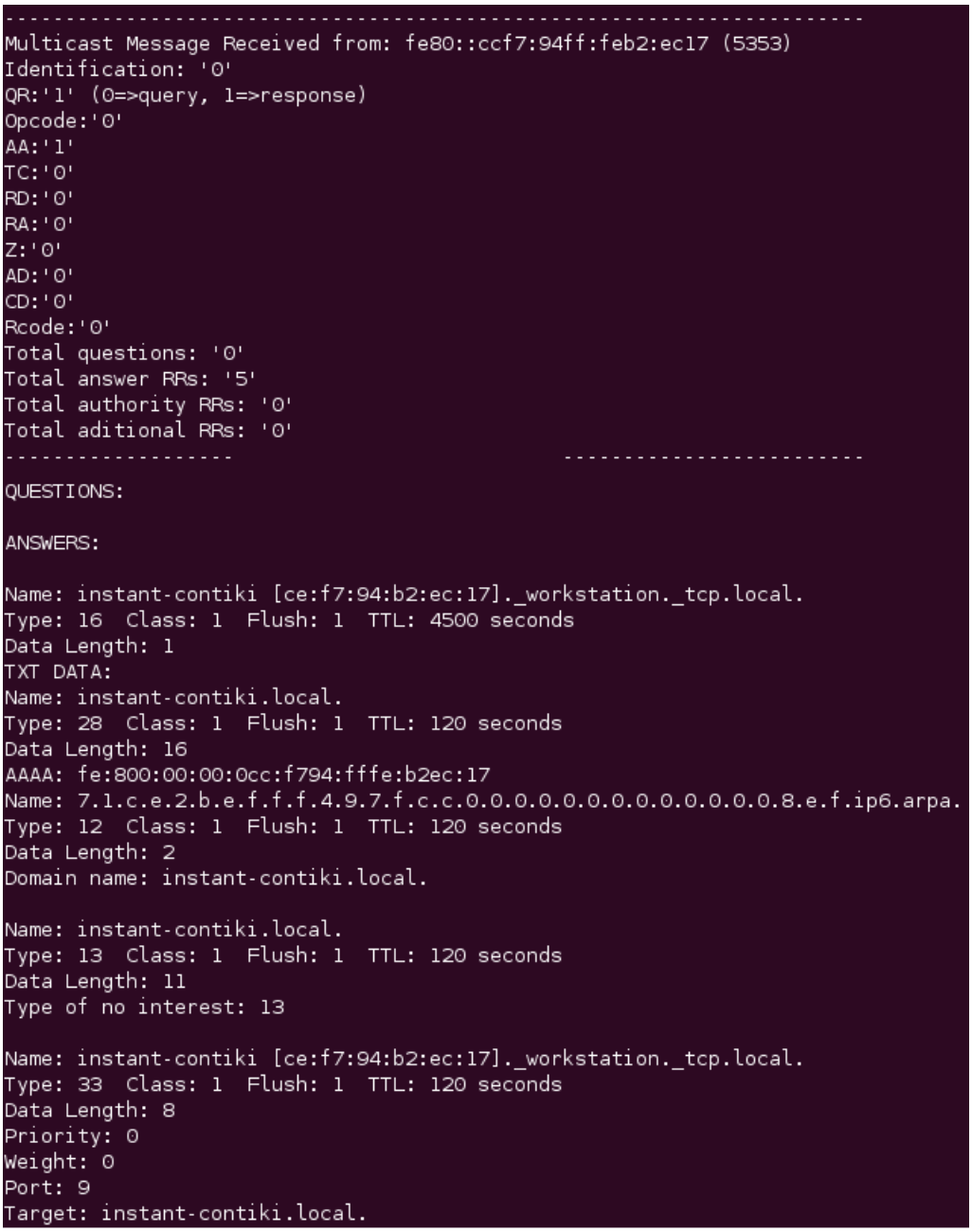

Figure 11: Example terminal results from processing received DNS packet

8 Evaluation

The goal of the project was to make use of mDNS and DNS-SD principles for performing device and service discovery in 6lowpan networks. Evaluating if this was accomplished should be done according to relevant evaluation criteria. First of all the most important criterion is interoperability i.e. the goal was to make use of the existing network and mDNS tools. Taking into consideration the limited memory resources of the targeted platform, the second evaluation criterion is the memory (Flash and RAM) and code footprint. Increasing the range may result in different latencies which implies that also the time, needed to perform device and service discovery under different circumstances, should be taken into account. This concludes the evaluation criteria of the project.

8.1 Interoperability

This criterion was successfully met as it can be seen in the testing phase. The goal was to create an mDNS application for the mote that will allow for the mote to be integrated with the existing mDNS and DNS-SD mechanisms. After many tryouts a working mDNS application was developed and this was proven by using already existing mDNS service discovery tools such as avahi. Namely the service advertised by the mote showed up in avahi-browse and all the details for the service were correct.

8.2 Memory usage

In order to obtain the memory usage of an implementation in Contiki it is necessary to first obtain the base memory usage of the basic Contiki OS. This is done by creating an application that does not do anything except that it starts a process and immediately stops it. This application is compiled with the "make TARGET=avr-raven <app-name>" command and then the "avr-size" utility is used for measuring the size of the application. The "avr-size" application provides us with the amount of Program, Data and EEPROM memory that is occupied. The Program memory contains the ".text", ".data" and ".bootloader" sections, which is basically an indication of the total Flash memory that is occupied. By analyzing the "contiki-avr-raven.map" file we can tell that the ".bootloader" section takes up 2KB of Flash memory. The results of a blank Contiki application are shown below:

```
AVR Memory Usage
----------------
Device: Unknown
Program: 39044 bytes
(.text + .data + .bootloader)
```

```
Data: 10961 bytes
(.data + .bss + .noinit)
EEPROM: 10 bytes
(.eeprom)
```

We already know what the ".bootloader" takes up, so the above means that in case of the basic Contiki OS ".text" and ".data" sections occupy about 36 KB. The results also show us that the Data memory that is being occupied is about 10.7KB. This means that the static RAM occupied by Contiki is 10.7KB and therefore the Flash occupied by Contiki is 25.3KB (this includes all constants, constant strings, etc.).

Now that we know what the Contiki OS occupies, we can compile the mDNS applications to see how much memory they take up and then subtract the base figures from that to obtain the memory usage of our implementation. So, for advertising a web server the mDNS application occupies 7.2KB in Flash memory and 0.2KB of static RAM. For processing the incoming queries and responses the mDNS application occupies 3.6KB of Flash memory and 0.7KB of static RAM. However, these figures are based on the mDNS application being split into two separate components for testing purposes i.e. one handling the advertisements and the other handling query and response processing. These figures could be reduced by combining the applications so that resources are shared. Further reduction could be done by using Flash memory to store constants and optimizing the implementation, possibly excluding the NSEC records, this was not done currently in order to obtain a working prototype of mDNS in Contiki.

8.3 Latency

The idea for testing the latency criterion was to compare the timestamps of the first packet that was sent by the application and the first packet sent by avahi in which avahi-browse queries for all the types of services that it has learned but also includes, in the answer section, PTR records for the service instances that it has discovered. The test was done using Wireshark and looking into the packets that avahi-browse sends to find the right packet that contains PTR record for the service advertised by the mDNS application. The test was carried out using minimal-net and the results are shown in Figure 12, where it can be seen that the first packet sent by the mDNS application that has number 11 has a time stamp 1.574467. The next packet that we are looking for is packet number 16 for which the contents can be seen bellow the packets in the same figure. The main point is that this packet contains PTR record for the service instance that the mDNS is offering and has a timestamp 3.104401, which leads us to the final result that the minimum time needed for the service, that the mDNS application is advertising, to be registered by the avahi tool is about 1.529934 seconds. However this is the result from testing the application using minimal-net, the test for the avr-raven was not carried out

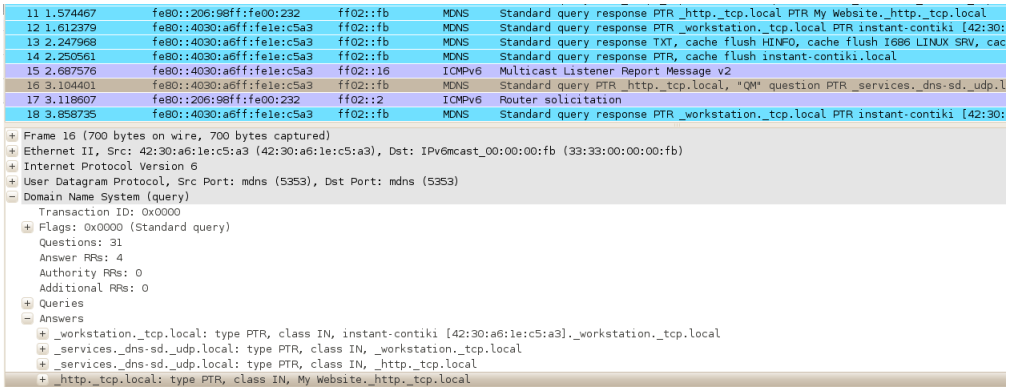

Figure 12: latency test carried out using minimal-net, avahi and Wireshark

since the time in that case depends on many variables such as channel conditions, number of obstacles, distance and battery power. Monitoring simultaneously all of those variables was not possible.

9 Possible usage scenarios of mDNS in embedded networks

There are different usage scenarios of mDNS in embedded networks. Currently there is a need of a list of maintained devices in case we need to know how many of them are available in the immediate neighborhood of a boarder router and mDNS could be used to perform discovery of such devices.

Another usage scenario would be keeping track of services provided by each individual node. While a manual list could be maintained, this method does not scale when the number of devices becomes hundreds or even thousands and as such an automated method to discover which services are offered by devices is important.

Many operating systems (Mac OS X, Windows, Linux) already have Zeroconf implementations which can be used to discover services advertised by network devices running mDNS. This is already widely used to advertise print services, file share, ssh and web-servers. Similarly this could be extended to the embedded devices to discover web-servers, snmp/netconf services and integrate them with discovery on regular operating systems for easy use.

The possibility of self-organizing network, where devices could automatically discover services offered in their neighborhood and possibly collaborate among each other based on those discoveries. An example application of that would be in the smart grid area, where a smart meter could discover all the devices in the home for which it is responsible for, and use the services discovered to change the time some devices turn on and off in order to balance the energy usage and not just read energy consumption from them.

10 Conclusion

The number of electronic devices connected to wireless networks increases each day and we come closer to living up to the concept of "Internet of Things". Having that in mind it becomes crucial to perform automatic device and service discovery which can help keeping track of all devices. Moreover, having such useful mechanism benefits in further development in this area. All related technologies as well as some alternative approaches and current research work, for accomplishing this, were reviewed and described as concise as possible in this paper. Considering most of the currently offered methods for service discovery and taking into account the ubiquitous deployment that DNS already enjoys, it is safe to make a conclusion that the ideal way for device and service discovery in 6LoWPAN is by implementing mDNS with DNS-SD. The focus of this project was implementation of mDNS and DNS-SD in Contiki on the AVR Raven platform. The approach for achieving this goal is described in detail in this paper and the working prototypes of this implementation can be used as a good starting point for the creative programmer.

References

[1] Apple. Technology Brief Mac OS X: Bonjour. http://hes-standards.org/doc/SC25_WG1_N1164.pdf, 2005.

[2] Atmel. RZRAVEN Hardware Users Guide. *Application Note*, 8117D-AVR-04/08, April 2008.

[3] Atmel. 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash ATmega329P/V ATmega3290P/V. *Preliminary Datasheet*, 8021ES-AVR-07/09, July 2009.

[4] Stuart Cheshire and Marc Krocmal. DNS-Based Service Discovery. *draft-cheshire-dnsext-dns-sd-07 (IETF Internet Draft)*, 25 October 2010.

[5] Stuart Cheshire and Marc Krocmal. Multicast DNS. *draft-cheshire-dnsext-multicastdns-12 (IETF Internet Draft)*, 25 October 2010.

[6] Adam Dunkels. The Contiki Operating System - About Contiki. http://www.sics.se/contiki/about-contiki.html, November 2010.

[7] J. Gutierrez, M. Naeve, M. Callaway, M. Bourgeois, V. Mitter, and B. Heile. IEEE 802.15.4: a developing standard for low-power low-cost wireless personal area networks. *IEEE Network*, 15(5), October 2001.

[8] Matus Harvan and Jürgen Schönwälder. A 6LoWPAN Implementation for TinyOS 2.0. In *Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme" (FGSN)*, Aachen, Germany, 2007.

[9] Oliver Kurth. avahi-daemon(8) - linux man page.

[10] N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions,Problem Statement, and Goals. *IETF RFC 4919*, August 2007.

[11] Luca De Nardis and Maria-Gabriella Di Benedetto. Overview of the IEEE 802.15.4/4a standards for low data rate Wireless Personal Data Networks. In *4th Workshop on Positioning, Navigation and Communication (WPNC '07)*, Hannover, Germany, March 2007.

[12] Inc. Network Sorcery. Dns, domain name system. In *RFC Sourcebook*, 1998-2011.

[13] M.Larson D.Massey R. Arends, R.Austein and S. Rose. Resource records for the dns security extensions. *IETF RFC 4034*, page 29, 2005.

[14] Jürgen Schönwälder. Introduction to IEEE 802.15.4 and IPv6 over 802.15.4 (6LoWPAN). In *3rd International Conference on Autonomous Infrastructure,*

Management and Security (AIMS '09), Enschede, Netherlands, July 2009. http://cnds.eecs.jacobs-university.de/presentations/2009-aims-6lowpan.pdf.

[15] Jürgen Schönwälder. Internet of Things: 802.15.4, 6LoWPAN, RPL, CoAP. In *University of Twente*, Enschede, Netherlands, October 2010. http://cnds.eecs.jacobs-university.de/presentations/2010-utwente-6lowpan-rpl-coap.pdf.

[16] Anuj Sehgal. A Practical Introduction to 6LoWPAN Programming IPv6 Wireless Sensor Networks with Contiki. In *4th International Conference on Autonomous Infrastructure, Management and Security (AIMS '10)*, Zurich, Switzerland, June 2010. http://cnds.eecs.jacobs-university.de/presentations/2010-aims-6lowpan.pdf.

[17] Anuj Sehgal. Introduction to Programming with Contiki - Handout. In *4th International Summer School on Network and Service Management (ISSNSM '10)*, Zurich, Switzerland, June 2010.

[18] Ryan Smallegan. Dns based service discovery (dnssd). 2005.

[19] IEEE Computer Society. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). 2006.

[20] The Avahi Team. What is avahi? 2005-2007.

[21] Wikipedia. Internet of things. 2010.

[22] Wikipedia. Smart grid. 2010.

[23] Wikipedia. Wireshark. 2011.

[24] Z.Shelby. CoRE Link Format. *draft-ietf-core-link-format-02 (IETF Internet Draft)*, 10 December 2010.

[25] Z.Shelby, B.Frank, and D. Sturek. Constrained Application Protocol (CoAP). *draft-ietf-core-coap-02 (IETF Ineternet Draft)*, 27 September 2010.