UPPSALA
UNIVERSITET

IT 13 040
Examensarbete 30 hp
Juni 2013

Multichannel Communication in Contiki's Low-power IPv6 Stack

Beshr Al Nahas

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Multichannel Communication in Contiki's Low-power IPv6 Stack

Beshr Al Nahas

Vast majority of wireless appliances used in household, industry and medical field share the ISM frequency band. These devices need to coexist and thus are challenged to tolerate their mutual interference. One way of dealing with this is by using frequency hopping; where the device changes its radio channel periodically. Consequently, communications will not suffer from the same interference each time; instead, it should be fairer and more stable. This thesis investigates the aforementioned problem in the field of low power wireless sensor networks and Internet of Things where Contiki OS is used. We introduce a low-power pseudo-random frequency-hopping MAC protocol which is specifically characterized as a duty cycled asynchronous sender-initiated LPL style protocol. We illustrate two flavors of the protocol; one that does not use any dedicated channel and another which allows dedicated broadcast channels that can implement frequency-hopping as well. We implement the protocol in C for real hardware and extensively test and evaluate it in a simulated environment which runs Contiki. It proved to work with Contiki's IPv6 stack running RPL (the standardized routing protocol for low power and lossy wireless networks). We compare the performance of the implemented protocol to the single-channel ContikiMAC with varying levels of interference. Results show a reduction down to 56% less radio-on time (1.50% vs. 3.4%) and 85% less latency (306 ms vs. 2050 ms) in the presence of noise, while keeping a good basecost in noise-free environments with 1.29% radio duty cycle when using 9 channels with no dedicated broadcast channels (vs. 0.80% for single channel) and 252 ms average latency (vs. 235 ms). Moreover, the results show that the multichannel protocol performance metrics converge to almost the same values regardless of the noise level. Therefore, it is recommended as a good alternative to single channel ContikiMAC in real world deployments where noise presence is anticipated.

Handledare: Simon Duquennoy
Ämnesgranskare: Thiemo Voigt
Examinator: Philipp Rümmer
IT 13 040
Sponsor: SICS Swedish ICT AB

Tryckt av: Reprocentralen ITC

Acknowledgements

It is with immense gratitude that I acknowledge the help, support and patience of my supervisor *Simon Duquennoy*. His inspirational discussions, professional critiques that come with a sense of humour and great motivations in the dark Scandinavian winter were invaluable to me personally and to the completion of this thesis. It gives me great pleasure as well in acknowledging the guidance and support of my thesis reviewer Professor *Thiemo Voigt*, who offered me the opportunity to do this work in the Networked Embedded Systems group at SICS in the first place, and provided essential support and feedback throughout the work. I wish to thank my Master program coordinator and thesis examiner *Philipp Rmmer*, who provided great help and academic guidance through my Master study. I would like to thank all the great guys in the Networked Embedded Systems group at SICS; especially, *Joakim Eriksson* for his inspirations and help, and *Niclas Finne* for providing invaluable support with various bugs in the toolchain. Last, but by no means least, I must thank my friends *Vilhelm Jutvik* and *Martin Russo* for the interesting discussions, encouragement and nice company.

Dedicated to my beloved parents,

Randa Tabba and Bakri Al Nahas,

who always provided unlimited support, love, faith and wisdom.

I owe you my deepest gratitude.

Contents

List of Figures

List of Tables

1 Introduction

After the communication and Internet revolution, the focus of new technologies in these areas has been drawn to connecting more devices, even everyday objects, to the Internet. This thrust has been driven by the humankind need to discover, understand and possibly control the surroundings (both the natural and the man made environments). This, in turn, fostered the realization of the new/old human dream of personification and making 'things' or 'objects' smart such that they can 'feel' and interact with each other and with the surrounding in order to simplify and enhance our quality of life and comfort. These visions can be realized and categorized under two main application areas

- *Wireless Sensor Networks (WSN)*, which focus on sensing the surrounding and providing measurements to *i.e.* a control center;
- and *Internet of Things (IoT)*, which focuses on making everyday objects smart and giving them the ability to connect to the Internet and interact with the surrounding.

This field has advanced rapidly in the last decade; consequently, these two areas are conveniently supported by operating systems and software stacks such as *Contiki* operating system which is Europe's leading open source operating system for sensor networks. *Contiki* is based on uIP, the world's first IPv6 stack for sensors [4]. Moreover, these two application areas had to solve the problem of wired installations which limited the possible use cases and installation environments due to the unavailability of wired connections or to the prohibitively high costs associated with them. On the other hand, the freedom coming from cutting the wires requires, at least, two additional features to be implemented in those smart Internet-connected devices; namely, a portable power source (*i.e.* a battery or an energy harvester) and wireless communication. Nevertheless, these features come with limitations too. Using batteries maybe impractical if we have to replace them often. In other words, the technologies enabling 'smart' objects should consume low power such that we do not need to bother changing the batteries for a long time, preferably, in the order of years. Therefore, the low-power wireless radio standard IEEE 802.15.4 [5] was developed and implemented to target the aforementioned application areas. However, low-power wireless communications are challenging in a number of ways

- the wireless links are unreliable due to noise coming from the environment, electrical machines, interference and cross-talk from other devices using similar wireless technology [6];

- the wireless spectrum is shared between a lot of devices that employ different technologies; thus, coexisting technologies interfere each other [1, 7, 8]. Figure 1.1 from [1] shows the effect of interference from a nearby IEEE 802.11 (WiFi) network on packet delivery ratio (PDR) in IEEE 802.15.4 network and the variation of PDR according to the used frequency channel;
- the nature of wireless propagation and multipath fading causes challenging link dynamics that affect the signal strength and packet reception rate in relation to a number of parameters; namely, the used frequency, the shape of the wireless path, the objects standing/moving in the path and the location of the transceiver [2]. Figure 1.2 from [2] shows the variations of packet delivery ratio when the transmitter's location is changed;
- and finally, the radio component is usually the most energy hungry part in a smart-object platform; thus saving power in wireless communications is crucial to low-power operation.

Our work, which is presented in this Master thesis, tackles the challenges of interference and coexistence in low-power wireless communications by developing a frequency-hopping MAC protocol, which, in the same time, keeps the radio off for most of the time in order to suit the battery-powered applications; especially, in the areas of Wireless Sensor Networks and Internet of Things. This thesis takes place in the context of Contiki's IPv6 communication stack.

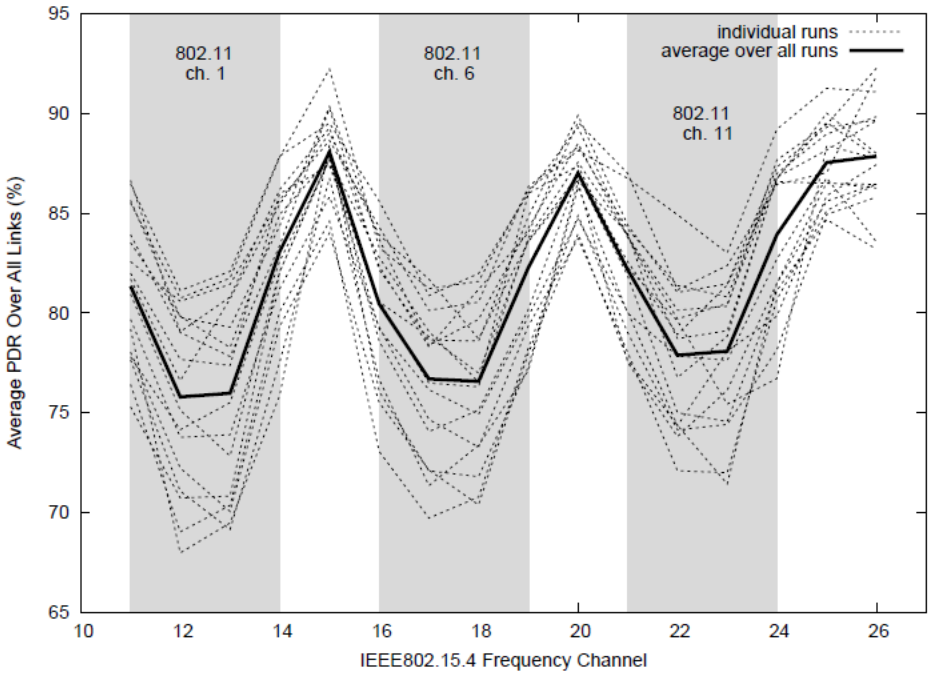


FIGURE 1.1: WiFi interference impacts IEEE 802.15.4 PDR [1]

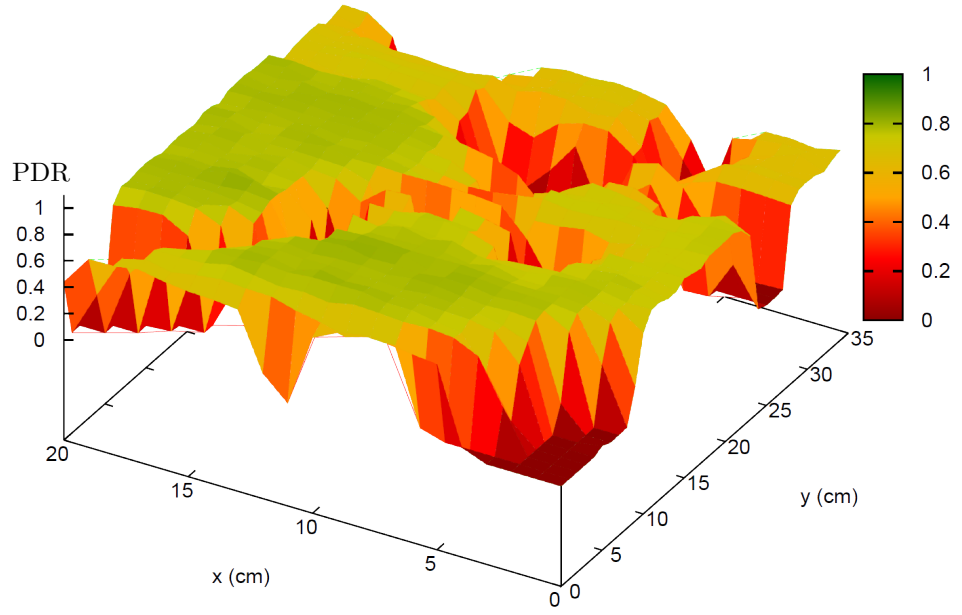


FIGURE 1.2: Multipath fading experienced when changing the location of the transmitter affects the PDR in IEEE 802.15.4 [2]

1.1 Problem Statement

The problem we approach in this thesis can be summarized in the following question:

> *How can we achieve frequency diversity in radio communications while keeping the solution ultra-low power, self-configurable, dynamic, distributed and, in the same time, integrated with the low-power IPv6 stack?*

In other words, we want to develop a low-power multichannel MAC protocol that suits dynamic connectivity in the IP/6LoWPAN stack [9] with the new standardized IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [10]. Furthermore, we aim to keep the frequency diversity overhead as low as possible compared to single channel solutions, while enjoying the advantages of frequency diversity, such as resilience to interference [11], and spatial reuse, which in turn decreases congestion and improves performance [12, 13]. The target application areas are dynamic Wireless Sensor Networks (WSN) and Internet of Things (IoT).

1.2 Methodology

The methodology we use is that of *experimental computer science*. We begin by illustrating the main existing solutions in the area of the multichannel MAC protocols for WSNs and IoT, then, we compare, contrast and reason about the characteristics of

existing protocols in the light of target properties. Later, we implement the missing characteristics that appear to solve the problem in a better way. Finally, we evaluate the chosen solution in terms of some quantifiable performance metrics.

1.3 Alternative Approaches

The topic of low-power multichannel MAC protocols has been studied in the recent years in several publications, and different protocols have been developed. The existing protocols base their designs on one or more of the following characteristics

- Synchronous; thus, requiring global synchronization which might be problematic using low-power low-bitrate wireless radios and comes with problems supporting self-configuration and distributed operation, *e.g.* Y-MAC [12], MC-LMAC [14], MuChMAC [13];
- Utilizing a common channel, which might become a bottle neck especially if interfered, *e.g.* Y-MAC [12];
- Dedicated a special kind of applications, which obviously makes it unsuitable for the Internet of Things, *e.g.* Chrysso [15];
- or Receiver-initiated; thus having a relatively higher base cost, *e.g.* EM-MAC [11].

Moreover, most of the existing protocols were not tested in the IP/6LoWPAN stack with RPL, which is essential for future deployments.

1.4 Our Approach

Many of the existing work in low-power MAC protocols utilize duty-cycling, which turns the radio off such that the idle listening time is minimized. Obviously, the protocols we consider are those utilizing multichannel communications. However, none of the existing protocols combined the following characteristics in their design; asynchronous, sender-initiated, duty cycled, frequency-hopping, general-purpose and supporting both unicast and broadcast operations. We find this approach interesting since it combines features that we suggest to fit well the target area of dynamic, low-power, wireless networks. Specifically speaking:

- Frequency diversity implemented by frequency-hopping is suggested to mitigate the effects of multipath fading [2], to improve reliability of the wireless network [1] and to improve resilience to interference [11];
- Asynchronous suits dynamic networks well as no global synchronization is needed;

- and Sender-initiated minimizes the base cost; thus, it suits generic IP communications that does not have a steady traffic pattern.

Furthermore, we base this design on ContikiMAC since it proved to be ultra-low power and robust in the case of single channel as shown in [16]. However, the challenges lie in bringing frequency diversity advantages without considerably increasing the base cost of the protocol in cases where a single channel MAC could perform well.

1.5 Our Contribution

We design, implement and evaluate an asynchronous, sender-initiated, duty cycled, multichannel MAC protocol that suits Wireless Sensor Networks and Internet of Things. Our implementation is in C for Contiki OS and is done on real sensor motes which are constrained in terms of CPU power, RAM and code size. Moreover, we integrate and test our protocol in the IP/6LoWPAN stack with RPL.

Evaluation of the protocol shows an improved performance in terms of duty cycle and packet delivery latency in presence of noise, while keeping a relatively low base cost compared to single channel ContikiMAC when no noise is present. We suggest this protocol as a viable alternative for real deployments where wireless interference is expected.

1.6 Thesis Outline

The rest of this thesis is organized in the following way: Chapter 2 provides necessary background information that first explains in a bit of details the enabling technologies and the IP/6LoWPAN protocol stack, and then illustrates the state of the art in the field; thus laying down the context and background for the design we choose. Chapter 3 motivates the design choices we make, illustrates the design in details and then explains the implementation related aspects of the protocol such that the interested reader understands the protocol thoroughly in case further development is desired. Chapter 4 illustrates the experimental setup for the evaluation, explains the details and parameters of the performed experiments, and evaluates the protocol in terms of selected performance metrics; *i.e.* duty cycle and packet delivery latency. Chapter 5 concludes the thesis and suggests possible future work.

2 Background

This chapter introduces background information to familiarize the reader with some terms and topics used in the thesis. We begin with defining the intended application areas, then we present the set of technologies used in the context of the thesis, and finally we conclude with a discussion of existing work that represents the state of the art in the field.

2.1 Embedded Systems and Smart Objects

Embedded systems are computer systems embedded in other systems. Thus, they are equipped with CPU and memory, but they do not look like PCs (or notebooks) and are designed to perform specific tasks in contrast to a general purpose computer system [17]. Examples are everywhere: Microwave ovens, washing machines, modern TVs and automotive among others.

Smart objects are one kind of embedded systems with the important addition of sensors and or actuators, communication means and a power source (i.e. battery). Therefore, smart objects are able to perform logical actions, interact with the environment (by sensing or actuating), communicate with other devices and be stand-alone. Smart objects tend to have a small form factor so that they can be embedded in everyday objects. This usually means a limited power supply (i.e. battery) that is expected to last for years. Due to both size and energy constraints, CPU performance and available memory is usually rather limited. Thus, software developed for them should consider that [18]. **Sensor nodes** are one kind of smart objects. They are mainly characterized by their intended use. Sensor nodes are usually intended to operate in industrial, medical, environmental or scientific applications, while smart objects refer to the concept of making everyday objects smart.

2.2 Connecting Smart Objects

When smart objects are connected, they open the door for a wide area of possible applications. We present two such application areas.

Wireless Sensor Networks (WSNs) A WSN is a network of sensor nodes with the purpose of collecting sensor measurements from the target environment, and sending these measurements over the radio. One classical example is environmental monitoring, where sensor nodes are distributed over the area of interest measuring some properties there; such as temperature for example.

Internet of Things (IoT) IoT refers to the concept of connecting everyday objects to the Internet, making possible a whole new set of applications and ideas where, for example, the plant standing next to your window can tweet how comfortable it feels (e.g. by sensing temperature and humidity), and perhaps tell the window curtains to close partially when it is too hot. This might have seemed unrealistic or prohibitively expensive and difficult to implement a decade ago. However, with the availability of cheap hardware for smart objects, and with the help of free and open source software developed specifically to support IoT, it is becoming affordable.

Connecting smart objects together and to the Internet need specialized software with networking functionality and IP support. We present in the following section one possible solution that is enabled by widely used open-source software.

2.3 Enabling Technologies

One way for implementing IoT is by using Contiki operating system with its networking stack that includes, IPv6 (network connectivity) with 6LoWPAN, RPL (routing), ContikiMAC and IEEE 802.15.4 (wireless radio). Each of the components is explained respectively in subsequent subsections.

2.3.1 IEEE 802.15.4

This is a standardized radio protocol for low power, low data rate, low cost, adhoc, self organizing network for home networking applications [3]. It is used for the radio component in many implementations of IoT and WSN. The standard was published in 2003. It was revised in 2006 and later in 2011 [5]. The standard defines the medium access control (MAC) layer and multiple variations of physical (PHY) layer.

PHY layer This layer is responsible for the actual data transmission and reception on the radio medium, frequency selection for the chosen channel (according to specifications), turning on and off the radio transceiver, estimating link quality indicator (LQI) for received frames and performing clear channel assessment (CCA) to detect busy channel condition which supports carrier sense multiple access with collision avoidance (CSMA-CA).

Band (MHz)	*Bit rate* (Kbps)	*Modulation*	*Channels*
779-787	250	O-QPSK, MPSK	0-7
868-868.6/902-928	20/40, 100/250, 250	BPSK, ASK, O-QPSK	0/1-10
950-956	20, 100	GFSK, BPSK	0-21
2400-2483.5 DSSS	250	O-QPSK	11-26
2400-2483.5 CSS	250, 1000	DQPSK-DQCSK	0-13
250-750 UWB	110, 850, 6.8M, 27M	BPM-BPSK	0
3244-4742 UWB	=	=	1-4
5944-10234 UWB	=	=	5-15
314-316, 430-434	defined by Chinese standards CWPAN.		

TABLE 2.1: PHY layer variations of IEEE 802.15.4-2011

The standard defines multiple frequency bands and data rates. They are summarized in table 2.1. The standard defines the general structure of the physical packet data unit (PPDU) to contain three main fields: Synchronization Header (SHR), Physical Header (PHR) and PHY payload. It should be noted that the maximum size for the payload is 127 bytes.

MAC layer This layer is responsible for regulating the access to the radio medium and providing a reliable link between two communicating nodes. It specifies the frame format as shown in figure 2.1. It supports several mechanisms for more reliable data transmission, such as acknowledged packet delivery, CSMA-CA, ALOHA, data checking, power consumption consideration and optional security sublayer.

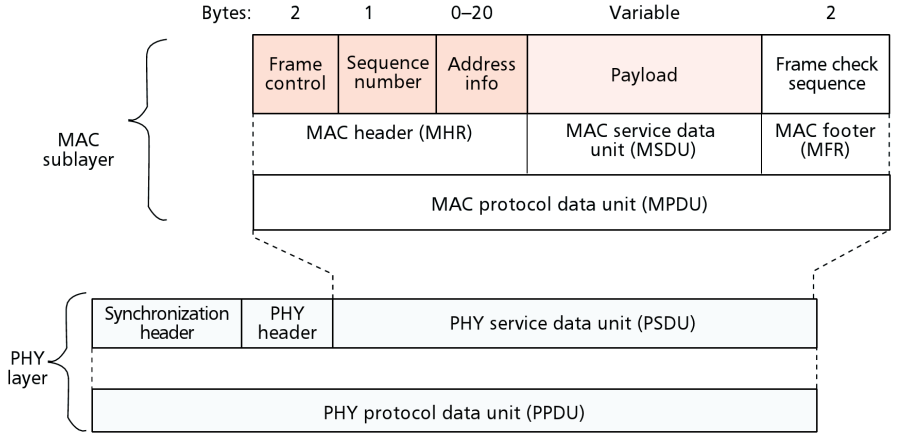


FIGURE 2.1: IEEE 802.15.4 Frame [3]

Acknowledged packet delivery is optional for unicast data packets, where the receiver is required to send an acknowledge packet to confirm successful delivery.

CSMA-CA and ALOHA CSMA-CA mechanism helps reducing collisions in sent packets. If activated, it checks the channel status (CCA) before trying to send. If the channel was clear, it sends the data. Otherwise, it performs back-off (waits for a random time) and repeats the process again.

Another option is to use ALOHA; where channel check is disabled, and packets are sent immediately. Later on, if the packet was not acknowledged, the sender performs back-off and tries sending again.

Data checking The MAC layer computes error detection code over the payload and header. It employs cyclic redundancy check (CRC-16) codes. The code is put in the FCS field of the sent packet. The receiver computes CRC for each received packet and compares the result with the CRC received. If they do not match, the packet is decided to be corrupt, and subsequently, will be dropped.

Power consumption consideration Majority of devices that use this standard will be battery powered and thus require more power-saving methods. Keeping in mind that the radio is usually one of the power hungriest components in a smart object, turning it off will extend the battery life considerably. *Duty cycling* is widely used for this purpose: The device turns its radio off for most of the time instead of actively listening to the radio medium. It wakes up only when it needs to send or receive.

Security sublayer The standard defines optional AES encryption, data integrity and authenticity support.

2.3.2 ContikiMAC

It is a very low power, asynchronous, duty cycled, sender initiated MAC protocol that enables sensor nodes to communicate efficiently while keeping their radio on for less than 1% of the time [16]. It is designed to be used primarily over IEEE 802.15.4. However, nothing prevents it from being used with other PHY layers, provided that its parameters are configured accordingly. In this section, we present ContikiMAC in details; as it represents the basis of the design of the new multichannel MAC protocol.

Overview Figures 2.2 and 2.3 depict the main events for unicast and broadcast in ContikiMAC. A ContikiMAC *receiver* R keeps its radio off for most of the time, and checks the radio medium for incoming packets periodically. This is referred to as clear channel assessment (CCA). When R senses radio activity, it keeps the radio on while receiving the packet. Upon reception of a valid packet, it sends an acknowledgement packet (ACK) back to the sender to signal successful communication.

When a node S wants to send a *unicast* packet to R, it checks the radio medium for any nearby radio activity; trying to avoid collisions, then it tries to send. Since it does not know whether R is awake or not, it tries to 'wake' R up by repeatedly sending the intended packet until it gets ACK from R, then goes back to sleep. S stores the relative time it got the ACK from R as it indicates R's wake up time in the period. When S tries to send to R next time, it schedules the send trials to happen a bit before R's expected wake up time, in order to minimize sending cost. We call this *phase-lock*.

When a node S wants to send a *broadcast* packet to all neighbors, it checks the radio medium for any nearby radio activity; then it sends the packet repeatedly for a full wake up period. This maximizes the possibility of broadcast reception. However, it does not wait for acknowledgements of reception.

Since the sender takes responsibility of waking up the receiver, ContikiMAC is said to be *sender initiated*. Since sending, receiving and the wake up cycles are not synchronized to a global clock, it is said to be *asynchronous*. However, the wake up cycle is independent from sending and occurs periodically when no sending operation is taking place. It should be noted that all radio operations in ContikiMAC takes place on a *single radio channel*, which is usually configured at compile time to be the same for the whole network.

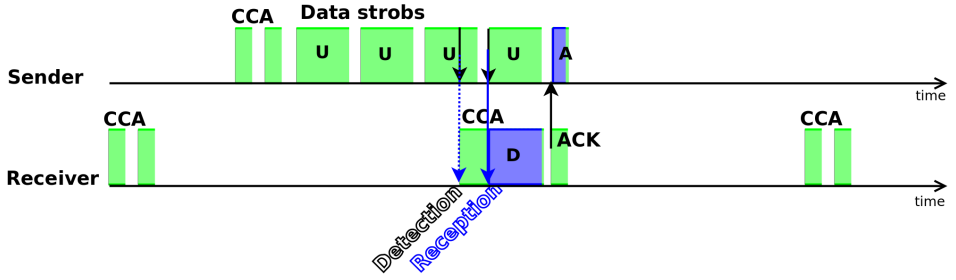


FIGURE 2.2: ContikiMAC unicast

Protocol timing ContikiMAC defines certain timing requirements for ensuring correct behavior with duty cycling, i.e. not missing incoming packets. In the same time it tries to minimize wake-up periods that are triggered by spurious noise. To save power on wake up, ContikiMAC uses a cheap physical layer function that does clear channel assessment (CCA). It is performed by turning on the radio transceiver for a very short period of time to measure received signal strength indicator (RSSI), which indicates radio activity if above a specific threshold.

On wake up, ContikiMAC performs two consecutive CCAs. As illustrated by figure 2.4a, each last for t_r and they are separated by radio off period equal to t_c. If one CCA is negative (i.e. channel is not clear and radio activity is detected), the radio is kept on for a longer period to receive a potential packet. When a packet is received correctly (as indicated by CRC check), an ACK is sent after t_a if requested in the packet header (specifically, in Frame Control field). Unicasts performed by ContikiMAC require ACKs, while broadcasts do not. To ensure that a packet is not missed, it should be long enough

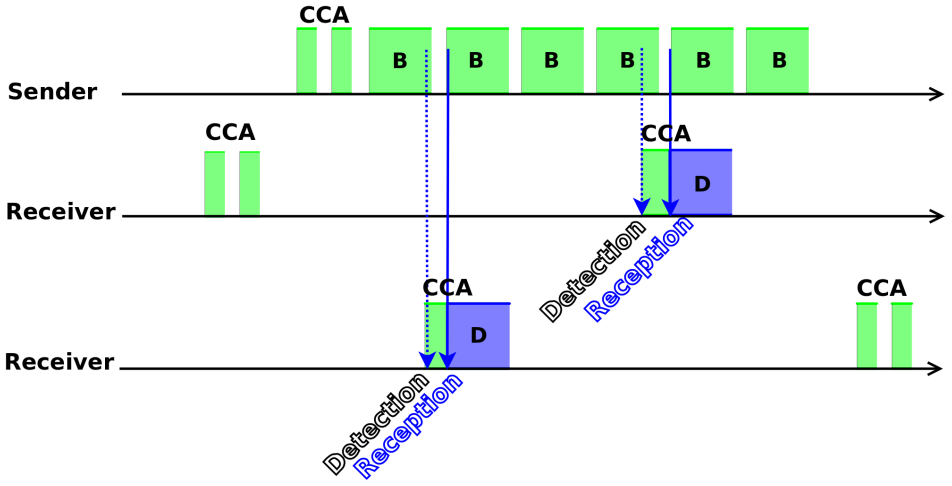


FIGURE 2.3: ContikiMAC broadcast

not to fall between the two CCAs, as illustrated in figure 2.4b. Therefore, the minimum packet duration t_s should be bigger than $2t_r + t_c$. A sender should repeatedly send the intended unicast packet in order to ensure reception by the receiver which might be asleep in that time. The sender expects ACK for the unicast; thus it should wait for time t_i between successive send trials. t_i should be enough for receiving ACK t_a and detecting it t_d. These constraints can be summarized by the following inequality, which are extracted from [16]

$$t_a + t_d < t_i < t_c < t_c + 2t_r < t_s \tag{2.1}$$

After replacing IEEE 802.15.4 specific values in equation 2.1, we get

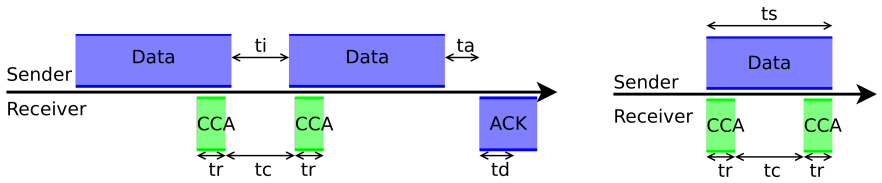
$$0.192 + 0.16 < t_i < t_c < t_c + 2(0.192) < t_s(milliseconds) \tag{2.2}$$

$$\equiv 0.352 < t_i < t_c < t_c + 0.384 < t_s(ms) \tag{2.3}$$

From this we can directly get a lower bound for $t_s > 0.736$ (ms). Given that IEEE 802.15.4 bit rate is 250Kbps, the minimum packet length is 23 bytes. Subtracting the length of preamble and PHY header gives 16 bytes.

Contiki 2.6 chooses the following values

- $t_i = 0.4$,
- $t_c = 0.5$ and
- $t_s = 0.884$ milliseconds.



(A) Timing (B) Minimum packet length

FIGURE 2.4: ContikiMAC parameters

Fast sleep optimization Since CCA mechanism does not differentiate between interesting radio activity and noise, the receiver needs to stay awake for a period long enough to receive a potential long packet t_l, and since a receiver might wake up in the middle of a valid transmission trial, it might need to stay awake for a time long enough to cover the reception of the first incomplete trial, the waiting time t_i and the next trial. This means in the worst cast, a receiver might need to stay awake for $2t_l + t_i$ after a negative CCA. This might be bad in the presence of noise (which is usually the case), because it wastes power listening to the medium.

Given that ContikiMAC enforces the timing constraints shown in equation 2.1, a receiver can detect a radio activity as noise if it does not respect them. More specifically, if radio activity lasts longer than t_l without waiting for t_i then it should be noise. On the other hand, if it lasts for less than t_s, it is not interesting as well. Moreover, if the packet-start delimiter is not detected after silence period of t_i, then the radio activity can be safely ignored. In the three cases, the receiver can go back to sleep earlier.

Phase-lock optimization A sender S can remember the relative time –in a the wake up cycle– of receiving an ACK from receiver R. This way, S does not need to strobe R for the whole wakeup period to send a packet. Instead, it can schedule a packet to be transmitted shortly before R's expected wake up time; thus reducing sending power cost. Figure 2.5 illustrates this.

To implement phase-lock, each sender constructs a neighbor table that contains neighbors' respective wakeup times. To maintain the table, each time a sender sends a packet successfully (as indicated by the reception of a valid ACK), the respective receiver wake up time is updated. Senders store failed transmissions count for each receiver as well. When this count surpasses a predefined threshold (16), it discards the respective neighbor from the table as an indication for the need of reestablishment of phase-lock. As another counter measure for clock drifts, a neighbor entry is discarded form the table as well, if no ACK is received within a specific time (30 seconds).

CSMA and packet queues This sublayer operates over ContikiMAC to optimize its performance and achieve a high throughput [19] by reducing collisions and enabling packets destined to the same receiver to be sent in one burst.

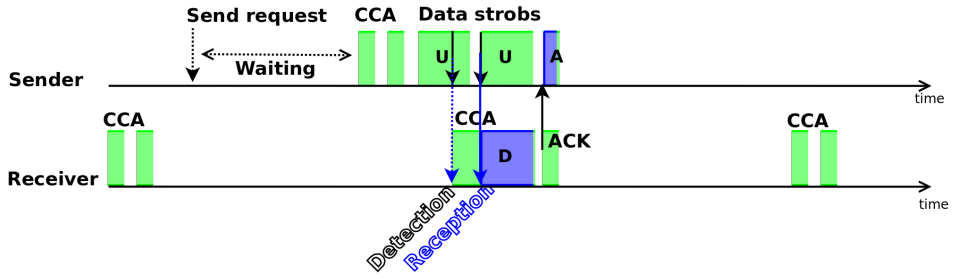


FIGURE 2.5: ContikiMAC phase-lock optimization

To reduce collisions, a sender performs *carrier sense* where it checks the radio for any ongoing activity before sending. This is achieved using at least two consecutive CCAs similar to those used in the wakeup cycle. If CCA check was negative, the sender performs exponential back-off, i.e. uses an exponentially increasing waiting time with each failed send trial and adds some randomization. Similarly, if the sender does not receive ACK, it assumes a collision and performs exponential back-off.

Another optimization is caching packets destined to the same receiver R in a special packet queue. The sender sends queued packet consecutively to R after receiving R's ACK for the first packet. It keeps sending packets as long as it has more queued packets and as long as it is receiving ACKs from R. The sender employs a special flag in the frame header (Frame Pending flag) to tell the receiver to stay awake and expect another incoming packet instead of going back to sleep after receiving the first one. The sender turns this flag off when sending the last packet in the queue. When R sees this flag enabled in a received packet, it keeps its transceiver on for $t_b = 31.25(ms)$ after sending the ACK, waiting for a packet. If it does not receive anything, it returns back to sleep. Otherwise, if it receives a new packet, it checks the 'Frame Pending' flag to decide whether to go back to sleep or not. This feature exploits the bursty nature of wireless links and time-correlated losses, where you expect the medium to stay clear shortly after a successful transmission. Thus, it reduces latency by sending a burst of packets when R is awake and link quality is good.

Related MACs ContikiMAC design was inspired by several MAC protocols including; X-MAC [20] which is sender-initiated and uses short preambles to wake up the receiver; BoX-MAC [21] which implemented the idea of using packets for wake-up instead of preambles, and WiseMAC [22] which introduced phase-lock optimization.

2.3.3 IPv6

The next generation protocol to be used on the Internet [23]; superseding IPv4 which was released in 1981 and is still in use. However, major Internet providers usually say that IPv4 will be replaced completely by IPv6 in a couple of years due to exhaustion of address-space, which has already started biting. For example, the *Réseaux IP Européens*

Network Coordination Centre (RIPE NCC) announced on 14th September 2012 reaching the last /8 address space; meaning that only 18 million IPv4 addresses are left [24].

As articulated already, the main feature of IPv6 is the practically infinitely **large address space**, which is enabled by the use of 128bits for addressing (in comparison to 32bit in IPv4). This is especially beneficial for IoT where each smart object is able to get its own unique Internet address. Another important feature is **auto configuration**, which means that you the network will work without the need to configure each single node in the network and supply it with i.e. an address. This feature is essential to IoT because it will allow seamless operation where smart objects could be connected to the network by just switching them on.

Overall, IPv6 architecture tries to standardize and integrate almost all components needed to operate the network and transport layers, putting together security support (IPSEC), auto configuration, neighbor discovery, supporting multicast, simplifying optional headers support (by putting a field pointing to next header) and moving back to the original design of IP where each device is able to have a unique address. It is beneficial for IoT to use IPv6 because of the features listed above and because of its native support of Internet connectivity and interoperability. However, given that IPv6 was designed for full fledged devices, one might argue that it is restrictively big and complicated for smart objects. This has been solved with the implementation of uIPv6 which is the smallest certified IPv6 Ready stack available for constrained devices [4]. Moreover, some of its features might be problematic to support for smart objects that have limited CPU, memory and connectivity options. IPv6 removes fragmentation, which makes it mandatory for devices to support a maximum transmission unit (MTU) of 1280 bytes, which is not the case for most smart objects. IPv6 uses 128bit addresses and supports a lot of features, which make its header size considerably large and thus costly to transmit for battery powered devices. These aspects justify the need for an adaptation layer over IPv6 for supporting smart objects.

2.3.4 6LoWPAN

The IETF standard for enabling IPv6 over low power wireless premise area network; specifically IEEE 802.15.4 networks [9]. As articulated earlier, IPv6 is crucial for interoperability and Internet connectivity in WSN and IoT, but it brings some demands and challenges for smart objects. 6LoWPAN is an adaptation layer working on top of IEEE 802.15.4 MAC to provide the following services to support IPv6 seamless operation [18]:

- Fragmentation and defragmentation of IPv6 packets (MTU $\leq$ 1280 bytes) so they can be carried over multiple IEEE 802.15.4 packets (MTU $\leq$ 127 bytes) of payload.
- Compression and decompression of IPv6 header to lessen the transmission overhead. It can compress the 40 bytes IPv6 header to as little as 4 bytes [25].
- Forwarding packets over multi-hop mesh networks.

The attractiveness of 6LoWPAN lies in enabling IPv6 with a small overhead [25] in terms of: Code size (12-22K), RAM requirements (4K) and header size (2-11 bytes).

2.3.5 RPL

This is the standard IPv6 routing protocol for low-power and lossy networks (LLNs) [10], or simply; it is the IP routing protocol for smart object networks [18]. It has been built specifically to support the requirements of LLNs which exhibit special characteristics such as: Limited energy, limited processing capabilities and highly dynamic topologies (because of link instability and node failures). RPL design was informed by Collection Tree Protocol (CTP) [26, 27]

RPL builds directed acyclic graph (DAG) representation of the network, with the possibility of having multiple DAGs for the same network but for different routing criteria such as estimated transmission count (ETX), latency, hop count, node power, etc. A DAG is a tree-like structure with a single root node that has no parents and usually represents a border router. The main difference from the tree is the possibility of having multiple parents for each node. Thus, DAGs support redundancy naturally.

RPL supports three modes of traffic [10]:

- Point-to-multipoint (*i.e.* multicast) such as downlink traffic from root to children.
- Multipoint-to-point (*i.e.* converge-cast) such as uplink traffic from children to root.
- Point-to-point (*i.e.* unicast).

How RPL builds a DAG The root broadcasts a DAG information object (DIO) message containing information about the proposed DAG characteristics, routing object function and path cost (rank). The advertised rank represents the node position in the DAG hierarchy, and in the same time, corresponds to the configured routing metric e.g. ETX. The root rank is zero (by default), while the children has another default initial value (e.g. for ETX it is 5). Nodes in the listening range receive the DIO and decide whether to join the DAG or not according to advertised routing configuration. If a node receives multiple DIO messages attributed to the same DAG, it compares them according to the objective function (e.g. the least ETX), chooses the best node as the preferred parent, updates its relative rank by adding the received value to it, and adds the other nodes to the backup parents set. If the node was a leaf node, it just joins the DAG. Otherwise, if the node was configured as a router, it advertises the DAG in turn by broadcasting a DIO with its updated rank. This way, the DAG grows until it covers all the leafs reachable by the routers; enabling packets delivery from nodes up to the root [28].

RPL example Figure 2.6 is a step by step example for building a new DAG.
Step 1:

- The root node A (the border router) starts by advertising a new DAG which uses ETX for the routing metric.
- Nodes B and C receive this advertisement and decide to join the DAG.

Step 2:

- Both B and C set A as their parent and update their ranks.
- After a while, both B and C update their ETX values according to link conditions. Node B figures out that it needs more transmission trials than it anticipated in the beginning, while C figures out better link conditions. Both of them reflect the changes in their respective ranks.

Step 3:

- New nodes (D, E, F, G) join the network.
- Both B and C advertise their routing information (DIO) according to their respective internal timers.
- Node D receives B's DIO, nodes F, G receive C's DIO, while node E receives both DIOs from both B and C.

Step 4:

- D, E, F, G decide to join the network.
- D, F, G each has only one option for the parent.
- E has to choose between B and C. It chooses C as the preferred parent because it has a lower rank, while it keeps B in the parent set as a backup.

Self healing, local repair and global repair RPL is able to detect loops, and dynamically restore network connectivity after node or link failures. If a node can't reach neither its preferred parent nor any backup (in the up direction), it initiates local repair to find another parent to connect to. Local repair is simply done by broadcasting a DAG information solicitation (DIS) message. Neighboring nodes reply to this by sending DIO messages back, enabling the requester to choose the best available parent to connect to. This might result in sub-optimal path for this part of the DAG, but it does not require a network-wide routing update. However, the root node can trigger a

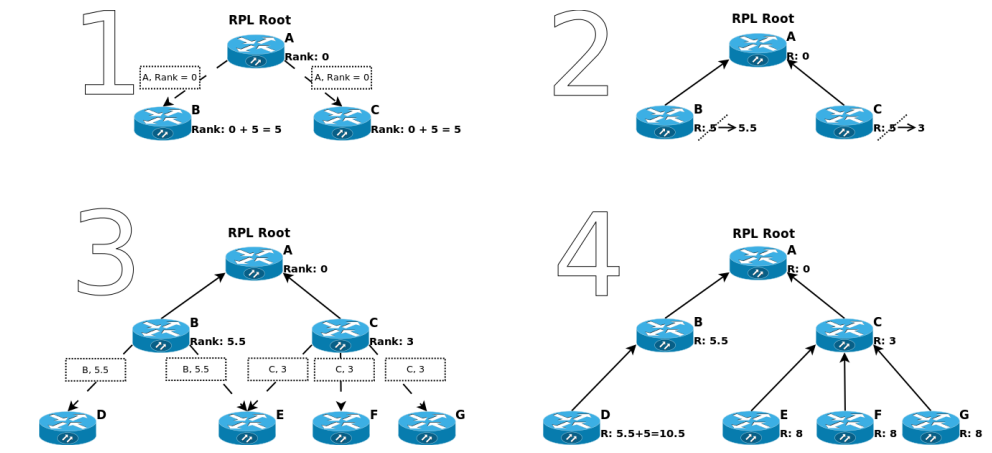


FIGURE 2.6: RPL example: Building a DAG

global repair, which rebuilds the whole DAG from scratch; giving a more optimal DAG at the cost of increased routing information traffic.

RPL employs a data-path validation mechanism to facilitate detection of possible loops. It adds to data packets a routing header with special flags that i.e. signal direction. Each router, in the data packet path, processes these flags and sets the respective direction flags i.e. up or down. When a router detects a loop while processing these flags, it discards the data packet and initiates local repair.

Adaptive beaconing For maintaining the network and adapting to changes, router nodes should send DIO messages periodically. In low-power networks, saving power is crucial. Therefore, RPL tries to reduce control messages as much as possible while still maintaining network dynamically. This is achieved using a trickle-timer [29] to schedule the advertisements of DIO messages containing updated routing information. A trickle timer increases the waiting period exponentially (up to a preset threshold), resulting in reduction of routing messages when the network is stable. On the other hand, when a network inconsistency or a parameter change is detected, the timer is reset to its smallest value; enabling routing updates to be sent more frequently, so the network can adapt to changes quickly.

2.3.6 Contiki

Contiki [30] is Europe's leading operating system for sensor networks. It is open source, realtime and built with certified IPv6 stack [4]. Contiki's IP stack implements all of the aforementioned enabling technology. That is it, Contiki has full implementation of IPv6 with TCP, UDP, RPL and ICMP. It supports IPv6 over IEEE802.15.4 out of the box by providing 6LoWPAN adaptation layer, a variety of duty cycled MAC layers and radio drivers for a variety of sensor motes and smart objects hardware platforms.

Contiki incorporates energy saving features, a light-weight multi-threading library (protothreads), a straight forward programming style corresponding to C standards, a rich API library and timers. It features an energy profiling library [31] that measures the time spent running various components of the sensor nodes. This eases the development of low power applications by giving quite accurate insights about where energy is spent the most during the run of the application. Contiki provides all the mentioned features and more [32], while keeping a small operating power profile and quite a small memory footprint in terms of both flash and RAM usage.

2.4 State of the Art

Multichannel communication has potential benefits for wireless networks that possibly include; improved resilience against external interference, reduced latency, enhanced reception rate and increased throughput. In this section, we review a selected set of existing peer-reviewed low power multichannel MAC protocols. Then, we compare them objectively, trying to highlight their features and limitations. The reviewed protocols represent –up to our knowledge– the state of the art in the field.

2.4.1 Y-MAC

Y-MAC [12] uses time division multiple access (TDMA), where time is divided to synchronized superframes. Each superframe is divided to two periods: Broadcast period and unicast period. Each period is divided to slots which are uniquely assigned to nodes in the network. Each slot consists of a contention window and a data window. Channel hopping schema is simple. Communication always starts in a base channel, and on subsequent packets both sender and receiver hop to a different channel.

Sender/receiver rendezvous Receivers tune to the base channel in the start of their time-slot, and listen in the beginning of data window. If no communication is detected, the receiver switch off its transceiver for the rest of the time.

Senders contend for a receiver in the contention window in the receiver's time-slot. The winning sender sends its first message in the base channel, then both receiver and sender hop to a new channel. Other senders that failed the contention follow the receiver in frequency hopping, and contend again. Once there are no more packets to send, the nodes go back to low power listening in base frequency.

Broadcast happens in a similar way, but is limited to the broadcast period, and always takes palce in the base channel. All nodes tune to base channel in the beginning of each broadcast slot. Senders contend in the contention window in each slot, and the winner can broadcast.

Time synchronization It is supported by control messages broadcasted periodically, which contain the remaining time till the end of superframe. At first, the sink node starts broadcasting timing information. A node trying to join, sets its time to the received time, and starts broadcasting timing information as well. Later on, each node averages its time with the received timing information to compensate for clock drifts.

2.4.2 MuChMAC

MuChMAC [13] combines synchronous techniques with relatively long superframes and asynchronous techniques inside the long time-slots. Each node switches its frequency channel every time-slot. Each time-slot is divided to 16 sub-slots. Each node picks a sub-slot according to node's ID and time-slot number. Nodes pickup a wakeup time in their chosen sub-slot, and sleep in the remaining time of the time-slot. Communication happens in a contention approach inside time-slots.

Sender/receiver rendezvous Channel selection is done by the receiver R based on *parallel rendezvous* principle. R utilizes a *pseudo-random function* which takes in node ID and slot number as parameters. Dedicated *broadcast* slots are inserted every n (i.e. 2) unicast slots. Broadcast slots follow the same pseudo-random frequency sequence as well, but they are *shared* among all nodes. Channel selection is *passive* and does not adapt to noise conditions.

The sender S should have *loose synchronization* with the receiver up to slot bounds, as it determines the communication channel. S uses the same pseudo-random function to figure out the receiver channel in the next time-slot. In the beginning of time-slot, multiple senders contend by sending short preambles to R until one gets back R's ACK, then it sends out the data packet and waits for R's ACK again.

Broadcasts takes place in the dedicated time-slots. Broadcast channel is calculated using the same *pseudo-random channel selection function*, but they depend only on slot number. S will send the data packet repeatedly instead of sending the preamble, and will not get ACKs back.

2.4.3 EM-MAC

EM-MAC [11] is a fully asynchronous, receiver-initiated MAC. A Receiver, R, selects its channel and wakeup time based on a pseudo-random function using unique node parameters. It conducts a CCA (clear channel assessment) each time it wakes up on a channel and records the channel condition using a positive badness metric. R advertises its time, selected channel and bad channels list in a periodic beacon. A sender, S, uses the same pseudo-random function to guess R's wakeup time and channel. It waits on

every calculated channel for R's beacon until it finds the correct one. It can explicitly ask R to embed its state in the ACK.

Channel and wakeup time selection is done by R based on the *pseudo-random function of type: linear congruential generator (LCG):* $X_{n+1} = (a.X_n + c) mod\ m$ where a, c, X_0 are unique parameters for each node. The pseudo-random generator needs to be run twice to find a channel and a wakeup time. Each node resets the pseudo-random function seed to the initial value when reaching a threshold to keep the prediction computation overhead bounded.

Adaptive channel selection is done by R. It conducts a CCA each time it wakes up on a channel and records the channel condition using a positive badness metric. Collisions and consecutively failed CCAs increase badness metric, while successful transmissions decrease it. Bad channels are blacklisted. A bitmap of channels' status is embedded in R's wake-up beacons.

Channel rendezvous Initially, S uses the unique parameters a, c, X_0 for R to calculate a channel. S waits for R's beacon on that channel for T_{wait}, where $T_{wait} = T_{black} + 2.Number_{channels}.T_{maxReceiverWakeupInterval}$ (where T_{black} is the time a receiver can skip a bad channel). If S fails after waiting for T_{wait}, it calculates the next possible channel and waits on it again. If S still can't rendezvous with R after waiting on all possible channels, it concludes that R is not reachable. If S succeeds in communicating with R after this costly long waiting, it employs more advanced tricks to enable quick rendezvous on later tries.

***S* models of *R*'s time** such that it can guess the wake up time correctly. S models R's time as $T_R = k.T_S + b$. k represents estimated clock rate drift and is set to 1 initially, assuming that both of them have the same clock rate. b represents time drift. Once S's prediction of R's wakeup time differs a lot from the actual time, S requests a state update in the next ACK from R. Using the new information together with the old one, S can compute R's parameters k, b to figure out the actual clock rate of R.

***S* wakeup-advance-time** S calculates R expected wakeup time, and wakes up in advance to it, such that it can accommodate small expectations errors. When S misses R, it goes to sleep and tries again.

Exponential chase algorithm After two consecutive misses of R wake-up, the wakeup-advance-time is (repeatedly) doubled for each unsuccessful wakeup try. Once this margin surpasses $T_{giveup} > T_{wait}$, the sender drops the packet and discards the prediction state of R.

Broadcast can be supported 'by sending a broadcast packet to neighboring nodes one-by-one'.

2.4.4 MC-LMAC

MC-LMAC [14] is a fully synchronous TDMA MAC designed for converge cast. It employs a common control period on a common frequency.

The time is divided into slots. Each slot consists of a common frequency (CF) period, a control message period and a data period. The common frequency period is slotted into sub-slots representing the available frequencies. This protocol guarantees a unique channel/time-slot combination for each (sender) node in 2-hop distance. Occupied channels and slots information is included in control messages.

S, R Rendezvous All nodes listen to CF in the beginning of each time-slot. A sender S addresses R in S frequency sub-slot. R listens to CF each time-slot to check for potential senders. At the end of CF, both S and R tune to the selected channel. Then, S sends a control message. Finally, it sends the packet in the data period.

Broadcast It is supported easily by informing receivers in the CF period, thus; all nodes will listen to the broadcast in the chosen channel.

Channel and time-slot assignment A node trying to join the network listens to the network, and keeps a list of free slots in each channel. Once a carrier is detected in a time-slot on a certain frequency, it will be marked busy. Other motes transmit their own lists of time-slots as well. The node will use this information to make a list of possibly free slots, and will choose one randomly. The node sticks to the choice until a collision is detected. When a collision is detected, a node reports it, and if it matches another node slot, that node releases it, and restarts the slot selection process.

2.4.5 Chrysso

Chrysso [15] is a sub-layer that sits between the MAC layer (*i.e.* X-MAC) and the network layer (*i.e.* Collect).

This protocol is designed for data collection applications, and supposes that the network is formed as a tree with a sink node, parent nodes and children nodes. Children nodes collect data and send it to parents. Parents in turn are children to other nodes which they relay data to, till data reaches the sink node.

Channel switching mechanism Each node can perform as a *parent* where it listens on *in-channel*, and/or as a *child* where it collects and sends data to its parent on *out-channel* (which represents the parent's *in-channel*). Channel switch decision could be either parent-coordinated or self-initiated. Children monitor their respective *out-channel* and piggyback quality information, i.e. congestion back-offs statistics, with data packets. A parent periodically computes average of back-offs and decides to switch the channel according to a predefined threshold. When the parent decides a channel switch, it sends an order to children. The channel-switch order is piggybacked in the subsequent data packets ACKs telling them to move to next *out-channel*. Children, upon receiving the order, move to the next *out-channel*. However, when the children notice that the channel quality is so bad that the send failure rate is high, they perform channel switch independently. Similarly, a parent performs a channel switch when the ratio of received packets drops below a threshold. After performing a channel switch, the channel switch mechanism is paused for some time because the nodes are expected to have long queues of packets waiting for being sent.

Neighborhood (parent) discovery Chrysso employs a special *scan mode* that is triggered by the routing protocol on demand. Parents send routing beacons periodically on their respective *in-channel*. A node in *scan mode* listens to routing beacons from all parents on each channel, and then chooses the best parent. Neighborhood discovery is initiated when setting up the network, and when the routing layer decides that link quality to one parent is severe on all channels.

2.4.6 Comparison and discussion

The main features of the presented MAC protocols are summarized in table 2.2. Giving another look reveals that existing MAC protocols suffer from one or more of the following issues:

- Special purpose design that supports only one mode of communication *i.e.* convergecast, which makes the protocol not suitable for IoT. However, this is supposed to enhance the intended mode of operation;
- Synchronous design (TDMA), which supposedly reduces collisions and latency, but limits the MAC with
  - The need for synchronization, which can be difficult to achieve using cheap low bit rate IEEE 802.15.4 radios. *i.e.* MuChMAC suffered from synchronization and bootstrap problems [15];
  - The need of a schedule, which affects the dynamics of joining and leaving a network, and thus makes it unsuitable for IoT which is by nature highly dynamic;

- Receiver-initiated design, which supposedly eases sender/receiver rendezvous, but might add more complexity for *i.e.* dealing with colliding beacons;
- The use of a common frequency or common period for the whole network, which supposedly eases sender / receiver rendezvous, but limits the capacity of the system and makes it more susceptible to noise that happens in the common frequency;
- Limited broadcast support.

What is missing is a multichannel duty cycled MAC protocol that is asynchronous and sender-initiated, and in the same time supports dynamic networks, general purpose applications and different modes of communication (*i.e.* not specific to converge-cast).

Protocol	*Medium access*	*Frequency diversity*	*Common period/frequency*	*Beaconing*	*Broadcast*	*Goal*
Y-MAC	TDMA + collision window	Receiver hops channel on subsequent packets	CF	Sender initiated	Common channel	General purpose
MuChMAC	TDMA + CSMA	Receiver hops channel pseudo-randomly each time-slot	No	No	Dedicated time-slots	General purpose
EM-MAC	CSMA	Receiver hops channel pseudo-randomly and adapts to noise	No	Receiver initiated	Yes	Noise and clock drift adaptation
MC-LMAC	TDMA	Sender channel assignment	CP & CF	Sender initiated	Yes	Converge-cast
Chrysso	Operates over MAC	Change channel when bad	No	Yes	No	Collect-tree

TABLE 2.2: Comparison of studied sensors MAC protocols

3 Design and Implementation

In this chapter, we first present the details of the design of the proposed multichannel MAC protocol. Later on, we discuss the respective implementation-related aspects as well.

3.1 Design

This section introduces the design of a low-power duty cycled sender-initiated asynchronous multichannel MAC for WSNs and IoT. The proposed MAC inherits its basic design from ContikiMAC [16] and extends it to support multichannel communication efficiently. We begin by motivating the high level design choices, then we proceed to overview the operation of the protocol, later we present the design details of main parts of the protocol and finally we conclude with a summary of the design.

3.1.1 Motivating high-level design choices

The selected MAC characteristics can be easily motivated in the context of WSNs and IoT. *Low-power* characteristic is a must since most nodes in IoT and WSNs are battery-powered. Hence, *duty-cycling* is necessary as it tries to keep nodes' radios off as long as possible when communication is not needed since the radio is one of the most energy hungry components in a smart object. However, sender and receiver should be both awake for communication to take place; thus, one of them should take responsibility of initiating the communication. Choosing *sender-initiated* over receiver-initiated design is motivated by the observation that receiving nodes should minimize the unnecessary radio-on time especially when no communication is taking place. Receiver-initiated communication usually requires receivers to send beacons to advertise wake-ups; thus, it wastes more power than what is needed for sampling the radio medium for incoming communication. On the other hand, sender-initiated communication approach increases sending cost since the sender needs to send wake-up strobes to the receiver before actually delivering the packet. However, in both cases, the sender needs to stay awake for some time either listening for receiver's beacons or sending wake-up strobes. Since sending and receiving energy costs are comparable [33], and since nodes usually spend more time listening than sending, it seems more power-efficient to use *sender-initiated* communication. *Asynchronous* characteristic was chosen because the MAC needs to

support dynamic environments, where nodes can connect and disconnect from the network at any time. Asynchronous design makes the setup simple and self-configurable as it does not need tight synchronization, which is usually costly and a bit difficult to achieve using low-cost radios with relatively low bit-rates. *Frequency diversity* implemented by *channel-hopping* is the central feature in the MAC proposed in this thesis. It adds agility to wireless communication, mitigate the effects of multipath fading [2], improves reliability [1] and resilience to interference [11] and thus; supports coexistence with other wireless technologies operating on the same frequency band.

3.1.2 Overview

Since ContikiMAC proved to be very efficient in single-channel case, we choose to inherit its design and integrate channel hopping in it. Thus, our multichannel MAC operations are very similar. We can summarize the steps for communicating between two nodes in

- Medium access;
- finding receiver's wakeup-time and channel;
- data transmission and acknowledgement;
- and dealing with losses/collisions.

Moreover, we need to take care of

- selecting node's channel (channel hopping);
- and maintaining wakeup time and channel for future communication with the same receiver.

Next, we overview how a successful unicast attempt looks like; then, we explain details about each step.

Idle nodes, which do not have packets to send, keep their radios off for most of the time, and wake up periodically to sense the radio. This wake-up period is constant and is configured to be the same in the whole network. Each time a node wakes up to listen, it *hops* (switches) the channel according to a pseudo-random sequence. When a node detects activity on the channel, it keeps the radio on for a longer time trying to receive a potential packet. Only if a packet is received correctly, the node sends an acknowledgement packet that also contains information about wake-up channel. Then, it goes back to sleep. If a node S has a packet to send to another node R, it needs to know wake-up time and channel of R. Assuming that it already has this information for all neighboring nodes (described in more details later), S schedules the packet to be sent just before R's expected wake-up, switches to R's expected channel, probes it to ensure it is clear, sends the packet and waits for acknowledgement (ACK). If S receives

ACK, it knows that communication was successful, it updates its information of R's wake-up time and channel and goes back to sleep. Otherwise, S retries the same steps. However, when S fails to receive ACK after a certain amount of retries, it assumes that its information of R's wake-up time and channel is wrong and needs update.

3.1.3 Duty cycling and protocol timing

Nodes turn their radios off for most of the time and wake up periodically to check for incoming packets. We employ this duty-cycling to save power since the radio is one of the most power hungry components in a smart object, and idle listening consumes about the same power as sending (*e.g.* CC2420 radio chip [34]). On every wake-up, each node selects a channel and samples it for incoming packets by performing two CCAs. If CCAs detect a radio signal strong enough to be a potential packet, the radio keeps listening trying to decode it. Otherwise, it goes back to sleep. The wake-up period is the same for all the nodes in the network. It should be mentioned that the choice of a shorter wake-up period means more frequent channel checks; thus, lower latency and higher energy consumption, and vice-versa.

We actually use ContikiMAC duty cycling and protocol timing to facilitate detection of packets using only two CCAs and to quickly discard radio activity that do not conform to the timing constraints, assuming that it should be spurious noise. See section 2.3.2.

3.1.4 Channel selection: Frequency hopping

Channel selection is the central step in multichannel MAC. The choices made in this step affect the design of other parts of the MAC; specifically, channel rendezvous and broadcast. In this step we have to answer two main questions: Who decides the channel, and which channel to take.

For the first question, we decide that receiving nodes choose the channel. Each node switches its channel periodically on every wakeup cycle. When the node is busy receiving or sending a packet, and its channel switch time comes, it updates its internal state for the channel without actually changing the radio channel so it does not interrupt the ongoing operation. This, however, guarantees that future channel selection is not affected by past activities and depends entirely on node's internal schedule.

This leads us to the topic of how to choose the channel. We choose to follow pseudo-random sequences for channel selection. We do this with the intent of giving nodes uncorrelated channel sequences such that neighboring nodes are unlikely to have the same channel choice every time. Moreover, we hope that this provides better resilience against interference as nodes do not follow the same sequence all the time. In the same time, nodes do not follow a clear pattern for channel selection (*i.e.* increasing channel number); instead, nodes *hop* between channels pseudo-randomly. To achieve that, we

generate the pseudo-random channel numbers using the following equation which is taken from [35]

$$
\begin{aligned}
X_{n+1} = \quad & (aX_n + c) \bmod m, && n \geq 0 \\
\text{where:} & && \\
& m, \text{the modulus} && m > 0 \text{ and } m = N \\
& N, \text{is the total number of available channels} && \\
& X_0, \text{the seed} && 0 \leq X_0 < m \\
& a, \text{the multiplier} && 0 \leq a < m \\
& c, \text{the increment} && 0 \leq c < m
\end{aligned}
\tag{3.1}
$$

This kind of generators is called *'The Linear Congruential Generator (LGC)'*, which is well explained in computer science literature [35]. Using 3.1 we obtain the sequence X_n which represent the desired pseudo-random sequence. To generate the actual channel number, we apply 3.2

$$
Channel = \quad X_n + N_0, \qquad N_0 \text{is the first available channel i.e. 11} \tag{3.2}
$$

We choose this kind of generators because the sequences they generate are uniformly distributed and they are not complex to compute. The properties of the pseudo-random sequence depend on the chosen parameters: X_0, a, c, m. We try to select these parameters according to the theorems explained in [35] such that the generated sequences appear random and contain each possible number in the range exactly once before repeating the whole sequence again. However, the generated sequence will be repeated after a specific number of generated values (*e.g.* when $X_{n+1} = X_0$), but we can try to generate another sequence after the first one has ended instead of repeating the same sequence again by choosing a set of the parameters a and c for each nodes, and using the possible combinations of the values in the sets. Therefore, we can generate long unique sequences if needed.

3.1.5 Sender/receiver rendezvous

Obviously, to be able to communicate, the sender and the receiver should be on the same channel and since we employ duty-cycling, they should be both awake and sending/listening in the same time. We choose a sender-initiated design such that receiving nodes operate independently from senders, wake up asynchronously, choose channel independently and do not advertise any information on wake up. However, senders have the obligation of waking up the receiver and catching it on its operating channel. ContikiMAC solved the problem of unknown wake-up time by strobing the receiver by sending the intended packet repeatedly for up to the whole wake-up period until it receives an ACK from the receiver. We use a similar technique, but we extend it to multichannel taking in consideration how receivers choose channels as presented in the previous paragraph.

Initial rendezvous

Since the receiver uses a uniformly distributed pseudo-random generator for picking the channel, and assuming the number of available channels is N, we can expect a certain channel to be picked with a probability $^1/_N$. Since the receiver hops its channel every wake-up cycle, we can expect it to pick a certain channel once every N cycles. This might lead us to believe that if the sender picks any channel from the receiver's sequence, and strobe it continuously for N cycles, it will find the receiver eventually. However, giving a second look reveals that this is not accurate. Keeping in mind that nodes are not synchronized, and that channel sequence is random; we can infer that we do not have a clue which channel should the receiver pick in the next wake-up cycle, which channel was chosen before, and when would a certain channel be picked up in later cycles; *i.e.* a sender might be unluckily trying to probe on a channel that the receiver has just chosen, and that channel will appear the last in the pseudo-random sequence after all other channels appeared twice.

Consider this illustrating scenario: Sender S picks channel 2 to strobe the receiver R just exactly after the receiver checked 2 and went to sleep after finding no activity. R is using the following sequence with 4 channels: $2, 1, 3, 4, 3, 1, 4, 2$ This means that R will pick channel 2 again after 7 wake-up cycles, and if S stops strobing channel 2 after 4 cycles, it will not be able to rendezvous with R.

This simple example showed that a sender might need up to strobe a channel for up to $2N - 1$ periods in order to catch up the receiver, even with a perfectly uniformly distributed random sequence. This is surly costly, especially with long wake-up periods (that are usually chosen to save power), and with a lot of available channels (a larger N). Therefore, we optimize this for subsequent packets sent from S to R using a technique similar to *phase-lock* in ContikiMAC. However, we evaluate the cost of this later. Anyway, we can choose to have repeated pseudo-random sequences instead of the long ones, such that only N channel strobes suffice. To illustrate this in the terms of the previous example, R will be using the following sequence with 4 channels: $2, 1, 3, 4, 2, 1, 3, 4$ This means that R will pick channel 2 again after 4 wake-up cycles.

Figure 3.1 illustrates the initial rendezvous for unicast in the case of two channels.

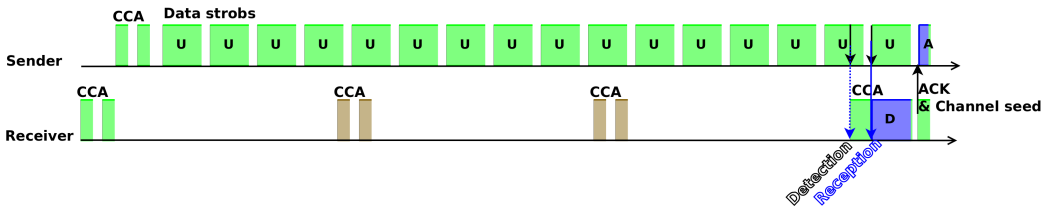


FIGURE 3.1: Multichannel MAC unicast with two channels

Phase and channel lock

When a sender S is probing a receiver R, it waits for ACKs between probes, so that it knows when R gets the packet. S saves the time it sent the last successful probe (which was ACKed), and thus, it has now information about R's wake-up time, as it is following periodic wake-ups. However, it does not have enough information about R's next wake-up channel because channel choice follows a pseudo-random sequence, which depends R's parameters.

To solve this, R sends channel information in ACK packets. Specifically speaking, it sends information about the pseudo-random sequence such as S is able to figure out the subsequent wake-up channels. S constructs a table of neighbors that it communicated with, and stores their related information, like wake-up time, and channel parameters for future reference. When S wants to communicate with R, it consults its private neighbors table. If it finds an entry, it schedules the packet to be sent to R on the expected channel and starts strobing R before a short *guard-time* of the expected wake-up, so it compensates for small timing errors. If S receives an ACK from R, it updates the parameters in the table. Otherwise, it tries again. After a limited number of failed attempts, it removes R entry from the table and tries the initial rendezvous again, by selecting a possible receiver channel and strobing it for up to $2N - 1$ wake-up cycles until it gets receiver's ACK. See figure 3.2.

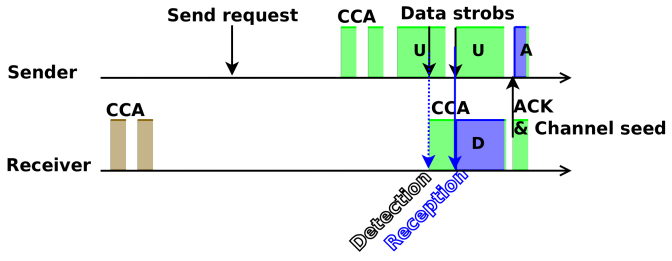


FIGURE 3.2: Multichannel MAC unicast with phase-lock (2 channels)

3.1.6 CSMA and packet queue

We inherit the use of *CSMA* such that collisions from neighboring non-hidden nodes are reduced, thus, improving throughput [19]. As explained in section 2.3.2, senders perform CCA before trying to send the packet. If an ongoing signal is detected, the sender backs-off and schedules a later retry. It should be noted that the asynchronous channel strobing (i.e. senders having different views for the same receiver wake-up time and employing a guard-time) play almost the same role as the initial random waiting in usual CSMA systems. CSMA does congestion control with hidden nodes as well, as it interacts with the MAC and performs back-off when the MAC reports that no ACK was received. These altogether reduce collisions and improve fairness.

Packet queues play a significant role in improving performance [19] in cases of collisions, delayed sending due to lack of channel information and sudden packet flow in the network. In the mentioned examples, the delivery of a packet destined to a receiver is delayed. In the same time, more packets destined to the same receiver might arrive. This optimization caches those packets and tells the receiver to expect more packets after the first one is successfully delivered. The receiver then stays awake waiting for more packets instead of going back to sleep after receiving only one. The sender can send the rest of the queued packets consecutively instead of having to wait for future wake-ups of the receiver. Thus, the overall latency will be minimized and the delay caused by the retries and/or channel strobes will be hidden, giving a higher overall throughput. This technique employs the bursty channel characteristic of wireless links in a smart way, and lets the sender to take advantage of good channel conditions that allowed the first packet to be delivered successfully. This helps bringing down the overall cost of *initial channel rendezvous* in a great deal.

3.1.7 Broadcast

Broadcast support is important in networks as it is used to relay information to all nodes in a specific part of the network. Routing protocols are one example of higher layer protocols that use this extensively. Multicast in IPv6 might be implemented using MAC-layer broadcasts as well. The main advantage of broadcast is that it delivers packets to nodes based on their presence in a neighborhood instead of their specific addresses. In the special case of wireless networks, a node's neighborhood is defined mainly by its wireless range; *i.e.* nodes that can hear a specific node N are said to be in the neighborhood of N.

Broadcast in the multichannel duty cycled asynchronous MAC is challenging due to several reasons

- nodes are not listening to the radio in the same time. Instead, each node wakes up once every wake-up cycle (once in a wake-up period);
- nodes are not guaranteed to be listening to the same channel in the same time. Instead, they are hopping between available channels randomly.

Next, we propose two methods for implementing broadcast: Continuous strobing and broadcast channels

Continuous strobing

To ensure the broadcast reach to all idle-listening nodes in an ideal collision-free environment, we pick any possible channel and send the packet repeatedly on that channel for $2N - 1$ wake-up cycles, where N is the number of available channels used for listening;

as shown in figure 3.3. This is similar to *Initial rendezvous* explained earlier. However, the broadcast is sent in a best-effort approach and not ACKed. This method is costly for the sender. It clearly costs up to $2N - 1$ times of a unicast. However, the latency and power consumption for receivers are kept the same, except that it does not require an ACK. Anyway, if we choose to have repeated pseudo-random sequences instead of the long unique ones, then only N channel strobes suffice.

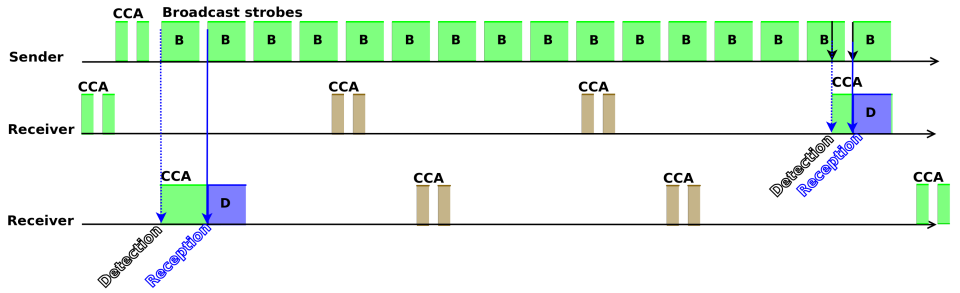


FIGURE 3.3: Multichannel MAC broadcast (2 channels)

Broadcast channels

Instead of strobing a channel for a long time, we provide a channel (or channels) for sending broadcasts; as shown in figure 3.4. Each receiver wakes up four times each cycle (two CCAs for each check), first checking the unicast channel (as explained earlier in 3.1.4), then performing another check for broadcast channel. This means that we guarantee that all idle nodes in the networks check the broadcast channel once every wake-up cycle. Thus, when a sender wants to send a broadcast, it is enough to switch to the broadcast channel, and strobe it for one wake-up period only; exactly as a ContikiMAC sender does.

The broadcast channel check might implement channel hopping as well if desired. Receivers hop between broadcast channels as well every wake-up cycle. In this case, a sender needs to pick a broadcast channel and strobe it for $2B - 1$ wake-up cycles, where B is the number of broadcast channels, in the case of long unique pseudo-random sequences, and only B channel strobes if broadcast channel uses short repeated pseudo-random sequences (as explained before).

In both cases, the idle listening cost increases twice; as we check radio channels twice each wake-up cycle (two CCAs for each check). The main advantages are a lower energy cost for sending a broadcast, less congestion due to shorter broadcasts and less latency as the broadcast takes a shorter time to be sent. Thus, we can anticipate that in a broadcast-dominated application this approach might work better.

Note that we can employ the broadcast channel for *Initial rendezvous* as well; where the sender sends the unicast packet destined to a receiver with unknown channel on the broadcast channel. This can reduce *Initial rendezvous* cost greatly.

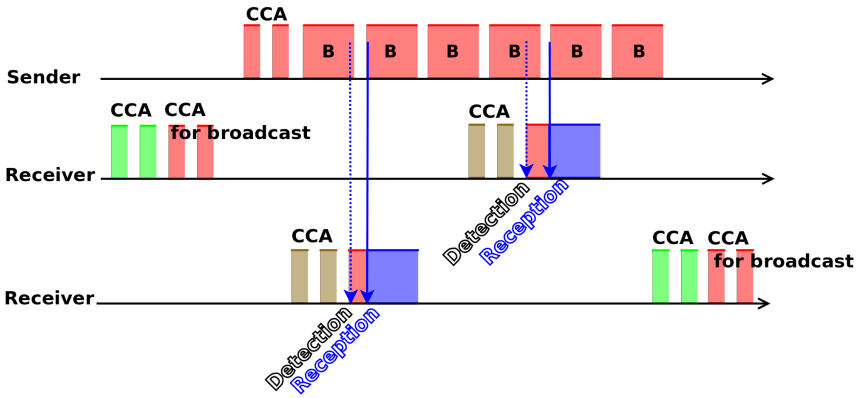


FIGURE 3.4: Multichannel MAC broadcast with a broadcast channel

3.1.8 Summary and discussion

The proposed multichannel MAC inherits the design of ContikiMAC, inherits CSMA and packet queue and extends it to include channel-hopping multichannel communication. It can be summarized by the following points:

- We use pseudo-random channel sequence;
- Receivers hop wake-up (receive) channel each cycle;
- Senders initiate communications by strobing one channel for $2 * N - 1$ cycles when the receiver's channel is unknown, when using long unique sequences, or N otherwise;
- The receiver sends an ACK including channel seed;
- The sender uses this for phase and channel lock optimization: It remembers wakeup time and channel seed;
- Broadcasts can be implemented either by:
  - Strobing all channels, which costs $2 * N - 1$ strobes for long unique sequences, or N otherwise;
  - or by employing dedicated broadcast channel(s), where each receiver should perform two channel checks each wake-up (two CCAs each). This increases the overall listening cost, but potentially reduces broadcast cost.

It should be noted that the design of some features is inspired from existing sensor MAC protocols. Specifically speaking, the use of LGC pseudo-random generators to obtain hopping sequence is used by MC-LMAC [14] and EM-MAC [11]. Moreover, the idea of the asynchronous channel hopping and initial channel scanning is similar to the technique used by EM-MAC [11]; however, we use sender-initiated approach while the former uses receiver-initiated approach.

3.2 Implementation

We provide a C implementation that is specific to Contiki OS version 2.6. Actually, the protocol is implemented as an extension to the existing implementation of ContikiMAC. We reuse most of the code of ContikiMAC and other related modules such as the neighbor tables and radio drivers, and we provide our own versions of them that are compatible with the rest of Contiki. In this section we describe the implementation platform, the run environment and the implementation choices we made.

3.2.1 Platform and run environment

The implementation is made for TMoteSky [36] sensor mote, which is a low-power and resource limited platform. TMoteSky main features are [36]

- IEEE 802.15.4 250kbps 2.4GHz Chipcon CC2420 radio;
- 8MHz Texas Instruments MSP430 8bit microcontroller with 10k RAM and 48k flash;
- humidity, temperature, and light sensors;
- USB interface for programming and data collection.

However, the implementation is generic for Contiki OS, and is expected to work as is on Contiki supported platforms, with an exception for the radio-specific part which should be ported to the specific radio. Next we describe how we add the necessary functionality to the existing implementation to make it multichannel.

3.2.2 Channel switching

Contiki's radio driver interface provides a generic structure for controlling most functions of the radio; such as, turning it on and off, sending and receiving. Thus, most radio functionality is portable to all supported platforms. However, it does not provide a generic function call to switch the radio channel. Consequently, we had to use a radio-specific function, that should be changed when porting to a new platform. The channel-switch function we provide *i.e.* `set_channel(int channel)` should be modified to call corresponding radio-specific method. This function is used to set the channel before sending and to hop the wake-up channel as explained in the following section. It should be mentioned that the radio standard we use supports up to 16 channels to choose from. They are given logical numbers from 11 to 26. They mostly overlap with WiFi channels except for channels with numbers (11, 15, 25, 26) according to [1].

3.2.3 Pseudo-random channel sequence

The channel should be picked up from a pseudo-random sequence as explained in the design section, however, instead of calculating the pseudo-random sequence in run time, we generate an array of pseudo-random numbers that represent radio channel logical number. This array is defined statically in code and shared among the nodes. Each row in this array represents a pseudo-random sequence that is generated using a generator with parameters different than other rows, and each node chooses a row in this array once and sticks to it as long it has the same network address. Each node maintains internal variables that stores the position/index in the hop-sequence, which corresponds to n in X_n which is shown in equation 3.1. This index is used to access the hop-sequence array and pick the next channel in the sequence.

It should be noted that the number of channels to use, whether or not to use dedicated broadcast channels and how many, can be configured by defining macros in the file `net/mac/happymacconf.h`, while the set of channel-hopping sequences is defined in `net/mac/hopsequence.h` such that the hopping sequence matches the chosen configuration. All these configuration are to be defined statically at compile time, just like other ContikiMAC-specific settings such as the wake-up period (`CYCLE_TIME`). In the next section we explain how channel hopping is implemented.

3.2.4 Channel hopping

At the beginning of each wake-up cycle, which consists of two or more CCAs, the node hops its receive channel by calling the macro `HOP_CHANNEL` which actually increases the internal channel index and returns the corresponding channel in the sequence. At this moment the receive channel is just updated internally, by storing it in the local variable `rec_channel`, eventually, the radio is ordered to switch the channel to `rec_channel` by the function `powercycle_turn_radio_on`, which ensures not to interrupt ongoing send operation or reception of a stream (queue) of packets. Next we explain how a sender gets a receiver channel information and how phase-lock and channel-lock are maintained.

3.2.5 Channel rendezvous and phase-lock

A sender strobes a receiver by repeatedly sending a packet to ensure it is delivery to the receiver which is duty cycled and hopping channels. A sender knows that a packet was delivered successfully when getting an ACK back form the receiver. The sender then saves the time it started sending the last successful strobe; as it corresponds to the receiver wake-up time. This could be known with out getting special time-stamp from the receiver as it is supposed to wake-up in the same time next cycle, and as it is supposed to have a clock that does not drift by more than 40 ppm according to IEEE 802.15.4 standards [5]. However, as a countermeasure against drifts, the sender keeps a guard-time and starts sending strobes ahead of the receiver expected wake-up time.

On the other hand, the sender can not deduce the next wake-up channel for a receiver even if it knows the current one because each receiver follows a different pseudo-random hopping sequence. Thus, it needs to get information about that sequence, which, in our implementation, is the channel index on the hop-sequence array. The receiver includes this index in the ACK packet, and the sender extracts this index for later use. The details of creating this specialized ACK that include channel information are explained in the next subsection. However, now we proceed to explain how a sender performs phase-lock.

The sender has obtained the channel index and the wake-up time for that sender. It creates a list that includes a structure for each receiver that it successfully communicated with and keeps the information specific to that receiver in this structure. It stores the receiver address, the wake-up time, the number of times it failed to send to, and a timer to schedule requested sends to happen just before the expected wake-up time. All this functionality was already implemented in `net/mac/phase.c`. We provide new functions that implement the explained functionality, and we keep the old functions for backward compatibility as well. We extend the implementation to include the channel index in the structure and we modify the functionality to maintain that channel index updated correctly when the sender tries to send to the corresponding receiver. The idea behind maintaining the channel index is simple: Since the receiver increments the channel index each wake-up period, the final result is increasing the channel index by the same amount of how many wake-up periods has passed since last communication try. In order not to get affected much by clock drifts, we implement a guard time; such that if the receiver's expected wake-up time is earlier than the guard-time, it waits till the next wake-up cycle.

3.2.5.1 Exchanging channel information and soft-ACK

In the platform we use we have a CC2420 [34] radio-chip, which supports auto ACK that can not be modified, but supports disabling it. In this case, we had to modify the radio driver to generate a special software-generated ACK (soft-ACK) that includes the channel index, just after receiving a valid unicast packet. We modified the interrupt handler `cc2420_interrupt` that is triggered when a packet is being received. This modification is not straightforward as we had to take into consideration several aspects;

- not all packets can be ACKed; namely, broadcast packets, or corrupted packets;
- we have to wait until the packet is received in full as to perform CRC check;
- and we have to access the packet buffer to read the sequence number and the frame control field FCF which determines whether an ACK is requested by the sender.

This hack we implemented for soft-ACK had some drawbacks, such as;

- accessing the packet buffer is not straightforward as it is not well supported in the chip interface;
- the soft-ACK incurs increased delay than what optimally required in the standard ($192\mu s$ vs. $250\mu s$) since we had to wait for the packet to be received, construct the soft-ACK packet and send it;
- and we had to flush the chip transmit buffer before sending the soft-ACK, which might cause some disturbances for applications that had a packet to transmit in that particular time.

However, all of this part is hardware specific and could be either much more efficient on some platforms, or not efficient at all on others.

3.2.6 Implementation structure and organization

This section provides detailed information about the implementation to make further development easier for the interested reader. The protocol is implemented by tailoring existing code of ContikiMAC. Therefore, we present the code structure of ContikiMAC to make it easier to handle our extension. The implementation utilizes the following files

- ContikiMAC source file `net/mac/contikimac.c` and corresponding header file `net/mac/contikimac.h`
- Phase-lock neighbor table source and header `net/mac/phase.c` and `net/mac/phase.h`; which define the interface and structure of the table responsible for keeping phase-lock information of the neighboring nodes to optimize the cost of sending.
- Radio driver for CC2420 chip source and header files `dev/cc2420.c` and `dev/cc2420.h`
- The new files `net/mac/hopsequence.h` which defines the hopping sequences and `net/mac/happymacconf.h` which configures the number of channels to use and whether or not to use a broadcast channel.

The interface functions of ContikiMAC, which can be called by upper-layers are

- `init` for initializing the radio and the used structures (timer and the protothread),
- the queue send functions `qsend_packet` and `qsend_list` which are responsible for scheduling a packet or a list of packets to be sent. They are called by upper-layers.
- The receive function `input_packet` which is responsible for turning the radio off after receiving a packet and delivering the received packet to upper-layers.
- `turn_on` and `turn_off` for enabling/disabling duty-cycling.
- `duty_cycle` for reading the configured duty-cycling value.

The actual operations of those interface-functions rely on the following utility functions that support lower-level operations, and, thus, we might need to modify:

- The wake-up cycle which performs periodic channel checks, detects and receives incoming packets and turns the radio off for the rest of the time. It is implemented in the function `powercycle` using a real-time timer (`rtimer`) that schedules the *protothread* that runs the function `powercycle` periodically;
- the code that actually schedules `powercycle` is `schedule_powercycle`;
- the wake-up function `powercycle` uses `powercycle_turn_radio_on` and `powercycle_turn_radio_off` to turn the radio on and off while ensuring correct operation like not interrupting on going send operation for example;
- the actual functions that turn the radio on and off are `on` and `off`;
- and the send function `send_packet` which is responsible for actually sending a packet.

It should be noted that the implementation defines many C macros to represent specific MAC configuration, such as the wake-up period `CYCLE_TIME`, and ContikiMAC specific timing such as the time to wait between successive packets (strobes) `INTER_PACKET_INTERVAL`. We follow the same approach and make use of C macros to remain coherent with the rest of the implementation and to reduce the overhead of RAM access and possible pointers.

3.2.7 Summary

We provide a C implementation in Contiki for our protocol, we reuse much of the available code for ContikiMAC, and we try to provide backward compatibility for some components like the phase-lock tables. We change the ContikiMAC ID in ContikiMAC header as well from 0 to 1; such that further compatibility with single channel ContikiMAC can be supported later. We also provide predefined hopping sequences and make use of macros such that it is easy to configure the protocol in terms of number of available channels and the option to use dedicated broadcast channel.

4 Evaluation

In this chapter we test and evaluate our protocol in a scenario that uses the IPv6 stack in Contiki; as to show how well it works integrated with other components. We add an interferer too to show how robust it is against external interference. We begin by describing the setup of the experiments and the related nuts and bolts, then we proceed to describe the evaluation metrics, we follow this a description of evaluation methodology and the experiments we run and we present the results we obtain, and finally we conclude with a discussion of the results.

4.1 Experimental setup

In this section we describe the test environment we use for evaluating our protocol. We begin with a description of the example use scenario and the overall setup, then we explain in a bit more details the components of the evaluation environment.

4.1.1 Overview

The scenario we try to evaluate is a data collection example. This is based on an example available with Contiki source, and it is intended to demonstrate the use of MAC with RPL and UDPv6.

The example consists of sensor nodes sensing various aspects of the environment; such as, temperature, light intensity and humidity, combined with internal measurements of each node's power consumption. These sensors send their readings periodically in a multihop network to a sink node which collects the measurements and performs as the root node of RPL; thus, creating a RPL DAG to enable forwarding the data from sensor nodes to itself. The network is visualized in Figure 4.1. It consists of 1 root node and 24 sensing nodes (nodes 2-25). The nodes are distributed as shown in the figure, where each square in the grid is $10m^2$. We modify the original example such that the nodes use our multichannel MAC, and each node provides periodic local log outputs containing the internal measurements of power consumption in addition to logging each sent message as well.

The nodes represent TMoteSky motes (see Section 3.2.1), which are emulated using MSPSim [37] emulator that mimics the actual hardware of TMoteSky and emulates the real hardware in cycle-accurate way. It is important to point out that we load the

exact same binary as we should use on real HW (based on the real implementation). Moreover, the whole network is simulated using COOJA [38] simulator which simulates the radio medium and provides time-accurate simulation of the network. Further, we add an interferer node (26) which is simulated in COOJA and can be configured to emit controlled bursty interference in the network, to test the protocol's resilience to interference. In the later subsections, we provide a slightly detailed description of the main components of the setup.

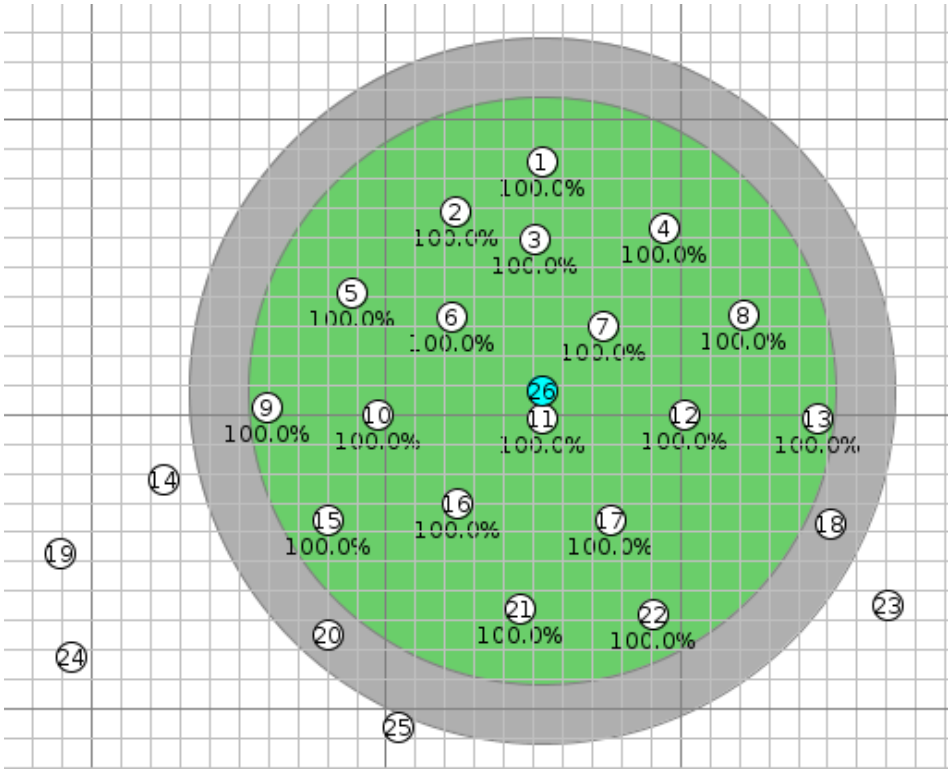


FIGURE 4.1: Data collection network with an interferer node. The circles show the interference range.

4.1.2 MSPSim

MSPSim [37] is a Java-based emulator for sensor nodes based on MSP430 CPU. It mainly emulates the MSP430 CPU in instruction-level, but it also supports emulating the whole sensor board with the radio-chip and other peripherals and sensors. For example, it provides a cycle-accurate emulation of TMoteSky platform out of the box; including the CC2420 radio chip, buttons and sensors. It supports loading the compiled binary file for TMoteSky, executes it as the real hardware does, provides debugging hooks and supports integration with COOJA to simulate a whole network.

4.1.3 Cooja

COOJA [38] is a cross-level simulator that simulates wireless sensor networks using different levels of abstraction, and simulates the radio medium and the various network events such as collisions. A node in COOJA can be simulated either in

- *Networking level* where it can be implemented in Java to simulate high-level operations such as sending a packet, as we do for simulating the *interferer* node;
- or in *OS level* where the node code is Contiki C code but runs on the host machine natively, so it does not reflect real timing and hardware constraints of sensor motes;
- or in *Machine code instruction set level* where the node is emulated using MSPSim and the loaded code is actually a Contiki C code that is compiled to the target hardware environment.

Cooja supports simulation of nodes of mixed types as well. In the same time, Cooja simulates the radio medium using simple models. We use the *unit disk radio medium model* as it is simple, straightforward, assumes perfect network conditions and thus can easily show the effect of added interference.

Overall, Cooja provides a very usable environment that visualizes radio events, captures packets, monitors variables, visualizes the network, shows nodes' logs and supports scripting and test automations. The interface of Cooja is shown in Figure 4.2. It eases the development and evaluation of network protocols and embedded development by a great deal.

Next, we describe in more details the used radio medium model and the generation of interference.

Unit disk radio medium model

It models the radio range of each mote as two circles; an inner circle that represents the effective communication range, which is characterized by successful communication to nodes in that region, and an outer circle that represents the interference range, which means that nodes in that region can hear the signal but can not interact with the sender; *i.e.* will just interfere the communication in that region. In this model, the medium is assumed to be lossless, and communication always succeed unless interfered. interferer When a node is within the interference range of two or more nodes and hears their transmissions in the same time, a collision happens in the vicinity of that node. Under this radio model, all collisions are assumed to cause packet losses. Each node is assumed to have up to $100m$ interference range with $50m$ of it functioning as transmission range, by default. Figure 4.3 illustrates the transmission range and the interference range of a node. Nodes 1 and 2 can communicate with each others, while nodes 1 and 3 will only interfere each other. Similarly, node 2 can communicate with both 1 and 3.

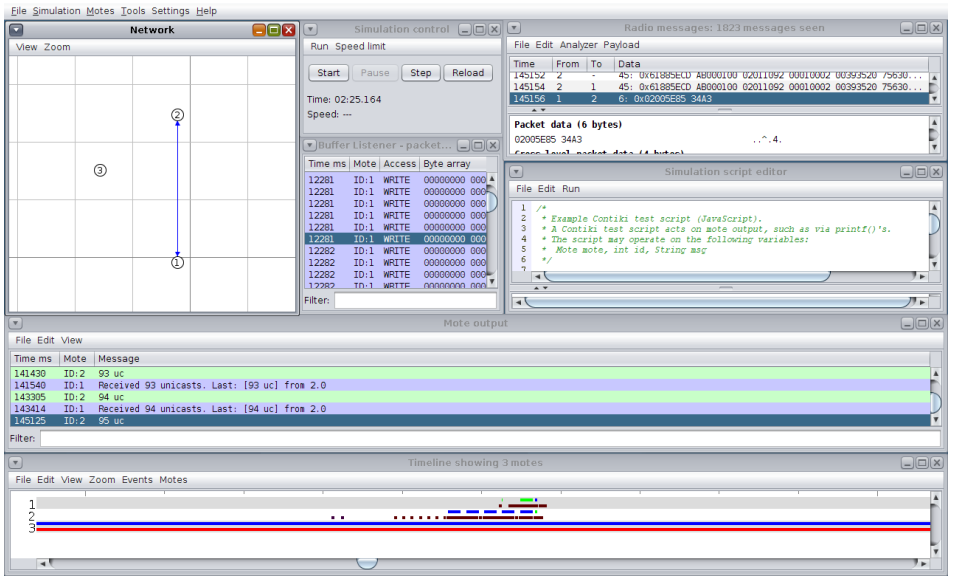
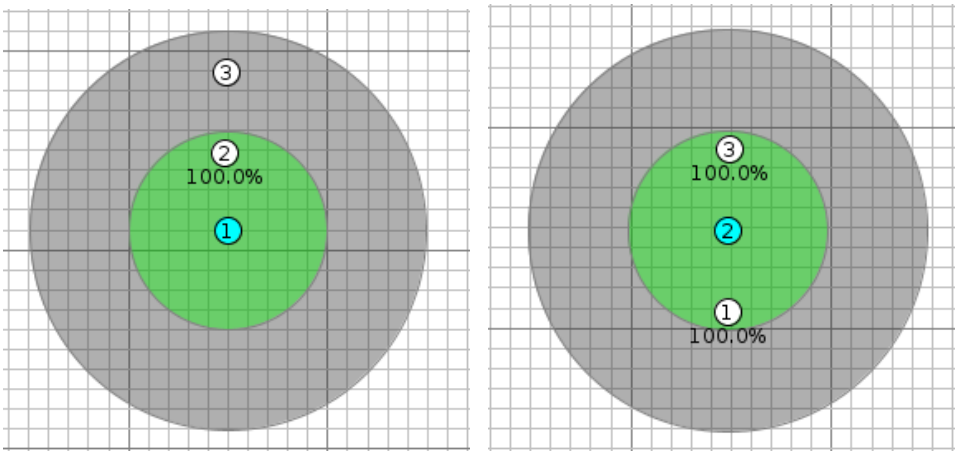


FIGURE 4.2: Cooja interface



(A) Node 1 communicates with 2 and interferes with 3

(B) Node 2 communicates with both 1 and 3

FIGURE 4.3: Unit Disk radio model

It should be mentioned that Cooja's support for multichannel communication was buggy, and we had to fix it. Originally, we experienced some misses of ongoing transmissions in a channel that a receiver has just switched to. However, we provided a fix for this bug that was caused by internal timing of radio events generation in Cooja's radio mediums, which did not consider receivers waking up on a channel different from the last one in a middle of ongoing transmission.

Interferer

To simulate interference in Cooja in a controlled and simple way, we implement a simple interferer node in Java as a Cooja network-level simulated node. We choose to implement it as a semi-periodic bursty interferer since it is a simple model that behaves like a bursty traffic generator; thus, resembling a simplified WiFi or Bluetooth transmitter.

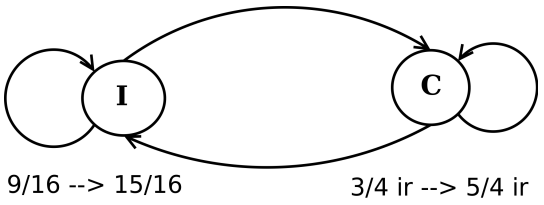


FIGURE 4.4: Interferer model

The model is described in [8] as a state machine that has two states; clear state *C*, where it does nothing, and interfere state *I*, where it generates packets. The interferer is configurable by setting a variable *ir* that represents *interference rate*. It operates as following; it stays in state *I* for a time *ti* that is uniformly distributed between $^{9}/_{16}$ and $^{15}/_{16}$ seconds, then, it transits to sate *C* and stays in it for a time *tc* that is uniformly distributed between $^{3}/_{4} * ir$ and $^{5}/_{4} * ir$ seconds. This model is illustrated in Figure 4.4.

4.2 Experimental Evaluation

In this section, we describe the experiments we run to evaluate our multichannel protocol and the evaluation metrics, then, we present the results.

4.2.1 Experiment description

We evaluate the protocol based on test runs in Cooja simulated environment with emulation of TMoteSky nodes. The nodes run the compiled binary of IPv6 data-collection program that is implemented using UDPv6 on top of RPL and our multichannel in Contiki. The network we simulate is illustrated in Figure 4.1. It consists of

- One emulated *root node* that stays always on and acts as a UDP server that is the sink of the collection tree, and the root node of RPL in the same time;
- 24 emulated *sender nodes* that are duty cycled and act as UDP clients that sense, collect and send
  - measurements of internal power consumption obtained using Contiki's *energest* library [31];

  - and the environment temperature, humidity and light intensity obtained using their hardware sensors;

- 1 *interferer node* that is simulated using Java code in Cooja network level. It generates semi-periodic bursty interference (as described previously) on one channel only (specifically channel 24), which is used in all hopping sequences. The interferer is node 26 as shown in Figure 4.1. The circles show the interference range.

The *sender nodes* send messages to the *root node* periodically every 60 - 62 seconds. The size of the message is 46 bytes. They produce periodic local logs to their USB-serial interface every 5 seconds as well; to take fine grain measurements for evaluation. They forward other nodes messages as well since they form a RPL-DAG together. The *root node* collects the measurements from the *sender nodes* and logs them to its USB-serial interface, too.

The messages sent by *sender nodes* have the structure shown in table 4.1, the *senders local log messages* have the structure shown in table 4.2, and *sink log messages* have the structure shown in table 4.3. It should be mentioned that Cooja prints two additional fields in the beginning of each log message; the simulation time in microseconds and the ID of the node that output the message. A sample log output is shown in Figure 4.5.

Field name	*Description*
Sequence number	A sequential number that identifies packets sent from the same source. It takes values between 128-255 and the program wraps it automatically so it can detect node restarts, which make it 0.
Length	Size in bytes of remaining fields in the message.
Clock	Local time indication.
Synchronized time	It uses *timesynch* library in Contiki, if enabled, to synchronize nodes' time.
CPU	The total time the node spends using the CPU.
LPM	The total time the node puts the CPU in low-power mode.
Transmit	The total time the node's radio spends sending.
Listen	The total time the node's radio spends listening or receiving.
Parent	The last two bytes of node's RPL parent.
Parent ETX	The estimated transmission count (ETX) to reach the node's parent.
Current routing metric	Node's rank in RPL.
Number of neighbors	The total number of neighbors in transmission range that the node hears routing advertisement (DIO) and can be possible parents.
Beacon interval	The interval (in seconds) of RPL routing advertisements (DIO).
Sensor readings	The readings of attached sensors. This field is made big (10 bytes) so it could handle more sensors when the code runs on a different mote type.

Table 4.1: Sender message structure

4.2.2 Evaluation metrics

We focus on three performance metrics

Field name	*Description*
Node address	It is compacted by reporting only one number that is obtained by shifting the second last octet of the address and adding it to the last octet.
String	This says whether this is only a local log (*local_log:*), or it is actually sent (*sending_msg:*).
Msg	This is a structure similar to actual sent message except for the last 10 bytes which are ignored since they represent sensor readings which are not necessary for calculating evaluation metrics.

TABLE 4.2: Sender local log message

Field name	*Description*
Number of fields	How many fields there are in this log message.
Node time	Higher two bytes of local time.
Node time	Lower two bytes of local time.
Separator	It puts 0 to separate fields in the message.
Sender's address	It is compacted by reporting only one number that is obtained by shifting the second last octet of the address and adding it to the last octet.
Sequence number	A sequential number that identifies packets sent from the same source.
Hop count	The number of hops the received message has passed through.
Separator	It puts 0 to separate fields in the message.
Msg	This is a structure similar to actual sent message except for the first field, which is the sequence number, as it is printed earlier.

TABLE 4.3: Sink log message

```
1574 ID:25 6425 local_log:  128 22 8328 0 2830  38127 0 398
3240 ID:25 6425 sending_msg:129 22 8541 0 21508 46695 11982 824
7814 ID:1 30 0 71 0 6425 129 3 0 22 8541 0 21508 46695 11982 824
```

FIGURE 4.5: A part of a sample log output

- *Duty cycle*, which represents the average ratio of radio usage times (sending and listening) to the total run time for one node, averaged among all nodes, except the sink which stays always on. We calculate the duty cycle for one node by summing the radio usage time and dividing it by the total run time, using the formula 4.1.

$$DutyCycle = \frac{TransmissionTime + ListeningTime}{CPUtime + LPMtime} \tag{4.1}$$

  Then, we calculate a weighted average for all sender nodes, and plot it;

- *Latency*, which represents the average time it takes a packet to be delivered from the source to the sink. Cooja reports the exact time in microseconds a node sends a message, and the time the root received a message. We match messages using node's address and sequence number included in the log; then, we calculate the difference between the two reported times, and; finally, we average this for all received packets in the network;

Next we explain how we perform the experiments and demonstrate the results we obtain.

4.2.3 Evaluation method and results

We make a test script that changes the protocol number of channels and number of broadcast channels and changes the interference rate of the simulated Java interferer, runs each experiment for one hour and stores the logs. We process the logs then using another script, that calculates the evaluation metrics and plots them. It should be noted that when we perform the experiment with only one channel, we use the original ContikiMAC, so it does not incur any overhead due to our multichannel MAC's channel management.

Varying number of channels and interference rate

We vary the number of channels between 1 and 16, and we vary the interference rate among the values 0 (no interference), 25%, 50%, and 75%. Each variation is run three times for one hour each. Each node sends one message per minute, and the total sum of sent messages in the network is about 1416 messages per experiment.

Duty cycle Figure 4.6 shows the average duty cycle of sender nodes in the network. The first phenomenon we notice is '*Increased interference increases duty cycle*'. The single channel MAC incurs the most penalty in comparison to multichannel MAC. The explanation is straightforward. Since the interferer is tuned on the same frequency used by the single channel MAC, the nodes in the network have a higher probability to coincide a carrier on the radio while performing periodic channel checks. Moreover, the interference collides with packets in the air causing irreversible errors in reception, henceforth, causing the packets to be discarded by the receiver, which in turn causes the CSMA sublayer in senders to retry sending. All these events require nodes to use their radios for longer times, which means increased radio duty cycle. On the other hand, using more channels tends to lower the duty cycle because nodes have a better chance of listening or sending on a clear channel; similarly, the legitimate communications are distributed on more channels; thus, the network capacity is increased and contention is reduced. We can notice a big improvement even by using only 2 channels because each receiver has a 50% chance of listening to a clear channel (on average). Generally, we notice that the cost of interference in terms of duty cycle is not very high; thanks to the fast sleep optimization that reduces the penalty of interference. It can be explained easily: Nodes get awaken up by the interference, try to decode a possible packet, then ignore it and go back to sleep after finding out that the signal does not follow ContikiMAC timing.

We can easily see that the multichannel MAC lowers the duty cycle when interference rate is 50% or higher, while it keeps a linear base cost with respect to number of channels in the absence of noise. Moreover, the duty cycle of the multichannel MAC does not increase much even when increasing interference rates up to 75%. In comparison to single channel ContikiMAC, the 16-channel MAC has a higher base cost in absence of

noise, but a better resilience to interference, a more stable behavior and possibly a lower cost when subject to interference. A closer look to the figure shows that the single channel ContikiMAC has an average duty cycle of 0.77% in clear channel conditions, while the 16-channel MAC has a duty cycle of 1.80% which is 2.33 times higher. On the other hand, the duty cycle for single-channel MAC increases to 3.35% (more than 4.3 times higher than its best case) when the interference rate is 75%, while the 16-channel MAC has a duty cycle of 1.95% with only 8.3% increase to its best case, and only 58.2% of the duty cycle of ContikiMAC.

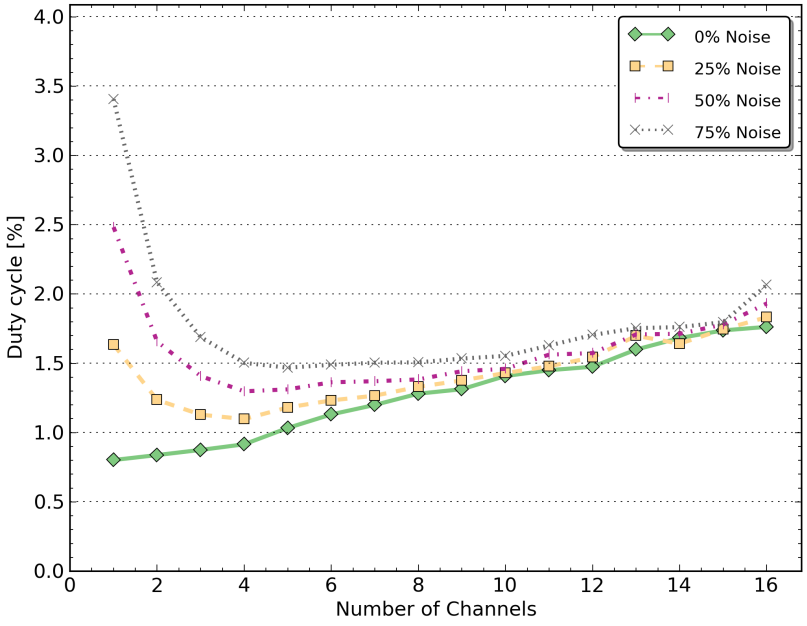


FIGURE 4.6: Duty cycle vs. number of channels and different interference rates on one channel. The results show that using more channels increases the resilience to interference.

Latency Figure 4.7 shows the average latency for delivering a message from a sender node to the sink node in the network. It should be noted that the network is multihop, and a packet could need 2 to 3 hops to reach the sink. The interference causes the single channel MAC to suffer the most such that it has the highest latency in comparison to any multichannel flavor. However, the explanation is straightforward. Each sender checks the channel three times before sending a packet, and defers sending on busy channel condition. Since the interferer is keeping the single channel that is used by single-channel MAC busy, sending a packet incurs large delays; especially that it has to be forwarded on multi-hops. On the other hand, channel-hopping MAC gives the sender more options as it is going to change the channel on next send retry; therefore, it has a higher probability of landing on a clear channel. This results in less latency for delivering packets over the multihop network. Quantitatively speaking, the 16-channel

MAC exhibit an average latency that is slightly higher (1.15 times) than that of single-channel MAC in absence of noise. However, the latency exhibited by single-channel MAC under 75% interference increases more than 6 times, while it increases slightly for the 16-channel MAC. The bottom line is that using more multichannels lowers the average latency under interference.

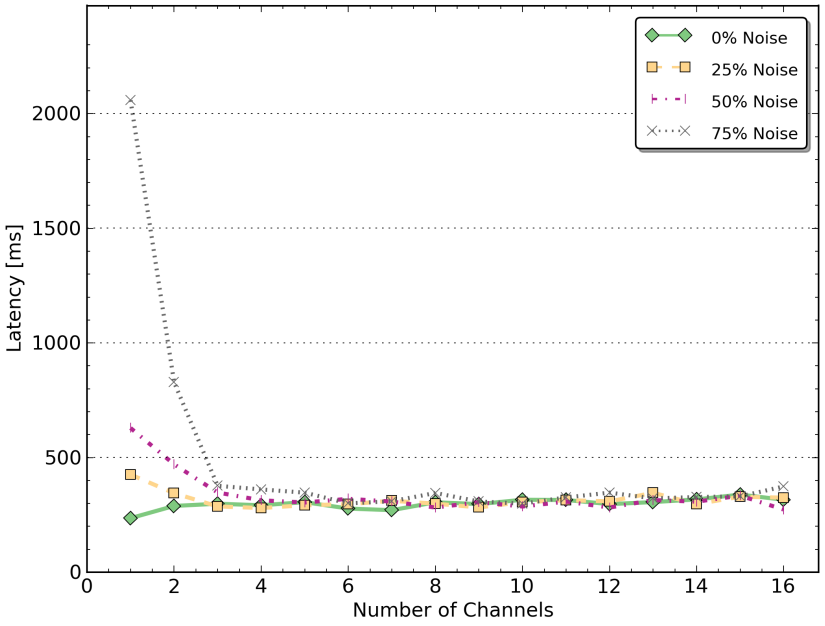


FIGURE 4.7: Latency vs. number of channels and different interference rates on one channel. The results show that using more multichannels lowers the average latency under interference.

Adding a dedicated broadcast channel

We repeat the previous set of experiments but we add a dedicated broadcast channel (channel 11) to the multichannel MAC; thus, increasing the actual number of used channels by one for all cases except for the single-channel MAC which we do not change, and for the 16 channel case, as it is already using the maximum number of available channels, but uses one channel for broadcasts as well. We keep the interferer interfering one unicast channel (channel 24). We vary the number of channels between 1 and 16, and set the interference rate to 0 (no interference), 25%, 50%, and 75%. Each variation is run for one hour each. We should point out that we use the optimization of initiating receiver channel rendezvous on the broadcast channel, where the sender probes the receiver on the broadcast channel when it does not know its channel hop-sequence parameters; thus, potentially lowering the cost for initial rendezvous.

Duty cycle Figure 4.8 shows the average duty cycle of sender nodes in the network. We notice a similar behavior to the settings without using a broadcast channel, except that the duty cycle is lower now, and shows a more stable behavior as it stabilizes around 1.4% to 1.6% depending on interference rate. To further examine the behavior, we pick a middle value for the number of channels; *i.e.* 8 channels, and we compare the duty cycle of sending and receiving when using a broadcast channel and without a broadcast channel. Figure 4.9 shows that using a broadcast channel causes the nodes to spend less time sending, but more time listening in comparison to the case without a broadcast channel. This behavior might improve coexistence with other networks, as the network generates less traffic on the radio channel; thus, making it less congested.

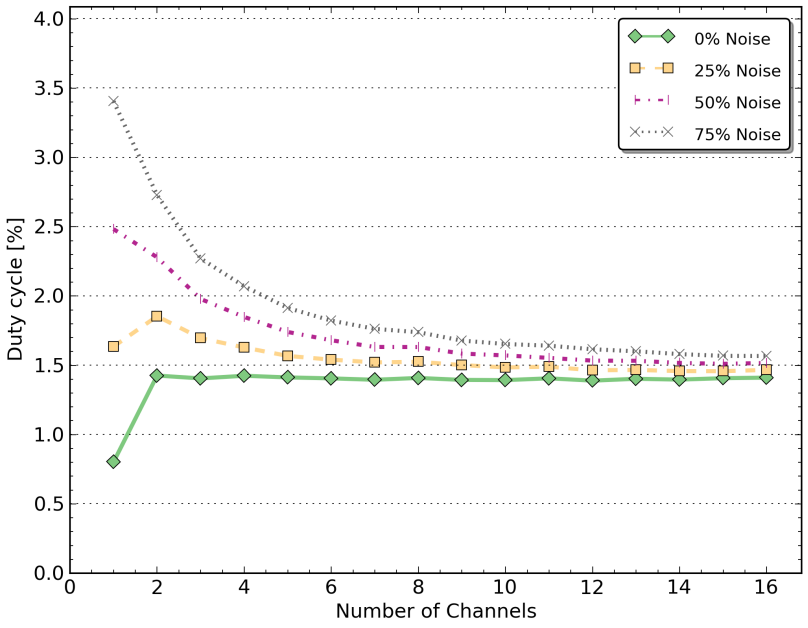


FIGURE 4.8: Duty cycle when using one broadcast channel. Using more channels lowers the duty cycle and offers a more stable behaviour under interference.

Latency Figure 4.10 shows the average latency for delivering a message from a sender node to the sink node in the network. The latency is better than the single channel MAC as in the previous experiment, but it does not improve much in comparison to the multichannel MAC without a broadcast channel. Figure 4.11 shows a bar graph comparing the latency when using 8 channels with vs. without a broadcast channel. It demonstrates that using a broadcast channel reduces the latency as compared to single-channel case and does not add panelty in comparison to the case without a broadcast channel.

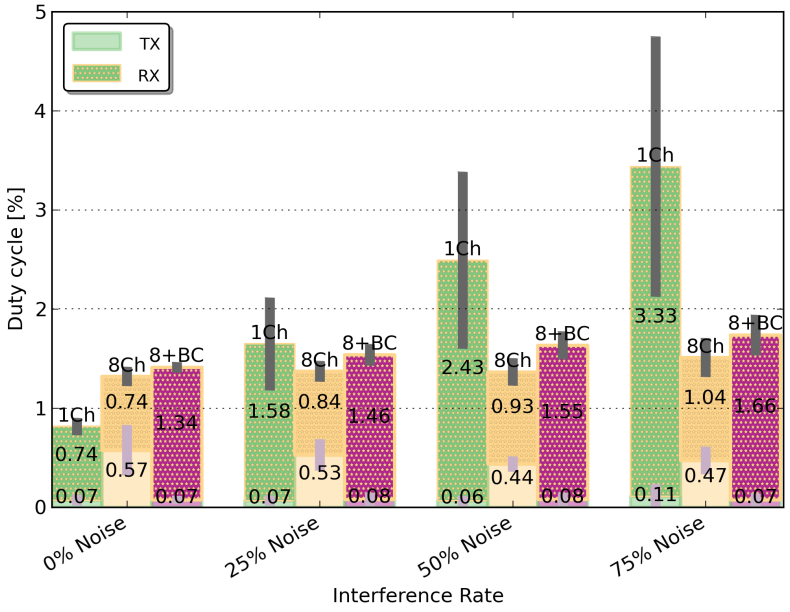


FIGURE 4.9: Duty cycle comparison: Using single channel, 8 channels and 8 channels with a broadcast channel. Using a broadcast channel causes the nodes to spend less time sending, but more time listening in comparison to the case without a broadcast channel.

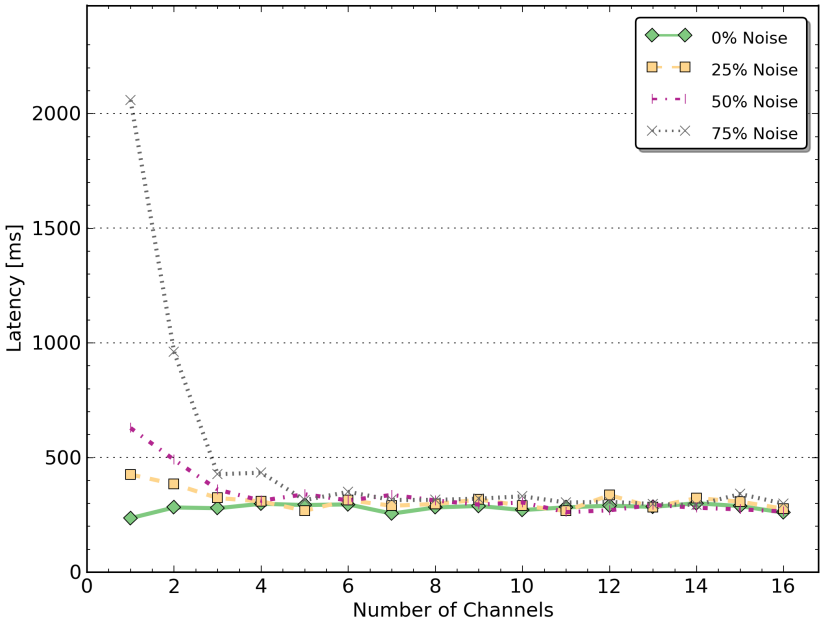


FIGURE 4.10: Latency when using one broadcast channel. It shows that using multi-channels redueces the latency.

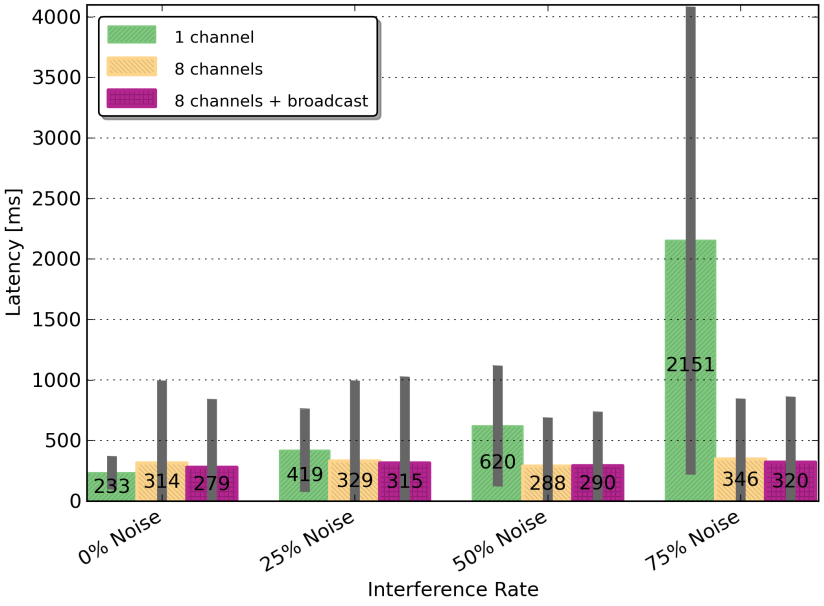


FIGURE 4.11: Latency comparison: Using single channel, 8 channels and 8 channels with a broadcast channel. It shows that using a broadcast channel reduces the latency as compared to single-channel case and does not add panelty in comparison to the case without a broadcast channel.

5 Conclusion and Future Work

In this chapter we conclude the thesis with an articulation and discussion of the presented work, and suggestions for possible future work.

5.1 Conclusion

We have presented the design of a multichannel MAC protocol for Wireless Sensor Networks and the Internet of Things. The protocol we presented is low-power, duty cycled, asynchronous, sender-initiated and frequency-hopping. We designed two flavours of the protocol; one that does not use any dedicated channels, and another which allow the use of dedicated channels for broadcast. Our protocol does not need network-wide synchronization, and uses low-overhead channel-rendezvous mechanism. We implemented the protocol with both flavours in C and evaluated it under RPL and the IPv6 stack of Contiki OS. We showed that it works well under the IPv6 stack, achieves lower duty cycle than single-channel ContikiMAC in the presence of noise, exhibits lower end-to-end latency for the delivery of packets, while it keeps an acceptable base cost in the absence of noise. The flavour that uses a dedicated broadcast channel improves the performance further and reduces the congestion in the network when the number of used channels is more than 9. Overall, our multichannel MAC seems to be a promising alternative to single-channel ContikiMAC enabled by default in Contiki OS. However, the protocol can be improved further by investigating the aspects suggested in the next section.

5.2 Future work

The context of this master thesis was limited to the design of the multichannel MAC; however, several aspects should be studied further or improved in the future work in this field. We might want to consider the quality of the available channels by local white-listing of good channels, to avoid sending on congested or noisy channels. This could be investigated further to have router controlled channel assignments and to come up with better routing metrics for multichannel links. Finally, we might want to look at how to make the protocol compliant with the new standard IEEE 802.15.4e.

Bibliography

[1] Thomas Watteyne, Ankur Mehta, and Kris Pister. Reliability through frequency diversity: why channel hopping makes sense. In *Proceedings of the 6th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, PE-WASUN '09, pages 116–123, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-618-2. doi: 10.1145/1641876.1641898. URL `http://doi.acm.org/10.1145/1641876.1641898`.

[2] Thomas Watteyne, Steven Lanzisera, Ankur Mehta, and Kristofer S. J. Pister. Mitigating multipath fading through channel hopping in wireless sensor networks. In *ICC*, pages 1–5. IEEE, 2010. ISBN 978-1-4244-6402-9.

[3] E. Callaway, P. Gorday, L. Hester, J.A. Gutierrez, M. Naeve, B. Heile, and V. Bahl. Home networking with IEEE 802.15.4: a developing standard for low-rate wireless personal area networks. *IEEE Communications Magazine*, 40(8):70–77, August 2002.

[4] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making Sensor Networks IPv6 Ready. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Raleigh, North Carolina, USA, November 2008.

[5] IEEE. 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks –Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs), September 2011.

[6] António Gonga, Olaf Landsiedel, Pablo Soldati, and Mikael Johansson. Revisiting multi-channel communication to mitigate interference and link dynamics in wireless sensor networks. In *DCOSS*, pages 186–193. IEEE, 2012. ISBN 978-0-7695-4707-7.

[7] B. Azimi-Sadjadi, D. Sexton, P. Liu, and M. Mahony. Interference effect on ieee 802.15. 4 performance. In *Proceedings of 3rd International Conference on Networked Sensing Systems (INNS), Chicago, IL*, 2006.

[8] Carlo Alberto Boano, Thiemo Voigt, Nicolas Tsiftes, Luca Mottola, Kay Römer, and Marco Antonio Zúñiga. Making sensornet mac protocols robust against interference. In *Proceedings of the 7th European conference on Wireless Sensor Networks*, EWSN'10, pages 272–288, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11916-6, 978-3-642-11916-3. doi: 10.1007/978-3-642-11917-0_18. URL `http://dx.doi.org/10.1007/978-3-642-11917-0_18`.

[9] G. Mulligan, N. Kushalnagar, and G. Montenegro. IPv6 over IEEE 802.15.4 BOF (6lowplan). Web page. URL `http://www.ietf.org/ietf/04nov/6lowpan.txt`. Visited 2005-02-21.

[10] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, March 2012. ISSN 2070-1721. URL `http://www.ietf.org/rfc/rfc6550.txt`.

[11] Lei Tang, Yanjun Sun, Omer Gurewitz, and David B. Johnson. Em-mac: a dynamic multichannel energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the Twelfth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '11, pages 23:1–23:11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0722-2. doi: 10.1145/2107502.2107533. URL `http://doi.acm.org/10.1145/2107502.2107533`.

[12] Youngmin Kim, Hyojeong Shin, and Hojung Cha. Y-mac: An energy-efficient multi-channel mac protocol for dense wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 53–63, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3157-1. doi: http://dx.doi.org/10.1109/IPSN.2008.27. URL `http://dx.doi.org/10.1109/IPSN.2008.27`.

[13] Joris Borms, Kris Steenhaut, and Bart Lemmens. Low-overhead dynamic multi-channel mac for wireless sensor networks. In *EWSN*, pages 81–96, 2010.

[14] O. Durmaz Incel, P.G. Jansen, and S.J. Mullender. Mc-lmac: A multi-channel mac protocol for wireless sensor networks. Technical Report TR-CTIT-08-61, Centre for Telematics and Information Technology University of Twente, Enschede, The Netherlands, 2008.

[15] Venkat Iyer, Matthias Woehrle, and Koen Langendoen. Chrysso - a multi-channel approach to mitigate external interference. In *SECON*, pages 449–457. IEEE, 2011. ISBN 978-1-4577-0094-1. URL `http://dblp.uni-trier.de/db/conf/secon/secon2011.html#IyerWL11`.

[16] Adam Dunkels. The ContikiMAC Radio Duty Cycling Protocol. Technical Report T2011:13, Swedish Institute of Computer Science, December 2011. URL `http://www.sics.se/~adam/dunkels11contikimac.pdf`.

[17] E. White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, 2011. ISBN 9781449320584. URL `http://books.google.se/books?id=VCOTy1xWZmQC`.

[18] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. ISBN 0123751659, 9780123751652.

[19] S. Duquennoy, F. Österlind, and A. Dunkels. Lossy Links, Low Power, High Throughput. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Seattle, WA, USA, November 2011.

[20] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Boulder, Colorado, USA, 2006.

[21] D. Moss and P. Levis. BoX-MACs: Exploiting Physical and Link Layer Boundaries in Low-Power Networking. Technical Report SING-08-00, Stanford University, 2008.

[22] A. El-Hoiydi, J.-D. Decotignie, C. C. Enz, and E. Le Roux. wiseMAC, an ultra low power MAC protocol for the wiseNET wireless sensor network. In *ACM SenSys*, 2003.

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

[24] RIPE NCC. IPv4 Exhaustion, September 2012. http://www.ripe.net/internet-coordination/ipv4-exhaustion.

[25] G. Mulligan. The 6lowpan architecture. In *Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (IEEE Emnets)*, pages 78–82, Cork, Ireland, 2007. ACM. ISBN 978-1-59593-694-3. doi: http://doi.acm.org/10.1145/1278972.1278992.

[26] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Berkeley, CA, USA, 2009.

[27] O. Gnawali, P. Levis, R. Fonseca, K. Jamieson, and D. Moss. CTP: Collection Tree Protocol. Web page: http://sing.stanford.edu/gnawali/ctp/. URL `http://sing.stanford.edu/gnawali/ctp/`. Visited 2012-10-30.

[28] J-P Vasseur et al. RPL: The IP routing protocol designed for low power and lossy networks, April 2011. IPSO Alliance White Paper 7, available from www.ipso-alliance.org.

[29] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI*.

[30] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (IEEE Emnets)*, Tampa, Florida, USA, November 2004.

[31] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (IEEE Emnets)*, Cork, Ireland, June 2007.

[32] A. Dunkels. The Contiki Operating System. Web page. URL `http://www.contiki-os.org/`. Visited 2012-10-16.

[33] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes. Powertrace: Network-level power profiling for low-power wireless networks. Technical Report T2011:05, Swedish Institute of Computer Science, March 2011.

[34] Chipcon AS. CC2420 Datasheet (rev. 1.3), 2005. URL `http://www.chipcon.com/`.

[35] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2.

[36] Tmote sky: Ultra low power IEEE 802.15.4 compliant wireless sensor module.

[37] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.

[38] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *International Workshop on Practical Issues in Building Sensor Network Applications*, Tampa, Florida, USA, November 2006.