University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Mateusz Banaszek
Student no. 346851

An Implementation and Evaluation of a Receiver-Initiated MAC Protocol for Dependable Low-Power Wireless Networks

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Konrad Iwanicki
Institute of Informatics

August 2018

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date | Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date | Author's signature

Abstract

The HENI project aims to develop a scalable and practical routing protocol for low-power wireless sensor networks. To a large extent, the performance of such a protocol, notably its energy consumption, latency, and reliability, is influenced by the underlying medium access control (MAC) protocol. The goal of this thesis is to implement a receiver-initiated MAC protocol which allows for reliably exchanging messages within a moderately congested network. Important requirements for the newly created solution, named CherryRiMAC, are short radio activity time and low power consumption of the device. To achieve these goals, CherryRiMAC combines the original receiver-initiated approach with already existing ideas from other solutions. Since the implementation of the new MAC protocol is targeted for the CC2650 chip, the thesis evaluates also the usefulness of the hardware features facilitating wireless communication, which are provided by the chip and other modern low-power radios. To prove the reliability not only in the design of CherryRiMAC but also the quality of its implementation and to demonstrate that the assumed goals are indeed achieved, the protocol is tested on actual target hardware in a number of configurations emulating a variety of usage scenarios. Finally, the performance of CherryRiMAC is accessed in comparison with an already existing implementation of the X-MAC protocol.

This work was conducted within the HENI project, which was supported by the National Center for Research and Development (NCBR) in Poland under grant no. LIDER/434/L-6/14/NCBR/2015 within the LIDER-VI program.

Keywords

CherryRiMAC, RI-MAC, receiver-initiated, wireless MAC protocol, low-power wireless networks, wireless sensor networks, whip6, CC2650

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics

Subject classification

- Networks∼Network protocol design
- Networks∼Link-layer protocols
- Networks∼Network experimentation

Tytuł pracy w języku polskim

Implementacja i ewaluacja protokołu MAC inicjowanego przez odbiorcę dla niezawodnych niskomocowych sieci bezprzewodowych

Contents

Acknowledgments

I would like to thank my supervisor, Konrad Iwanicki, for his support (both technical and non-technical) I received when writing this thesis.

Special thanks also to Szymon Acedański, who helped me a lot understand how the radio works, Przemysław Gummienny, for his valuable feedback on early versions of CherryRiMAC, and other HENI project members who created the platform, infrastructure and tools facilitating software creation and evaluation.

Glossary

A list of basic terms used in this thesis (in alphabetical order):

ack beacon
A special-purpose frame used by a MAC protocol acknowledging a transmission.

Always Listen mode
A special mode of CherryRiMAC in which a node is able to receive a frame during its whole cycle.

base beacon
A special-purpose frame used by a MAC protocol.

broadcast base beacon
A type of the base beacon used by CherryRiMAC to initiate transmission of a broadcast frame.

broadcast transmission
A transmission of a message from one node to all its neighbors.

CC2650
A Texas Instruments CC2650 SimpleLink ultra-low power wireless microcontroller [1].

CherryMote
A low-power wireless device created within the HENI project, equipped with the CC2650 radio chip.

CherryRiMAC
A new MAC protocol described in this thesis.

data frame
A frame with data, formatted according to the IEEE 802.15.4 specification [2].

HENI
The project that this thesis is conducted within.

neighbor
A node that is within a radio range of another node.

Neighbors List
A module/component that stores information about node's neighbors (incl. the timestamp of the last base beacon and cycle length).

Neighbors Scan
A special mode of CherryRiMAC which is used to discover the neighbors of a node.

nesC
A programming language [3] [4] used within the whip6 operating system.

node
A device that is a part of a network.

RI-MAC
An original Receiver-Initiated MAC protocol developed by Yanjun Sun et al. [5], on which CherryRiMAC is based.

scan base beacon
A type of the base beacon used by CherryRiMAC during Neighbors Scan.

standard base beacon
A type of the base beacon used by CherryRiMAC to initiate transmission.

unicast transmission
A transmission of a message from one node to another one.

whip6
An operating system for low-power wireless sensor devices [6].

Chapter 1

Introduction

The phrase *Internet of Things* (*IoT*), nowadays so popular in both industry and academia, covers a wide range of devices that aim to connect the physical word with the digital one. Such devices usually collect data about the environment they are embedded in by means of sensors and can also interact with their surroundings by triggering different types of actuators.

1.1. Low-power wireless devices

This thesis concentrates on a class of IoT devices that form so-called *low-power wireless networks*. A good example is a system for monitoring temperature in a green house [7]. The main part of such a system is a fleet of small devices (the size of a matchbox) consisting of a microcontroller, a radio, a sensor (i.e., a thermometer) and a battery. The task of the devices is to periodically measure temperature and send the readings to a server (usually, through a special gateway device), where they are further processed. Being small, low cost and fully wireless (i.e., in therms of both communication and power supply), the devices are easily deployable in large quantities, even in locations with limited access. However, this means that to be usable, they should require little or even no maintenance, especially battery charging or replacement. In fact, minimizing power consumption is arguably the greatest challenge when designing such devices: they should use minimal amounts of energy, thereby being able to operate for years with small batteries as their only power supply.

To achieve this, the designs of such devices are based on energy-efficient microcontrollers that offer limited resources (i.e., memory and CPU power) and run software operating on an event-only basis (i.e., inactive most of the time). However, the main sources of energy consumption on the devices are wireless radios. For example, a Texas Instruments CC2650 SimpleLink ultra-low power wireless microcontroller [1] requires below 0.003 mA when it sleeps, about 0.55 mA when it is idle, 1.5 mA when it performs some calculations and about 6 mA when it uses its radio [8]. Therefore, to reduce power consumption, one has to minimize the time when the radio is active. This means that appropriate radio management is a crucial part of networking software for low-power wireless devices.

1.2. Problem statement

This requirement for managing radio duty cycle leads to the need for specially designed medium access control (MAC) protocols. The main task of a MAC protocol is to manage a device's access to the medium on which communication is performed. Widely known MAC protocols are presented in standards: IEEE 802.3 for wired Ethernet networks [9] and IEEE

802.11 for wireless Wi-Fi networks [10]. They describe, among others, how a transmission looks like, when a transmission can take place and how to handle a situation when many devices want to transmit at the same moment.

A low-power wireless device commanded by an ideal MAC protocol should be able to handle a high traffic in a network and consume very little energy at the same time. Moreover, it should offer an optimal performance when used both in a small deployment in which few messages are exchanged during long intervals and in a highly congested network in which a high throughput is constantly expected. Furthermore, transmissions of frames should be performed with a low latency in a reliable way, not favoring any of the recipients. However, most of these requirements are contradictory. Therefore, many different ideas have been proposed in search of the balance between them.

The HENI project [11], within which this thesis has been written, aims to develop a new scalable and practical routing protocol for the Internet of Things. To prove its properties, the protocol is planned to be tested in a real-world environment. To this end, an evaluation network of custom-designed devices named *CherryMotes* is being built. However, the performance of the routing algorithm, notably its energy consumption, latency, and reliability, depends indirectly on the MAC protocol managing the wireless communication. Therefore, a proper solution is needed which will provide the expected performance, especially of the aforementioned aspects, when deployed in a moderately congested network.

Particularly promising is exploring an idea of receiver-initiated communication, which has been incorporated into the RI-MAC protocol, created by Yanjun Sun et al. [5]. This seemingly counter-intuitive approach, according to its authors, offers reliable communication that is efficient both in term of throughput and energy consumption. It also utilizes the medium in a way that increases the capacity of the whole network, compared to previously existing solutions. At the same time, the simple design of the protocol should result in a relatively straightforward implementation which will be easily expandable to provide new functionality.

1.3. Contribution

The aim of this thesis is to design, implement and evaluate an efficient receiver-initiated MAC protocol that could be used for the HENI's CherryMote network. The contributions of this thesis are threefold.

First, the thesis introduces a new MAC protocol for low-power wireless devices: **CherryRiMAC**. It is a duty-cycling receiver-initiated protocol based on RI-MAC. However, it differs from RI-MAC in a number of features: its design is modified to accomplish a lower power consumption, facilitate functions required by some higher-level network protocols and increase performance on the edge of a network. These enhancements are achieved by incorporating already existing ideas from other protocols (or their actual implementations) like phase awareness, best-effort broadcast transmissions and an always listen mode. At the same time, all benefits of the receiver-initiated communication and simplicity of the original design of RI-MAC are preserved.

Second, the thesis presents an implementation of CherryRiMAC. It is prepared for CherryMote devices equipped with the CC2650 chip running the whip6 [6] operating system. This hardware-software platform provides simple interfaces for controlling the radio: powering it up and down, configuring, sending a command and receiving interrupts. The implementation of CherryRiMAC consists of three main components: commanding the radio, executing logic of the newly created protocol and managing outgoing frames and buffers for incoming frames.

Furthermore, it provides flexibly designed interfaces that enable a higher-level network protocol to use features of CherryRiMAC. The implementation also includes some additional exemplary components that allow for fast creation of simple applications that require radio communication.

As the final contribution, the thesis preliminarily evaluates CherryRiMAC on actual hardware. The main goal of the evaluation is to assess the correctness and performance of CherryRiMAC in a variety of possible deployments. Different test setups indicate that the new protocol handles message exchanges in a reliable way under moderate traffic in a network. When the medium is highly congested, the delivery ratio is lower but still acceptable. The evaluation confirms also that introducing into CherryRiMAC concepts like the phase awareness allows for significantly reducing radio activity time, especially compared to X-MAC [12]. Finally, all performed tests indicate that the implementation of CherryRiMAC reliably signals transmission outcomes.

1.4. Thesis organization

The rest of this thesis is organized in the following way. Chapter 2 presents the original RI-MAC protocol. Chapter 3 describes the new CherryRiMAC protocol, stressing similarities and differences compared to RI-MAC. Chapter 4 discusses the actual implementation of CherryRiMAC in whip6 for the CC2650 chip. Chapter 5 elaborates on the performed preliminary evaluations of the implementation. Finally, Chapter 6 concludes and proposes future improvements to the protocol.

This work was conducted within the HENI project, which was supported by the National Center for Research and Development (NCBR) in Poland under grant no. LIDER/434/L-6/14/NCBR/2015 within the LIDER-VI program.

Chapter 2

RI-MAC

This chapter presents the original RI-MAC protocol created by Yanjun Sun et al. [5]. The description provides some background information and focuses on parts of the protocol that are necessary to later understand the design of CherryRiMAC.

2.1. Background information

A MAC protocol is usually the lowest software layer in a network stack of a device (see Figure 2.1). Its role is to manage message exchanges with other devices through a medium. However, in most cases it does not initiate communication by itself, but is only responsible for delivering packets generated by higher network layers to designated recipients. It also does not analyze content of received messages itself, but passes these packets back to the higher network layer which requested the receiving.

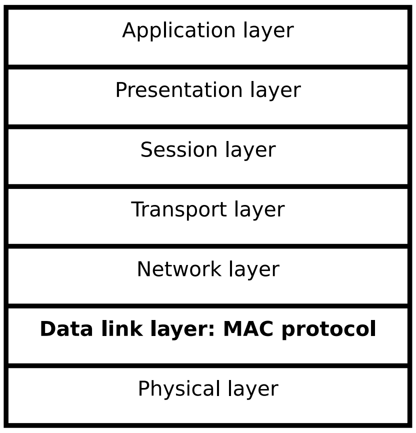


Figure 2.1: A typical network stack

Although there are many MAC protocols for wireless networks available for different classes of devices, most of the already existing solutions are not applicable to low-power wireless devices: they assume no or too little limitations of radio usage. Even standards like IEEE 802.15.4 [2], which were designed with limited resources in mind, are considered to be too complex and too energy-hungry for such applications. Therefore, many new ideas have been considered for low-power wireless devices [13, 14, 15, 16].

Such protocols generally work in a cycle, activating the node's radio (in the context of a network, devices are called *nodes*) only for a short period of time when a transmission is expected, and deactivating it for the rest of the cycle. Existing solutions can be roughly divided into two main categories: synchronous and asynchronous. In a synchronous approach nodes coordinate their radio-on/radio-off (called *active* and *sleep* periods) cycles with other nodes that are within radio range (these devices are called *neighbors* of the given node). As a result, they know exactly when to start listening or transmitting, which greatly reduces surplus radio activity. However, the synchronization can be a complex process, requiring additional transmissions and is hard to manage when, for example, two neighbors work on different schedules. S-MAC [17], T-MAC [18], RMAC [19] and DW-MAC [20] represent such solutions. The other group, asynchronous MACs represented by Aloha with Preamble Sampling [21], B-MAC [22] and WiseMAC [23] implements an opposite idea: each node works according to its own cycle, independently of its neighbors. Because of the lack of cycle synchronization, to perform a transmission the sender needs to "catch" a receiver, which is usually done by sending a preamble that lasts longer than a sleep period of the other device, which implies extra radio usage. However, they are said to perform reasonably well under light traffic and are less complex in design.

MAC protocols differ also in functionality they provide. Practically all of them are able to perform a *unicast* transmission (i.e., one node sends a message to another node), whereas only some protocols implement *broadcast* transmissions (i.e., one nodes sends a message to all its neighbors simultaneously). For example, the Crankshaft [24] protocol has broadcast transmissions incorporated into its design, WiseMAC has been enhanced with an energy-efficient broadcasting scheme [25], whereas RI-MAC [5] concentrates only on unicast transmissions.

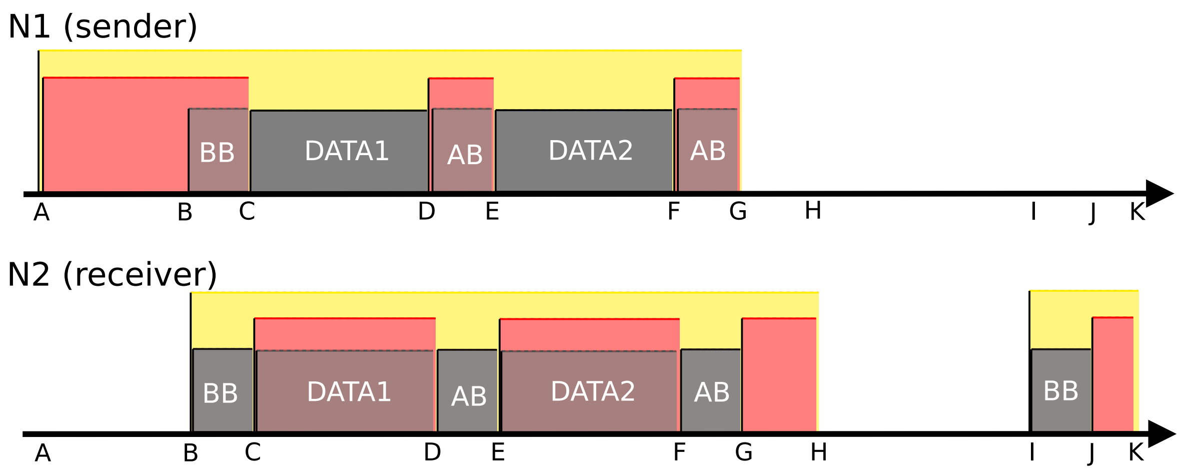
Diverse are also guarantees which a MAC protocol provides. As an example, in the original design of X-MAC [12], a sender does not assuredly know whether its frame has been successfully delivered. However, a variation of this protocol implemented in a UPMA package [26] for TinyOS [27] incorporates an idea of acknowledgments send by a receiver when it successfully gets a frame intended for it.

2.2. Overview of RI-MAC

Receiver-Initiated MAC (RI-MAC) is an asynchronous protocol that implements an interesting idea: let the receiver initiate a transmission. The sender, instead of sending a preamble that can be detected by a receiver when it wakes up, listens for a special frame, called a *beacon*, that is sent by the receiver when it is ready to receive a data frame. This approach, according to its authors, offers reliable communication that is efficient both in term of throughput and energy consumption.

To receive a data frame, each node wakes up according to its cycle. After activating its radio, it transmits a special frame, a base beacon, and listens for incoming frames. Therefore, a node that wants to send a data frame needs to power up its radio and start listening. When it receives the beacon from the intended receiver, it replies with the data frame. The receiver, after receiving the data frame, transmits an ack beacon which acknowledges the reception and additionally announces readiness to receive another data frame. When a receiver does not receive any frame in a specified time after transmitting its beacon, it goes to sleep and wakes up again in the next cycle. An interval between the previous and the next base beacon is calculated as a preset value randomized each time in a 50%-150% range to avoid coincidental synchronization of nodes. Figure 2.2 represents an exemplary transmission.

For the sender, represented in the figure by node N1, the process of sending a data frame starts at moment A when it activates its radio and starts listening. The receiver (node N2 in the figure) wakes up, according to its schedule, at moment B and transmits a base beacon.



BB – base beacon, AB – ack beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

Figure 2.2: RI-MAC: Overview

The sender receives the beacon and, as it has come from the expected node, transmits the data frame (moment C) and starts listening again (moment D). The receiver receives the frame, ensures that it is the intended recipient of the frame and transmits an ack beacon (moment D). When the sender receives this ack beacon, it is informed that the transmission was successful. In the example in the figure the sender has another data frame intended for the receiver, so it transmits the frame in reply to the ack beacon (moment E). As before, the sender verifies the received data frame, transmits a new ack beacon (moment F) and continues listening. When the sender successfully receives the ack beacon and does not have more frames intended for the receiver, it deactivates its radio and goes to sleep (moment G). On the receiver's side, when a specified time has passed without receiving any frame, the receiver goes to sleep too (moment H). It wakes up again in its next cycle (moment I), transmits a base beacon and starts listening for incoming data frames (moment J). As this time in the figure it does not receive any frames, it ends listening and goes to sleep again (moment K).

Since each node can both send and receive data frames, sending and receiving modes are being interlaced when a device works.

2.3. Beacon frames

In the original design of RI-MAC, a beacon frame (of both the base and the ack type) is of a reserved IEEE 802.15.4 frame type containing all necessary standard fields [2]: Hardware Preamble, Frame Length, Frame Control Field and Frame Check Sequence used in compliance with the standard. Additionally some RI-MAC-specific fields are used.

A base beacon contains also a 16-bit Source Address field – probably (as it is not explicitly named) a short IEEE 802.15.4 address of a node that sends it.

An ack beacon contains a Source Address and also a Destination Address field (16-bit) – an address of the node that has sent a data frame which reception is being acknowledged. The role of the ack beacon is twofold: it is an invitation for the node's neighbors to send another data frame (the same way as a base beacon is) and, additionally, the sender of the data frame gets an acknowledgment that the transmission has been successful.

A beacon frame can also contain a 1-byte field containing the current size of a backoff window. The purpose of this field is described in Section 2.4.

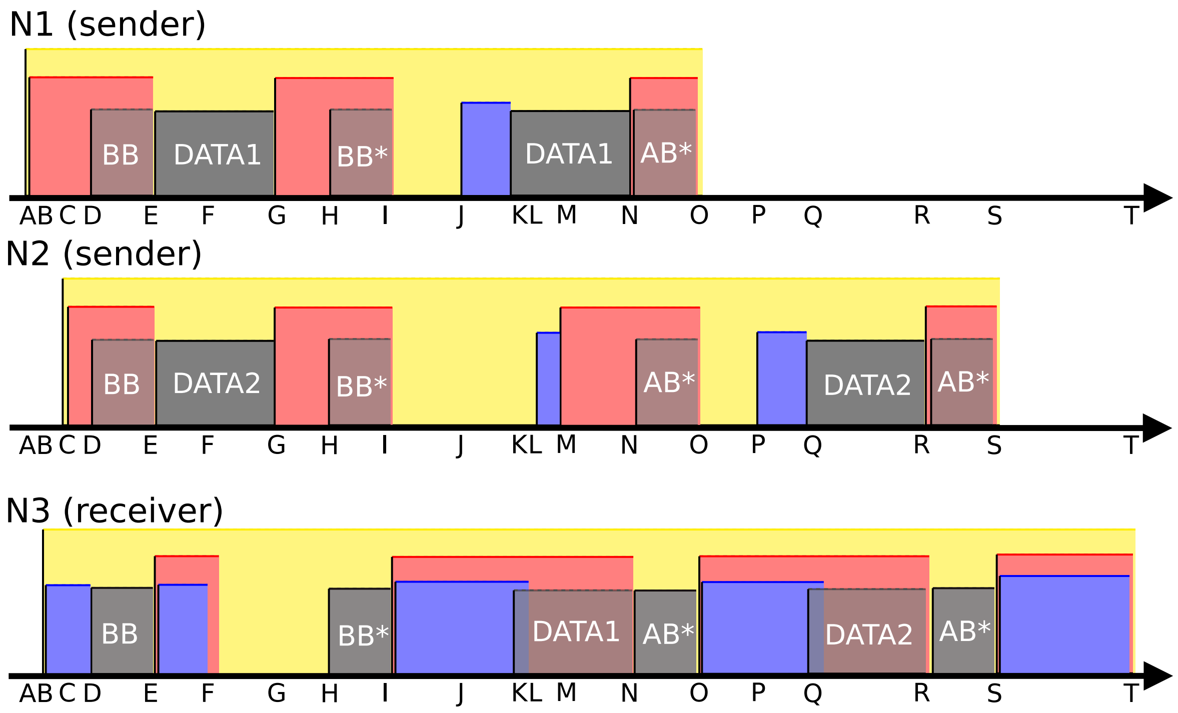
A node can distinguish between different beacon types by analyzing the length of a received frame.

2.4. Contending nodes

The example from Section 2.2 presents an optimistic scenario, in which there is only one sender and one receiver and no collisions of frames occur. However, the RI-MAC protocol includes mechanisms for dealing with situations in which nodes contend for the medium.

When a receiver wakes up, before sending its base beacon it checks whether the medium is free using *clear channel assessment* (CCA). If it is not, the node postpones an attempt to send its base beacon. If the medium is clear, the node transmits the base beacon without a backoff window size (BW for short) field. It is information for potential senders that they should transmit their data frames immediately. However, when more senders transmit their data frames simultaneously, a collision may occur. As a result, the receiver does not receive a valid data frame, but the CCA checks being performed while listening indicate the medium activity. Therefore, after waiting for the duration of the longest possible data frame transmission, the receiver transmits a new beacon but, this time, with a BW field set. Its value is calculated with, for example, a binary exponential backoff strategy. When senders receive the beacon with the BW field set, before transmitting their data frames they should wait for a random time (calculated based on the received BW value) and then check if the medium is free using CCA. This waiting duration cannot be shorter than the time required to generate and start a transmission of an ack beacon to avoid collisions with the acknowledgment if another sender has had a shorter waiting time and has already sent its data frame. If a collision of data frames happens again, the receiver transmits the next beacon with a more increased BW value. This approach allows RI-MAC to actively adapt to the level of current medium congestion: the more collisions occur, the longer BW is to prevent them. When collisions keep occurring and BW reaches its maximum value, the receiver abandons further attempts and goes to sleep. Similarly, a sender keeps retry counters and if it does not receive a base beacon when listening for a duration equal to three times the sleeping period, it abandons further attempts to send this data frame. The operation is canceled as well if the node does not receive an ack beacon when listening for a duration of the maximum BW size after transmitting its data frame. The BW value from the last beacon is also used by a receiver to conclude that there are no data frames intended for it: it listens for at least the duration of the current BW.

Figure 2.3 presents with details an exemplary transmission with contending senders. Two senders (nodes N1 and N2) want to send their data frames to the receiver (node N3). When the receiver wakes up (moment B) and senses that the medium is clear, it transmits a base beacon (without the BW field) (moment D) and starts listening and performing CCA checks (moment E). The beacon is received by both senders, so they both reply immediately (as there is no BW field in the beacon) with their data frames (moment E). Simultaneous transmissions lead to a collision, so the receiver is not able to detect Start of Frame Delimiter (SFD) in an expected time, but CCA checks indicate a medium activity (moments E–F). Therefore, the receiver concludes that a collision has occurred and thus stops CCA checks (moment F), waits a duration of the longest possible data frame transmission (moments F–H) and sends a new beacon with the BW field set to a non-zero value (moment H). Both senders receive the beacon and start their backoff time of a randomized duration based on the received BW value (moments I–J for node N1 and I–L for node N2). As sender N1 senses that the medium is clear (moments J-K) it sends its data frame. It is successfully received by the receiver



N1 (sender)
BB
DATA1
BB*
DATA1
AB*
A B C D E F G H I J K L M N O P Q R S T
N2 (sender)
BB
DATA2
BB*
AB*
DATA2
AB*
A B C D E F G H I J K L M N O P Q R S T
N3 (receiver)
BB
BB*
DATA1
AB*
DATA2
AB*
A B C D E F G H I J K L M N O P Q R S T

BB – base beacon, AB – ack beacon, * – beacon includes BW field, DATA – data frame
yellow – node is active, red – listening, gray – transmitting, blue – CCA checks

Figure 2.3: RI-MAC: Contending senders

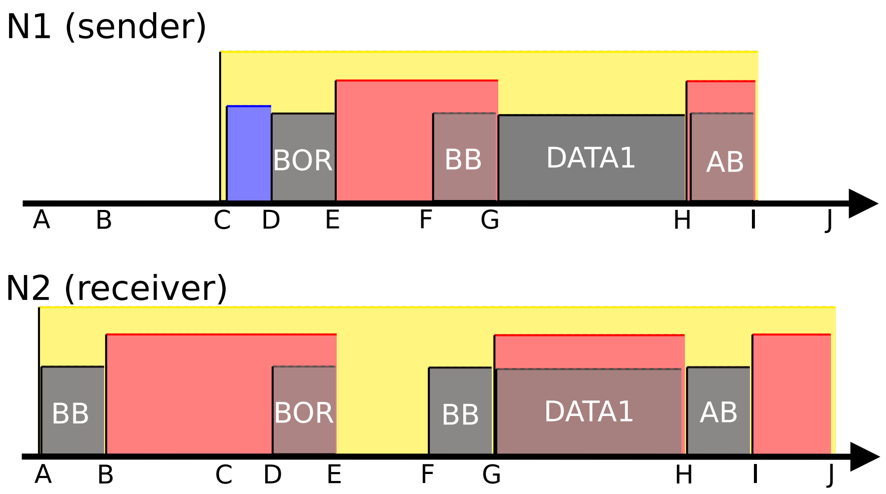
which replies with an ack beacon to this sender (moment N). Note, that a BW field in this beacon is still set to the current BW value. Sender N2, which ends its backoff time at the moment L, checks that the medium is occupied (moments L–M) so it ends this attempt to send the data frame (with failure) and starts the process again beginning with listening for a beacon (moment M). At a moment N it receives the ack beacon (that has been sent to sender N1) with the BW field set, so it performs backoff waiting (moments O–P), verifies that the medium is clear (moments P–Q) and starts transmission of its data frame (moment Q). This frame is successfully received by the receiver which replies with a new ack beacon (moment R) and starts listening for further data frames (moment S). Because it detects no SFD in a time long enough for a potential sender to end a backoff period and CCA checks indicate no medium activity, the receiver concludes that there are no more data frames intended for it and goes to sleep (moment T).

Surprisingly, as RI-MAC authors note, their experiments on physical devices show that a concurrent transmissions of data frames, as a reply for a beacon, does not necessarily lead to a collision because of a so-called capture effect [28]. The capture effect is, simplifying, a phenomenon in which despite a theoretical possibility of a collision, one frame is successfully received without any harmful interference from other frames. The capture effect occurs especially where distances (thus signal strengths) between a receiver and senders differ significantly.

2.5. Beacon-on-Request

To start a transmission a sender has to wake up and passively listen for a beacon from an intended receiver. However, when the receiver has already sent a beacon and now is listening for an incoming data frame, the sender will not notice it and will listen the whole receiver's cycle until the next base beacon is sent. As a result, there is a lot of excessive idle listening, although the transmission could be performed.

To solve this problem, RI-MAC includes an optimization called *beacon-on-request*. When a sender wakes up, it transmits a beacon with a Destination Address field set to an address of the intended receiver, verifying beforehand that the medium is clear. If the receiver is already active, it receives it, waits at least for a duration of a BW value from the beacon and transmits its base beacon. Now, the sender replies to it with the data frame, as always. This way, the transmission happens in the same cycle as the sender woke up. Figure 2.4 represents an exemplary transmission with the beacon-on-request feature. The receiver (node



BB – base beacon, AB – ack beacon, BOR – beacon-on-request beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting, blue – CCA checks

Figure 2.4: RI-MAC: Beacon-on-request

N2) transmits its base beacon at a moment A and starts listening. Therefore, the sender (node N1) which wakes up later (moment C) is not aware of that. Is performs a CCA check (moments C–D) and, as the medium is clear, starts transmission of a beacon-on-request with Destination Address field set to the receiver's address (moment D). When the receiver receives it, it waits longer than a duration of a BW from the beacon-on-request (moments E–F) and starts transmitting its base beacon (moment F). It is received by the sender which replies with its data frame (moment G) and the transmission continues normally.

It may also happen that the receiver is inactive when the sender transmits a beacon-on-request. Such a situation is not directly commented in the original description of RI-MAC, but then, most likely, the sender simply listens for a base beacon transmitted by the receiver in its next cycle as would happen without this optimization.

It seems to me that such a beacon-on-request can be misinterpreted by a third device as an ack beacon from a now-sender to a now-receiver. Therefore, after executing its backoff it can transmit a data frame intended for the now-sender. Perhaps, a collision is avoided by the randomization of the waiting time and CCA checks, as presented in Section 2.4. However, the original RI-MAC description does not discuss such a situation.

2.6. Lacking information

RI-MAC is presented by its authors in a very detailed and precise way, as far as the subject of a transmission is concerned. They present also how to exactly calculate all waiting and listening periods which were omitted or simplified in the previous descriptions. However, when designing my CherryRiMAC I found out that implementing a working solution requires some additional information which is not present in the original description of RI-MAC and it is not clear how the authors of RI-MAC have solved these problems as their actual implementation is not publicly available.

First of all, because the process of sending and the process of receiving a data frame differ significantly from the perspective of a node, only one of these modes can be executed at any moment. Thereby, a device needs to decide somehow if the next action is sending or receiving. However, the process of decision making is not described and I do not see a straightforward answer to it. If these two modes are not balanced properly, starvation of one mode can occur or, at least, communication performance can be worsened: a higher latency and power consumption may result. A simple tactic of interlacing one by one will not work: if the other device does the same in phase, both will be receiving, then both will be sending, and so on. The decision what to do next can also be made at the network layer that is implemented above RI-MAC (although the question remains), but at that layer the information when the next base beacon is planned is not available, so any possibility of optimized scheduling (leading to a higher performance) is forfeited.

A similar decision can (should) be made if two or more data frames to different recipients are already prepared for transmissions: which one should be served first? Sending them in the same order as they have been prepared is a fair algorithm. However, it is not the best from the performance point of view, as it may happen that a beacon that arrives as the first one is from the node that the second data frame in the queue is intended for. Moreover, a transmission of this data frame may be finished successfully before a beacon from the other recipient arrives. In such a situation, a tactic ”which beacon arrives earlier” greatly reduces an overall latency. On the other hand, there is a pitfall when the second (in time) receiver is in a ”shadow” of the first one: it wants to send its beacon when the previous transmission is still in progress so it has to postpone its attempt. Because in RI-MAC in such a situation a node performs a backoff waiting and a cycle length is randomized each time, it may not lead to a starvation (a situation in which this receiver is always in the ”shadow” of the other one) in practice, but it may result in extra congestion, causing a lower performance.

What is more, RI-MAC is concentrated on unicast transmissions and does not include an ability to broadcast a frame. However, this feature is required by some higher-level protocols (e.g. Trickle [29], RPL [30], Deluge [31]). The authors of RI-MAC state that the broadcast functionality can be easily added by sending a frame in unicast transmissions to every neighbor or by repeatedly transmitting the frame for at least the duration of a sleep interval or by an approach combining both of these methods. When implementing broadcast transmissions in the CherryRiMAC protocol, it turned out that adding this functionality induces some additional fundamental decisions and changes to the protocol (see Section 3.5).

Finally, RI-MAC-specific beacons include one 16-bit field for a source address and one 16-bit field for a destination addresses. Yet, according to the IEEE 802.15.4 standard, each device has an extended address (64-bit, EUI-64) which is globally unique or a short address (16-bit) which can be associated with a device and should be unique in a personal area network (PAN). The short address is not guaranteed to be unique across different networks, whereas RI-MAC beacon does not include a PAN Identifier (PAN ID) field. This leads to a question how the implementation will behave in a deployment where there is another network (perhaps IEEE

802.15.4 compliant) which uses the same 16-bit addresses to distinguish its nodes. Moreover, IEEE 802.15.4 short addresses are usually associated with devices in a process that already requires radio communication.

The original RI-MAC description is organized around individual aspects of a transmission, so an overall all-encompassing node's algorithm has to be reconstructed by the reader. For this reason, implementing RI-MAC requires an in-depth analysis of all described features and combing them into a consistent list of steps. The main aim of RI-MAC authors was to present a new approach, a receiver-initiated MAC protocol, and they have achieved it. However, I strongly believe that answers to the aforementioned questions are required to create a working implementation that is usable in real-world deployments.

2.7. (Non-)existing implementations

A simple survey on the two most popular operating systems for low-power wireless networks, Contiki OS [32] and TinyOS [27] (according to Oliver Hahm et al. [33]), indicates that the idea of a receiver-initiated communication is not used in actual implementations. Contiki OS offers ContikiMAC [34], a modified X-MAC (CX-MAC) [35] and TSCH [36]. TinyOS provides BoX-MAC [37]. Additionally, the main operating system used within the HENI project: whip6 [6], which is similar to (but not compatible with) TinyOS, has only a version of X-MAC.

To the best of my knowledge, there is no publicly available usable implementation of a RI-MAC-like protocol at all. Although the idea behind RI-MAC has recently been described in IEEE 802.15.4 [2, chapter 6.12.3 RIT], the only implementation of this approach that I was able to find is Contiki's Low-Power Probing (LPP). What is more, this implementation was removed from the repository in 2013 [38].

Chapter 3

CherryRiMAC

This chapter discusses CherryRiMAC: the new MAC protocol for low-power wireless networks based on RI-MAC (described in Chapter 2). It can be classified as a hybrid asynchronous-synchronous protocol since each node works according to is own schedule, but, at the same time, devices learn schedules of their neighbors.

Like RI-MAC, CherryRiMAC implements the principle of receiver-initiated transmissions in a similar duty-cycled fashion. It also uses beacon frames inspired by RI-MAC ones and follows the same rules of their usage. Moreover, both protocols are compliant with the IEEE 802.15.4 physical layer standard in terms of frame formats. Finally, CherryRiMAC offers the same guarantee as RI-MAC: sending a data frame is reported to be successful only when its recipient has acknowledged the reception.

However, there are important differences between the protocol that warrant this Chapter. They stem from the fact that CherryRiMAC aims to incorporate ideas already existing in other protocols to increase its performance (especially in terms of reduced power consumption) and improve usability. At the same time, it tries to preserve all advantages of RI-MAC (primarily, the relatively simple design). Therefore, the following sections describe CherryRiMAC by pointing how it differs from RI-MAC.

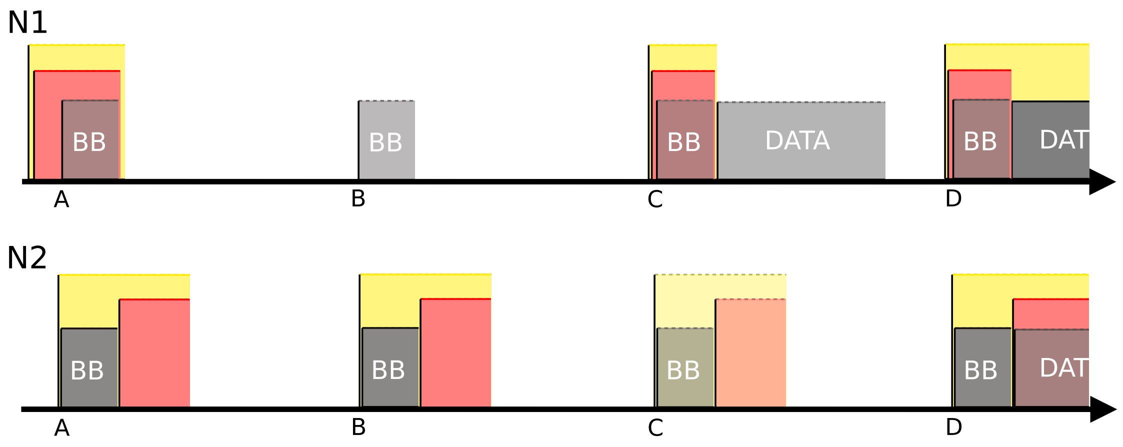
3.1. Phase Awareness

In RI-MAC, when a node wants to send a data frame it starts listening for a beacon from the intended receiver. However, this listening can last a whole duration of the receiver's cycle (or even longer if the beacon cannot be sent because performed before the transmission CCA checks indicate medium activity). This problem of excessive radio usage can be solved by introducing neighbors' phase awareness. The idea has already been implemented in Contiki X-MAC [39], and so CherryRiMAC incorporates a similar mechanism.

To this end, CherryRiMAC enforces a constant cycle duration: a node always wakes up again after the same interval, without any randomization. Moreover, there are just a few preset interval durations. Consequently, a node that knows the time of its neighbor's last activity and the neighbor's cycle duration is able to calculate when the neighbor's next activity will happen. Note that collecting this information about each neighbor does not require any synchronization of a node: a neighbor is active as receiver when it sends its base beacon, so it is enough for the node to save a timestamp of the last base beacon frame from this neighbor. Since CherryRiMAC allows each node to work according to one of a few preset cycle durations, the base beacon frame includes also information about the sender's cycle length (see Section 3.7). In this way, when the node wants to send a data frame, it does not need to wake up and start listening until a base beacon of the recipient arrives, but can instead calculate when to expect the base beacon and activate its radio just in time. This aims to significantly reduce idle listening, thereby lowering energy consumption.

When nodes boot up, each one checks its current wall-clock time, randomizes it in a range of a cycle duration (to minimize the possibility of coincidental synchronization of devices) and saves the result as its cycle start time. It is important that all subsequent wake-ups to send base beacons happen exactly according to this cycle. If a node cannot send a base beacon on time because, for example, it is performing a data frame transmission, it should skip this base beacon (not postpone it, as RI-MAC does) and try to send the next one on time. Precise time keeping allows for a further reduction in energy consumption: to calculate when the recipient's next base beacon should arrive, the sender does not need to know the timestamp of the latest beacon, but a timestamp of any beacon. This means that when a node does not have any data frames to send or does not have any free buffers to receive data frames to, it can keep its radio switched off without needing to send or receive base beacons. In practice, however, because of clock drifts, information about neighbors' cycles should be periodically refreshed, which can be easily achieved through control traffic ordered by a higher network layer.

Note also that CherryRiMAC does not need the beacon-on-request feature (described in Section 2.5): although the sender has to wait (even for a duration of the receiver's whole cycle), it keeps its radio switched off for this time, not listening idly. CherryRiMAC approach can thus lead to higher latencies, as the receiver gets the data frame not in the current but in the following cycle. However, avoiding extra transmissions has the advantage of a lower likelihood of collisions. Figure 3.1 presents the idea of phase awareness.



BB – base beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

Figure 3.1: CherryRiMAC: Phase awareness

When node N1 receives a base beacon from node N2 (moment A), it saves the timestamp of the frame and node N2's cycle length which is included in the beacon. Based on this information it is able to calculate when node N2 wakes up in the future. Although node N1 is not active at moment B it can still precisely wake up for the next beacon when it has a data frame ready to send (moment C). This time, however, node N2 skips its base beacon for some reasons so the data frame is not transmitted. However, the next one is sent on schedule (moment D) and the transmission of the data frame can be performed.

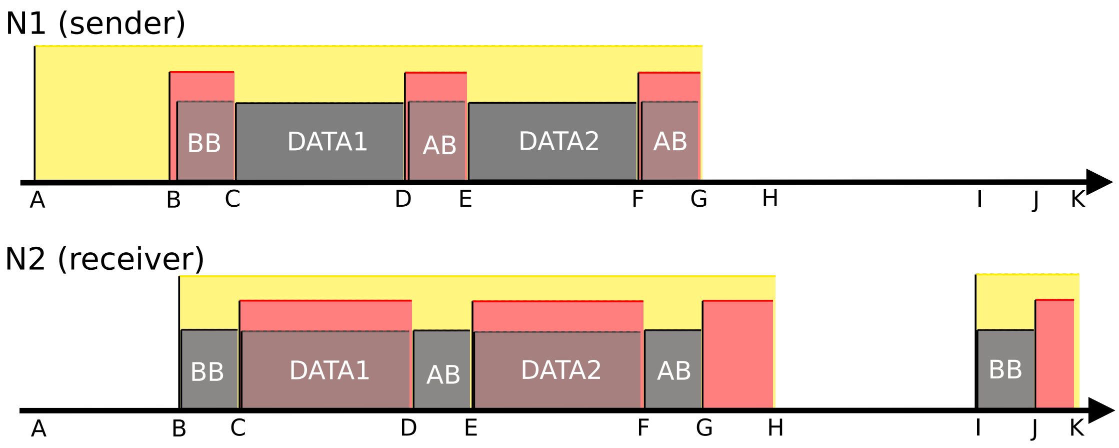
Technically, a module that keeps information about neighbors (let us call it *Neighbors*

List) is not a part of CherryRiMAC. CherryRiMAC requires only that such information is available for a recipient of every data frame that the node wants to be sent. The protocol provides at each node mechanisms to collect and update these data (see Section 3.4) but itself does not manage the Neighbors List, leaving this task to higher networking layers.

3.2. Sending and receiving data frames

In short, sending an unicast data frame [1] in CherryRiMAC follows the same rules as in RI-MAC (see Section 2.2) but, additionally, the phase awareness is incorporated into the process.

When a sender wants to send a data frame, instead of activating its radio (as in RI-MAC) it queries its Neighbors List and calculates the exact time when the next beacon from the recipient of the frame is expected. Then it waits with its radio still switched off and starts listening just a moment before the time. The receiver wakes up according to its (constant) schedule and transmits a base beacon (let us call it *standard base beacon* to distinguish it from others, which are introduced later in the description). The following steps are the same as in RI-MAC: when the sender receives the standard base beacon, it replies with the data frame. The receiver receives this frame, verifies its destination address and transmits an ack beacon. Here, its role is also twofold: it acknowledges that the data frame has been received successfully and requests a transmission of another data frame (from the same or another sender). If a sender does not receive a base beacon or an ack beacon (after transmitting a data frame), it may make another sending attempt of the frame later. In CherryRiMAC, however, it does not continue listening waiting for the next time, but it calculates the expected time of the next standard base beacon and deactivates its radio until that time. If a receiver does not receive any data frame after transmitting its base or ack beacon for a specified time, it stops listening and goes to sleep until its next standard base beacon is scheduled. Figure 3.2 illustrates an exemplary transmission. Note that it is similar to the respective Figure 2.2 describing RI-MAC but contains less listening on the sender side.



BB – standard base beacon, AB – ack beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

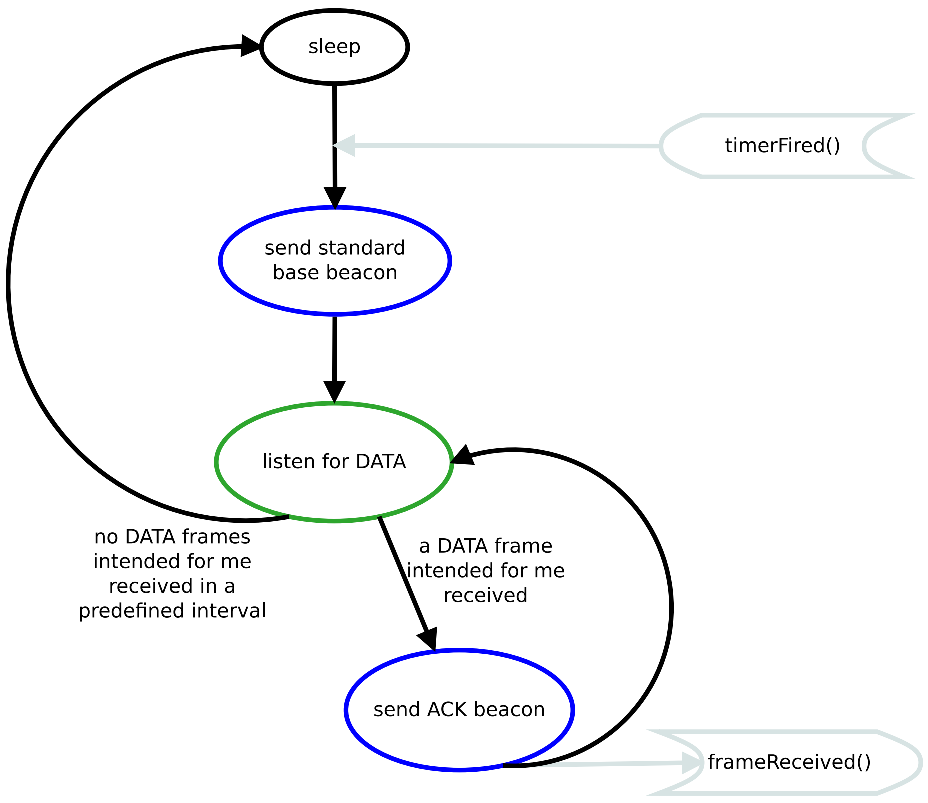
Figure 3.2: CherryRiMAC: Overview

At moment *A*, the sender (node N1) queries its Neighbors List, calculates time of the

[1] If not specified differently, terms *sending* and *data frame* refer to unicast (not broadcast) transmissions in this chapter.

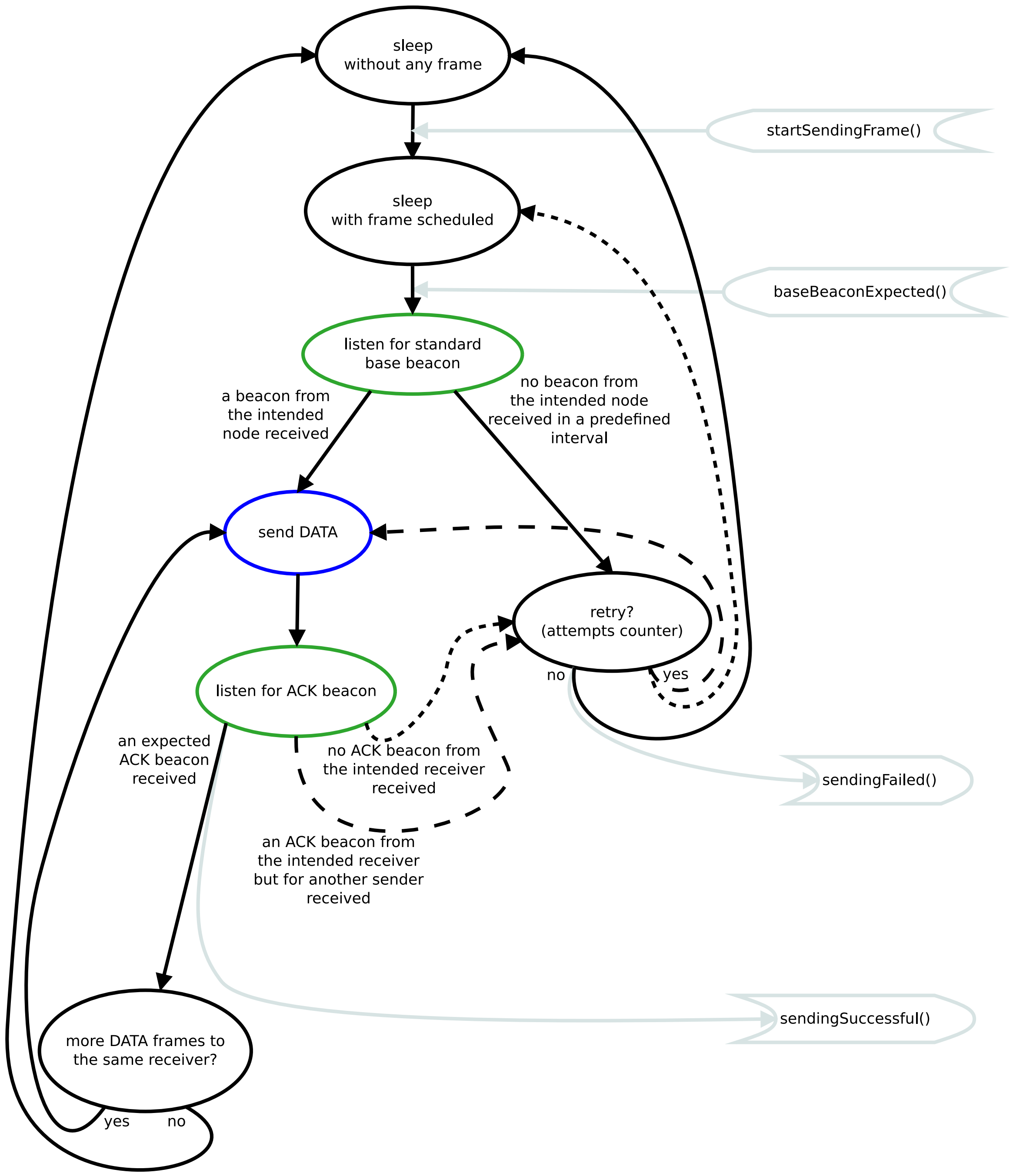
next standard base beacon from the receiver (node N2) and waits until this moment (many microcontrollers can power down not only its radio, but also the main CPU and wake up at an interrupt from an internal timer, thus this period will not be not marked as active in subsequent figures). The receiver wakes up when its standard base beacon is scheduled and transmits it (moment B). The sender starts listening just in time to receive the beacon and, as it comes from the intended node, it transmits the data frame (moment C). It is received by the receiver which acknowledges the reception with an ack beacon (moment D). Because the sender has also another data frame intended for the receiver, it transmits it as a reply for the ack beacon (moment E). It is also successfully received by the receiver, so a next ack beacon is transmitted (moment F). This time, there is no other frame intended for it, so after listening for a specified time it stops listening and goes to sleep (moment H). When the sender receives the acknowledgment of the reception of the second data frame, it also goes immediately to sleep (moment G) because it has no other frames to be sent to the receiver. The receiver wakes up again when its next base beacon is scheduled (moment I), transmits the beacon, listens for incoming data frames (moment J) and, as none is received, it goes to sleep (K). For the sender, as it does not have any more data frames intended for the receiver, there is no reason to wake up to receive the base beacon (moments I–K).

Figure 3.3 presents the internal states of a receiving node. Figure 3.4 presents in turn internal states of a sending node. There is a slight but important difference between CherryRiMAC and RI-MAC here. In CherryRiMAC, the node starts listening when a standard base beacon is expected, not already when a data frame is passed to be sent. Additionally, if a sender retries transmission of a data frame, it goes to sleep and starts listening again when the next beacon is expected (RI-MAC restarts listening immediately).


sleep
timerFired()
send standard base beacon
listen for DATA
no DATA frames intended for me received in a predefined interval
a DATA frame intended for me received
send ACK beacon
frameReceived()

States: blue – transmitting, green – listening, black – radio not in use

Figure 3.3: CherryRiMAC: Receiver – states



sleep without any frame
startSendingFrame()
sleep with frame scheduled
baseBeaconExpected()
listen for standard base beacon
a beacon from the intended node received
no beacon from the intended node received in a predefined interval
send DATA
retry? (attempts counter)
no
yes
listen for ACK beacon
an expected ACK beacon received
no ACK beacon from the intended receiver received
sendingFailed()
an ACK beacon from the intended receiver but for another sender received
sendingSuccessful()
more DATA frames to the same receiver?
yes
no

States: blue – transmitting, green – listening, black – radio not in use
Differently dashed arrows distinguish different control flows passing through the same state.

Figure 3.4: CherryRiMAC: Sender – states

3.3. Contending nodes

The current version of CherryRiMAC does not implement any mechanisms to prevent and resolve frames collisions or to automatically counteract medium congestion. A higher network layer should monitor link quality which can be indirectly done by analyzing failures when sending data frames (CherryRiMAC provides a feedback why sending a frame has failed), frequency of successfully received data frames and contents of the Neighbors List. If the layer detects lower-than-normal performance it can command to change the cycle length or to reinitialize a start time of the cycle.

The decision not to implement such mechanisms was based on preliminary tests on an actual hardware that indicated a really strong capture effect. Moreover, as CherryRiMAC does not randomize the cycle length each time but, contrarily, requires strict time keeping, the behavior of nodes should be much more regular and stable (hence predictable) than when using RI-MAC. However, if an actual performance is lower-than-expected, a mechanism similar to those used in RI-MAC (see Section 2.4) should be easily adaptable for CherryRiMAC.

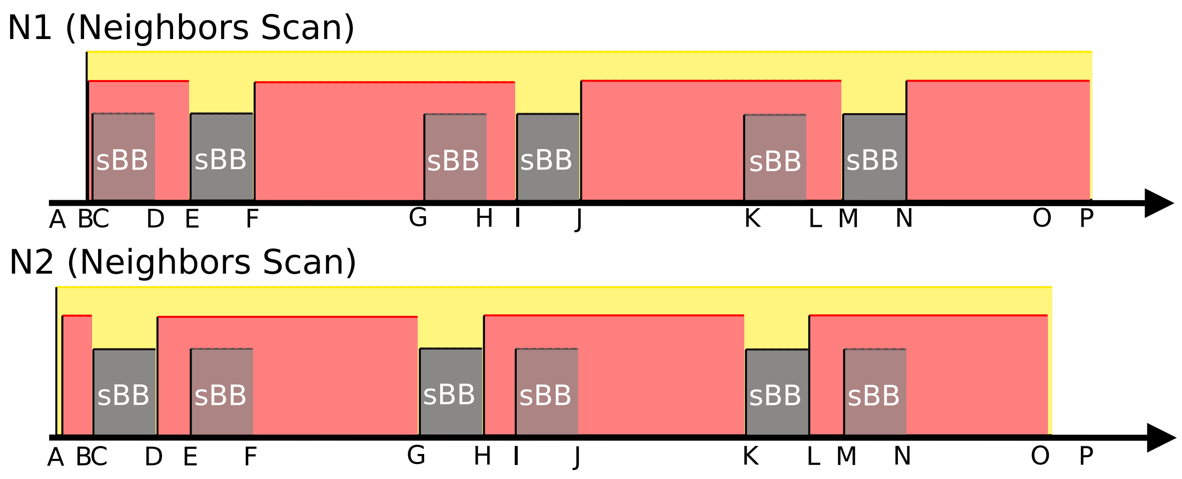
3.4. Scanning neighbors

The primary way of collecting necessary information about a node's neighbors (see Section 3.1) is a *Neighbors Scan* functionality provided by CherryRiMAC. It is a special mode which is mutually exclusive with sending and receiving data frames. When requested, the node activates its radio and starts listening for a duration of k times the longest possible cycle interval ($k = 3$ by default; the scan can be stopped earlier if needed). The node analyzes each received frame and if it is a base beacon, its source address, timestamp and contained cycle length is added to the node's Neighbors List. More precisely, CherryRiMAC provides this information to the Neighbors List which can decide whether to save them or not according to its own policy. Received frames that are not base beacons are dropped because the node executing Neighbors Scan should not participate in any data frame transmission. This is deliberate because if this was allowed to do anything else during a scan, it could suspend listening and perform transmissions, thus it could omit another base beacon or cause a collision with it.

Since many nodes may be scanning neighbors simultaneously, each one should also regularly (according to its schedule) send its own base beacons. Otherwise, if a whole network scans neighbors, none will be discovered. However, CherryRiMAC has to prevent data frame transmissions as a reply for these base beacons because a node drops such frames when it is scanning neighbors. To this end, a special new base beacon type is introduced (let us call it *scan base beacon*). It is the same as the standard base beacon, except it has a special tag set (see Section 3.7 for details). Scan base beacons are transmitted only when a node performs Neighbors Scan (but they also follow the node's schedule) and a node that receives them should not reply. Note that the node that is scanning neighbors processes all received base beacons (not only scan base beacons) as they all are sent according to the sender's schedule.

To automate updating information in Neighbors List, CherryRiMAC additionally processes received base beacons also when performing a unicast or a broadcast transmission. Therefore, when some nodes communicate frequently with each other, their Neighbors Lists are always up-to-date and Neighbors Scan in needed only once at the beginning to discover other nodes.

Figure 3.5 presents an exemplary situation in which two nodes, N1 and N2, are scanning neighbors. While performing Neighbor Scan, nodes are active for a duration equal to k



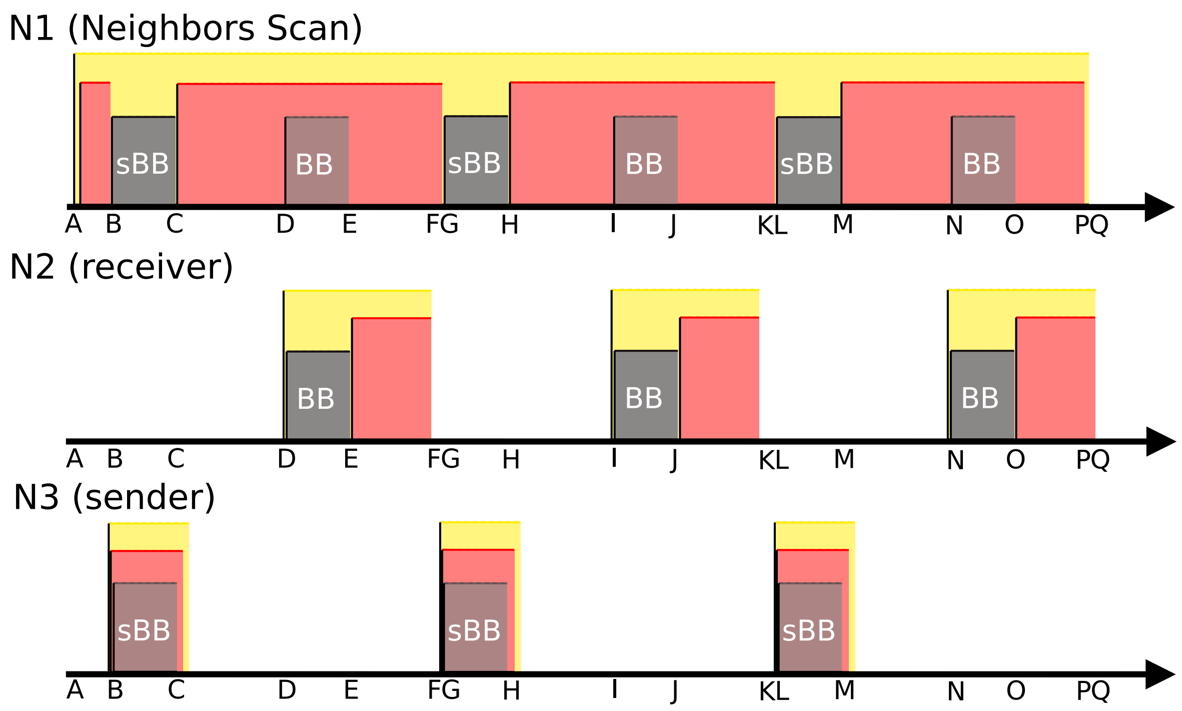
sBB – scan base beacon
yellow – node is active, red – listening, gray – transmitting

Figure 3.5: CherryRiMAC: Neighbors Scan

times the longest cycle length (moments B–P for node N1 and A–O for node N2). Node N2 transmits its scan base beacon at moment C and it is received by node N1 as it is listening. Node N1 records a timestamp of the beacon and adds to its Neighbors List information about node N2's cycle. Node N1's scan base beacon is transmitted at moment E. As node N2 has been listening since it finished its previous transmission (moment D), it receives the frame, analyzes it and adds to its Neighbors List information about node N1. The following scan beacons are transmitted according to nodes' schedules: moments I and J for node N1, G and K for node N2. These beacons are also processed and entries in Neighbors Lists updated. When node N2 finishes Neighbors Scan, it has the up-to-date information about node N1, and node N1 has about node N2.

Figure 3.6 pictures interactions between scanning neighbors and sending or receiving. Node N1 performs Neighbors Scan that lasts from moment A to P. At the same time, node N3 wants to transmit a data frame to node N1 (assuming that it already has required information about node N1 in its Neighbors List) and node N2 can receive a data frame. Node N2 calculates that the next base beacon from node N1 is expected at moment B so it wakes up just in time to receive it. However, as node N1 is performing Neighbors Scan, it transmits a scan base beacon, not a standard one. This is information for node N3 that it should not reply to it, therefore it finishes listening (moment C) and tries again later (moment G). Since then it also receives a scan base beacon, it goes to sleep (moment H) and wakes up for another attempt at moment L. Meanwhile, node N2 executes its own schedule: it transmits its standard base beacons (moments D, I, N) and listens for incoming data frames. None are received, so each time it goes to sleep. However, these beacons are received by node N1 and, as they are standard base beacons (so they are transmitted according to node N2's schedule), node N1 discovers node N2 and adds it to its Neighbors List. Note that since node N3 does not want to receive data frames and thus it does not transmit its base beacons, it is not discovered by node N1.

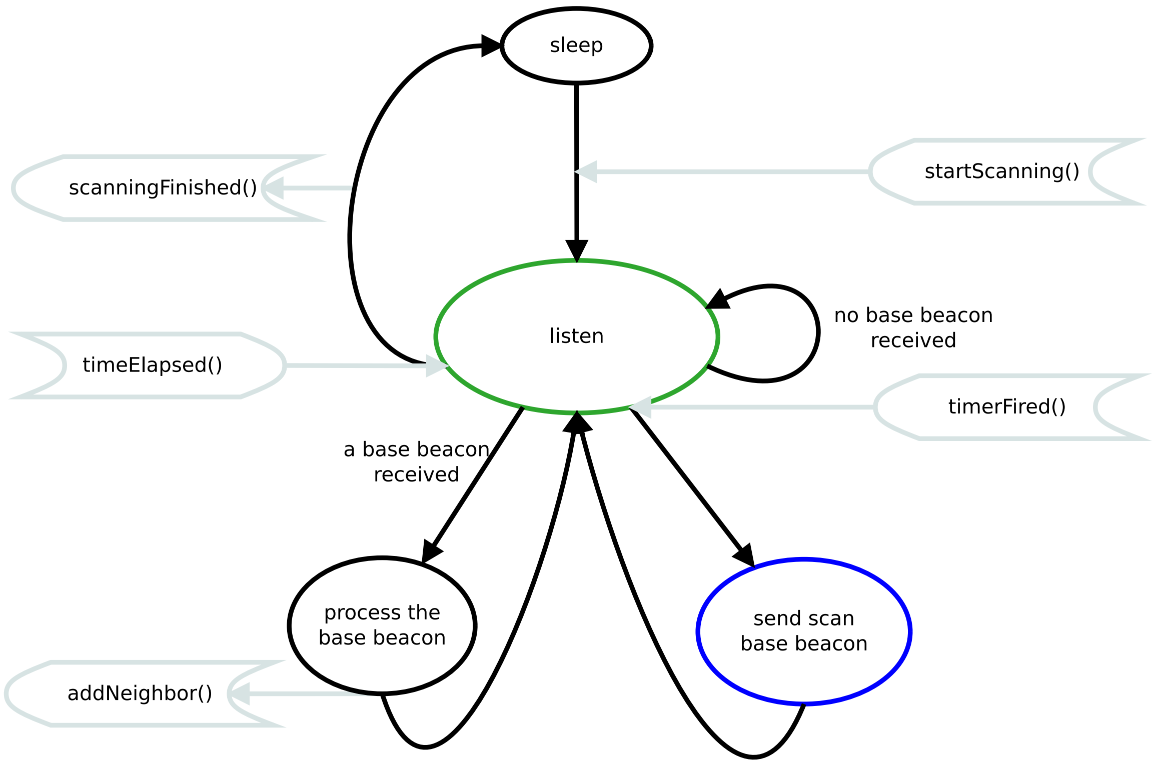
Figure 3.7 presents internal states of a node that executes Neighbors Scan.



N1 (Neighbors Scan)
sBB
BB
sBB
BB
sBB
BB
A B C D E FG H I J KL M N O PQ
N2 (receiver)
BB
BB
BB
A B C D E FG H I J KL M N O PQ
N3 (sender)
sBB
sBB
sBB
A B C D E FG H I J KL M N O PQ

BB – standard base beacon, sBB – scan base beacon
yellow – node is active, red – listening, gray – transmitting

Figure 3.6: CherryRiMAC: Interactions between Neighbors Scan and sending or receiving



States: blue – transmitting, green – listening, black – radio not in use

Figure 3.7: CherryRiMAC: Neighbors Scan – states

3.5. Broadcasting a data frame

CherryRiMAC provides an ability to broadcast a data frame by implementing a simple, best-effort approach.

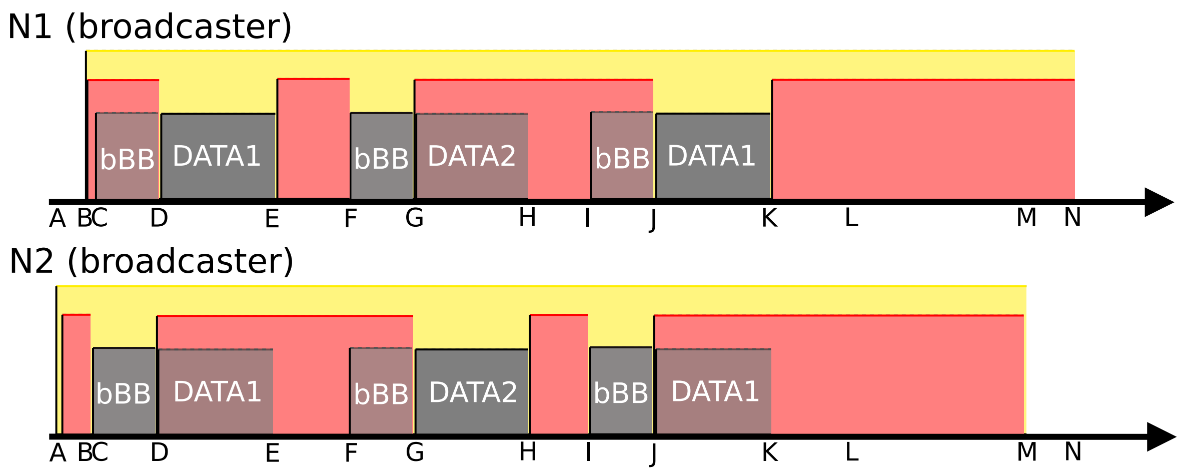
The process of broadcasting a frame is a special mode that lasts a specified time during which the node listens for beacons. If it receives some (both base and ack types), it replies to them by transmitting the broadcast data frame. This way the frame can be received by a node that is executing standard receiving (as described in Section 3.2). However, if the node performing the broadcast also wants to receive frames (e.g. it has free buffers for incoming frames), it should also be sending its base beacons (according to its schedule) and analyzing received frames whether they are data frames.

Contrary to a unicast transmission, broadcast data frames should not be acknowledged: a node that receives such a frame does not reply with an ack beacon. This decision reduces the number of transmitted frames, thereby potentially increasing effectiveness of the broadcast by decreasing the likelihood of a collision.

To simplify implementation and to maximize the time when a broadcasting node is able to receive and reply to a beacon with the data frame, another decision was also made: when executing the Broadcast mode, the node can only receive broadcast data frames. Therefore, to prevent potential senders from transmitting their unicast data frames intended for the node, a new base beacon type is introduced (let us call it *broadcast base beacon*). Similarly to the scan base beacon, broadcast base beacon has an extra tag set (see Section 3.7 for details). The beacon is transmitted only by a node that is broadcasting its data frame and also wants to receive frames simultaneously. When a node receives such a beacon, it can reply to it only with a broadcast data frame.

This approach is a best-effort one: a sender does not know if the frame has been successfully received (and if it has, by which devices). The same data frame can also be received by one node multiple times, because the process of broadcasting should last at least for the duration of the longest possible cycle.

Figure 3.8 presents an exemplary situation in which two nodes are broadcasting their data frames. Node N1 starts the process of broadcasting its frame at moment B by waking up

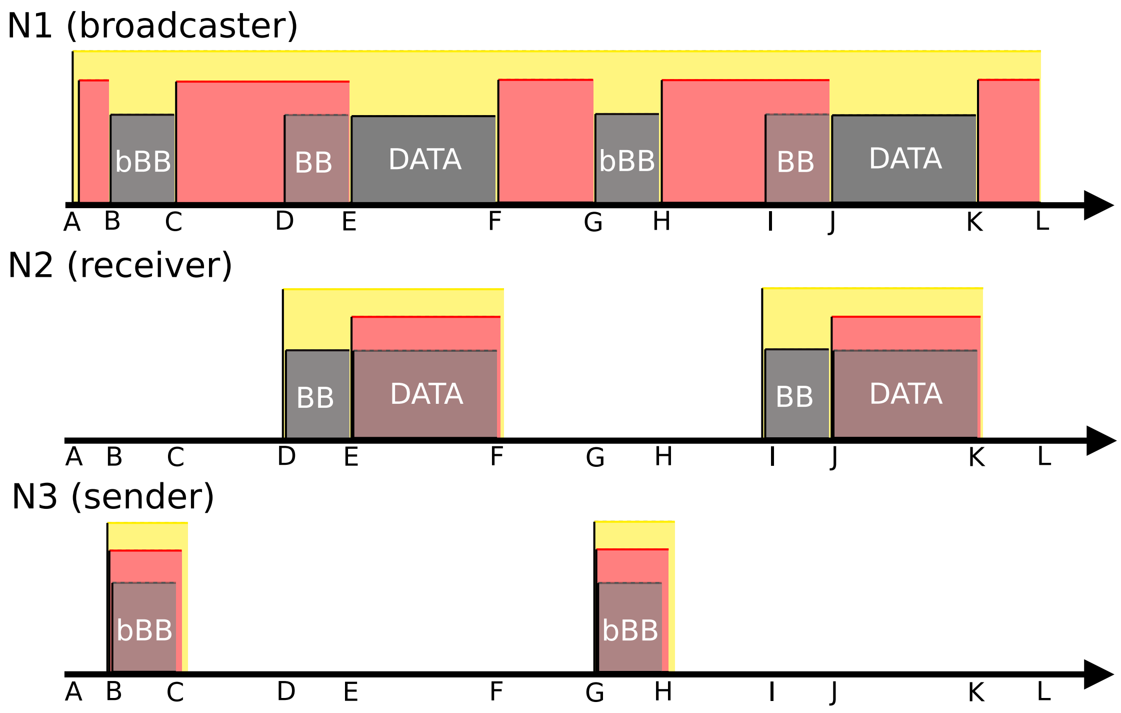


bBB – broadcast base beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

Figure 3.8: CherryRiMAC: Broadcast

and starting listening. The same does node N2 at moment A. Because node N2 also wants to receive frames, it transmits its broadcast base beacon (moment C) which is received by node N1. The latter one replies with its broadcast data frame (moment D). Note, that node N2 does not send an ack beacon after that, but simply continues listening (moment E). Since node N1 also can receive a frame, it transmits its broadcast base beacon according to its schedule (moment F). The frame is received by node N2 which replies with its broadcast data frame (moment G) and continues listening (moment H). In this example node N2 has another free buffer for incoming frames so it transmits its broadcast base beacon also in the next cycle (moment I). It is received by node N1 that replies with the broadcast data frame (moment J). Note, that thereby node N2 receives the same data frame twice. Node N1, obversely to node N2, does not want to receive more frames (e.g. it does not have more free buffers) thus it does not transmit its broadcast base beacon in its next cycle (moment L). Both nodes continue listening until the end of their broadcasting processes and goes to sleep (node N1 at moment N, node N2 at M).

Figure 3.9 illustrates interactions between broadcasting and receiving or unicast sending. In this example node N1 performs broadcasting its data frame (moments A–L) and wants to

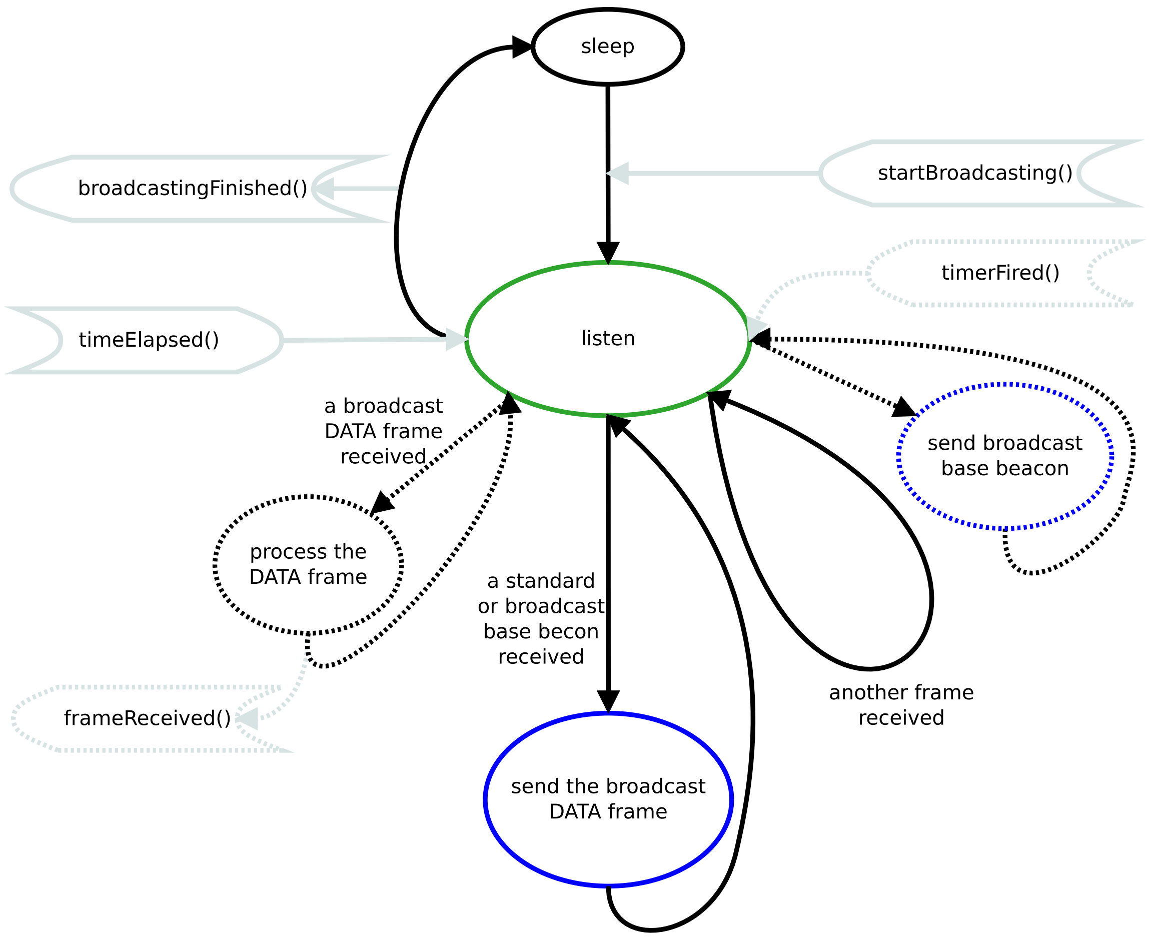


BB – standard base beacon, bBB – broadcast base beacon
yellow – node is active, red – listening, gray – transmitting

Figure 3.9: CherryRiMAC: Interactions between Broadcast and sending or receiving

receive data frames, node N2 executes standard receiving steps and node N3 has an unicast data frame intended for node N1. Node N1 transmits its broadcast base beacon at moment B. As it is sent according to its schedule, node N3 can wake up just in time to receive it. Since the received frame is not a standard base beacon, node N3 does not transmit its data frame but goes to sleep (moment C) and tries again later (moment G). As node N1 continues listening after transmitting its beacon (moment C) it receives a standard base beacon sent by node N2 (moment D). Therefore, it replies with its broadcast data frame at moment E. Node N2 successfully receives it and does not transmit an ack beacon (because it is not a unicast frame) but goes to sleep (moment F). The following cycle is the same as the first one: node N1 transmits its broadcast base beacon (moment G); node N3 receives it and does not reply with its data frame; node N2 transmits its base beacon (moment I) and gets the data frame in the reply (moment J). At the end, node N2 has received the broadcast data frame twice. However, node N3 has not been able to successfully transmit its unicast data frame, because node N1 has been executing the Broadcast mode all the time.

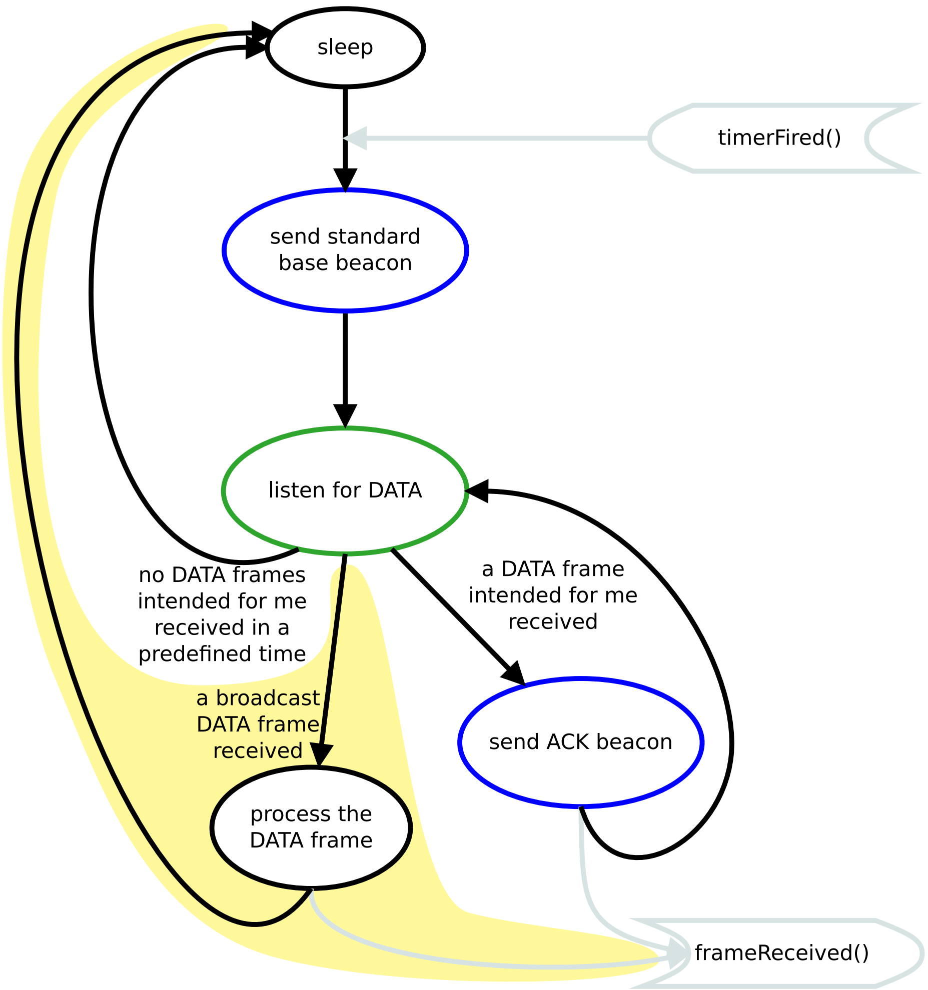
Figure 3.10 presents internal states of a node that is broadcasting a data frame. Note that the node should transmit its broadcast base beacon and process received broadcast data frames only when it also wants to enable receiving.



States: blue – transmitting, green – listening, black – radio not in use
Dotted lines indicate optional actions.

Figure 3.10: CherryRiMAC: Broadcast – states

Figure 3.11 illustrates internal states of a receiving node highlighting actions executed when a broadcast data frame is received (this is an enhanced diagram from Figure 3.3).


sleep
timerFired()
send standard base beacon
listen for DATA
no DATA frames intended for me received in a predefined time
a DATA frame intended for me received
a broadcast DATA frame received
send ACK beacon
process the DATA frame
frameReceived()

States: blue – transmitting, green – listening, black – radio not in use
Yellow background highlights actions executed when a broadcast data frame is received.

Figure 3.11: CherryRiMAC: Receiver – states

3.6. Always Listen mode

Many deployments of low-power wireless networks have a topology similar to the one illustrated in Figure 3.12: nodes can exchange messages with their neighbors using wireless radios, but to upload some data to a server or to enable analyses of the collected readings, the messages need to be send through the Internet (or an intranet). Since the devices usually do not have direct access to the Internet (and, very often, do not even have proper hardware interfaces), a router (called a *gateway*) is required: it forwards packets between both networks.

From the devices' point of view, the gateway is just one more node in the network. However, there is at least one significant difference: the router has unlimited, permanent power supply. Therefore, it does not need to follow radio duty cycling. This dissimilarity can be used to compensate for a problem that usually occurs at the gateway: higher-than-normal traffic as messages from the entire network need to go through this device to reach the Internet. The increased medium congestion also leads to a greater likelihood of frame collisions which deteriorates the performance of the network even further: the necessity to retransmit frames

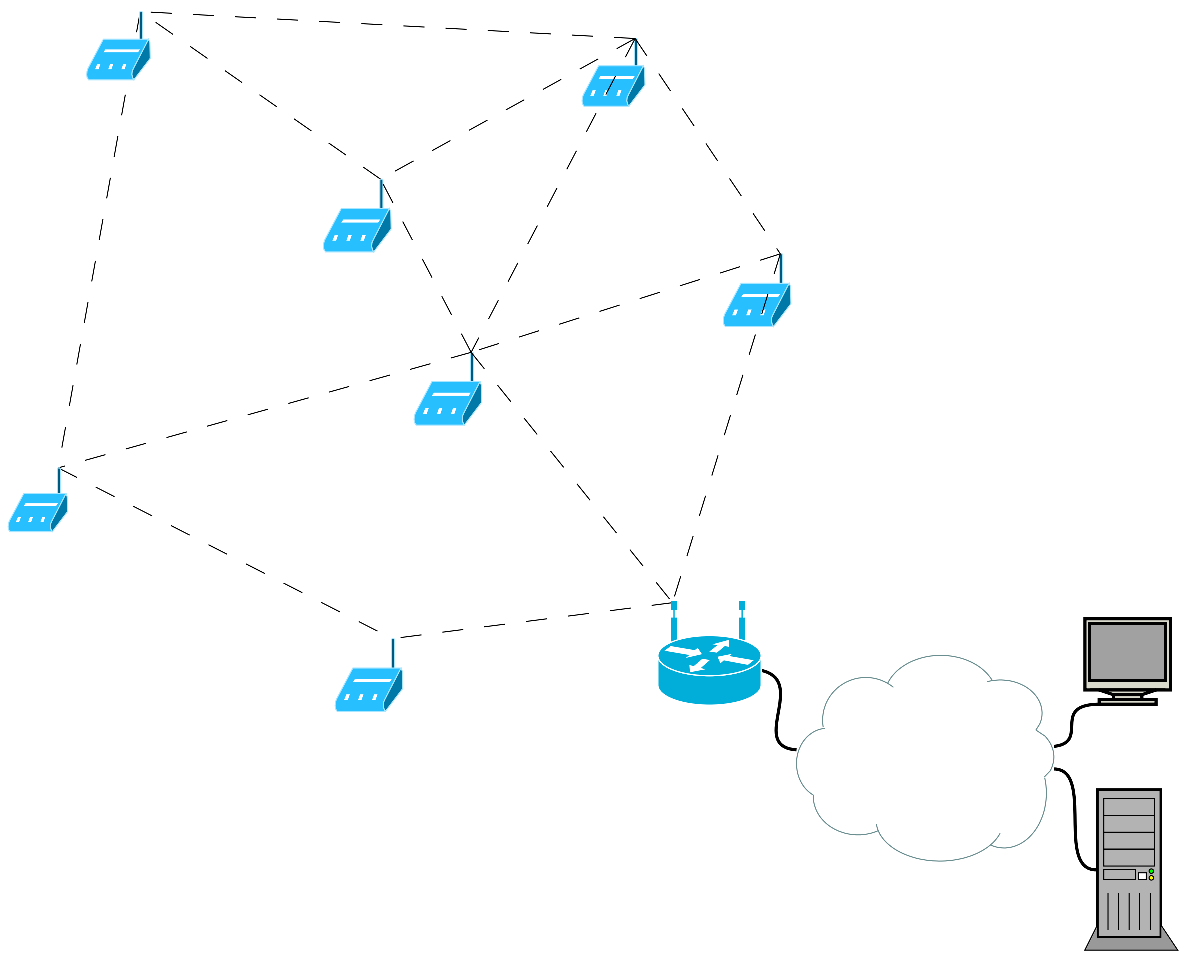


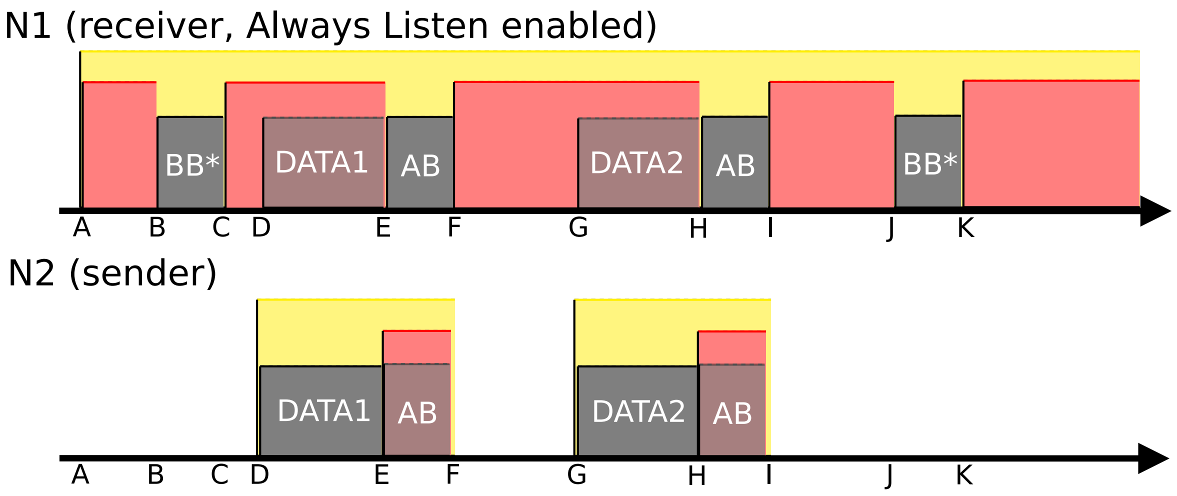
Figure 3.12: A typical topology of a low-power wireless network

increases their latencies and node power consumption. Therefore, CherryRiMAC provides an *Always Listen* mode that, when activated at the gateway, enables a higher throughput by benefiting from the gateway's unlimited power supply.

Always Listen is a background-mode of a node that maximizes time when the node is listening for incoming data frames. It can be active when the node is sending, receiving, scanning neighbors or broadcasting. As a rule of thumb, if the node wants to receive (i.e., has free buffers), in the Always Listen mode it performs listening during the time when it normally sleeps. Thereby, if the node is commanded to only receive frames (no sending, scanning neighbors, etc. is ordered), it is able to receive data frames for the whole cycle and it suspends listening only to transmit its base beacons. In this approach, the neighbors of the node can send their data frames intended for it immediately, without waiting for its standard base beacon. The following steps of the protocol (i.e., transmitting ack beacons and so on) are executed as always.

To notify its neighbors that it is in the Always Listen mode, a node sets a special tag in its base beacons instead of providing its cycle duration (see Section 3.7 for details). In this way, transmissions to a gateway that implements the Always Listen mode can occur multiple times during one cycle, which greatly increases the overall network throughput.

Figure 3.13 presents an exemplary situation in which a receiving node N1 works in the Always Listen mode. The receiver starts receiving at moment *A* by activating listening. Note

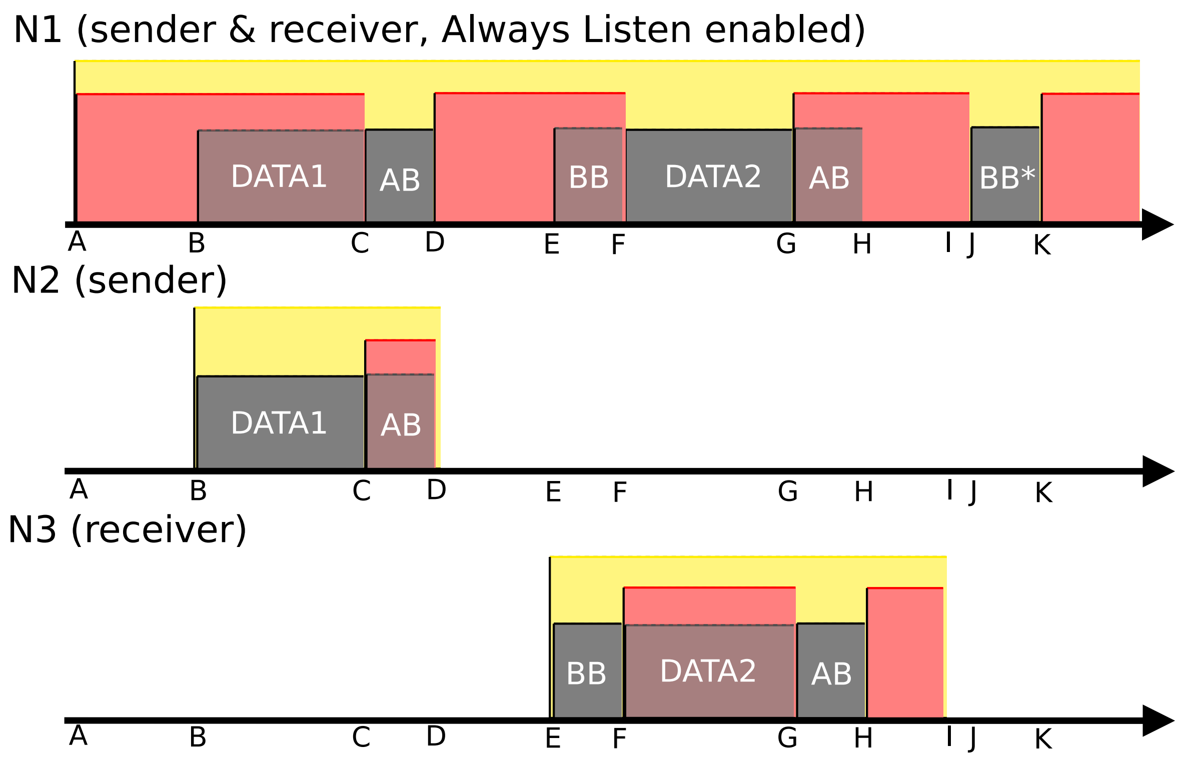


BB* – standard base beacon with Always Listen tag set, AB – ack beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

Figure 3.13: CherryRiMAC: Receiver with Always Listen mode enabled

that it does not wait with its radio switched off until its base beacon is scheduled as happens during standard receiving (cf. Figure 3.2). In contrast, the receiver suspends listening to transmit its standard base beacon on time (moment B) and resumes it afterward (moment C). The sender (node N2) is passed a data frame intended for the receiver at moment D. It queries its Neighbors List and since there is information that the receiver has the Always Listen mode enabled, it transmits the data frame immediately (moment D), without waiting for a standard base beacon from the receiver. The data frame is got by the receiver, thus the sender gets an acknowledgment (moment E). The receiver, having more free buffers for incoming frames, does not go to sleep when it does not receive another frame after transmitting the ack beacon (as during standard receiving), but continues listening (moment F). When the sender is passed another data frame intended for the receiver (moment G), it transmits the frame also immediately, not waiting for the next standard base beacon of the receiver. Since the receiver is listening, it receives the frame, acknowledges it (moment H) and continues listening. Later, it transmits its next base beacon according to the schedule (moment J) and then also resumes listening. Note that in this example the sender was able to transmit its second data frame in the same receiver's cycle as the first frame, not needing to wait until the next base beacon.

Figure 3.14 pictures a situation in which node N1, with the Always Listen mode enabled, wants to send a data frame to node N3 and also wants to receive frames from node N2. It is assumed that all nodes have their Neighbors Lists up-to-date. When node N1 starts, it calculates that a base beacon from node N3 is expected at moment E and starts listening until this time. Note that the node does not wait until this time sleeping, as during a standard transmission, but continuously listening. Node N2 is passed a data frame intended for node N1 at moment B, so it transmits it immediately (without waiting for a base beacon). As node N1 is listening, it receives it, replies with an ack beacon (moment C) and continues listening (moment D). Node N3, which wakes up according to its own schedule, transmits its base beacon (moment E) and as it gets the data frame from node N1 in the reply (moment F), it transmits an ack beacon (moment G). Since node N1 has another free buffer for an incoming frame, it continues listening (moment H) and sends its base beacon according to



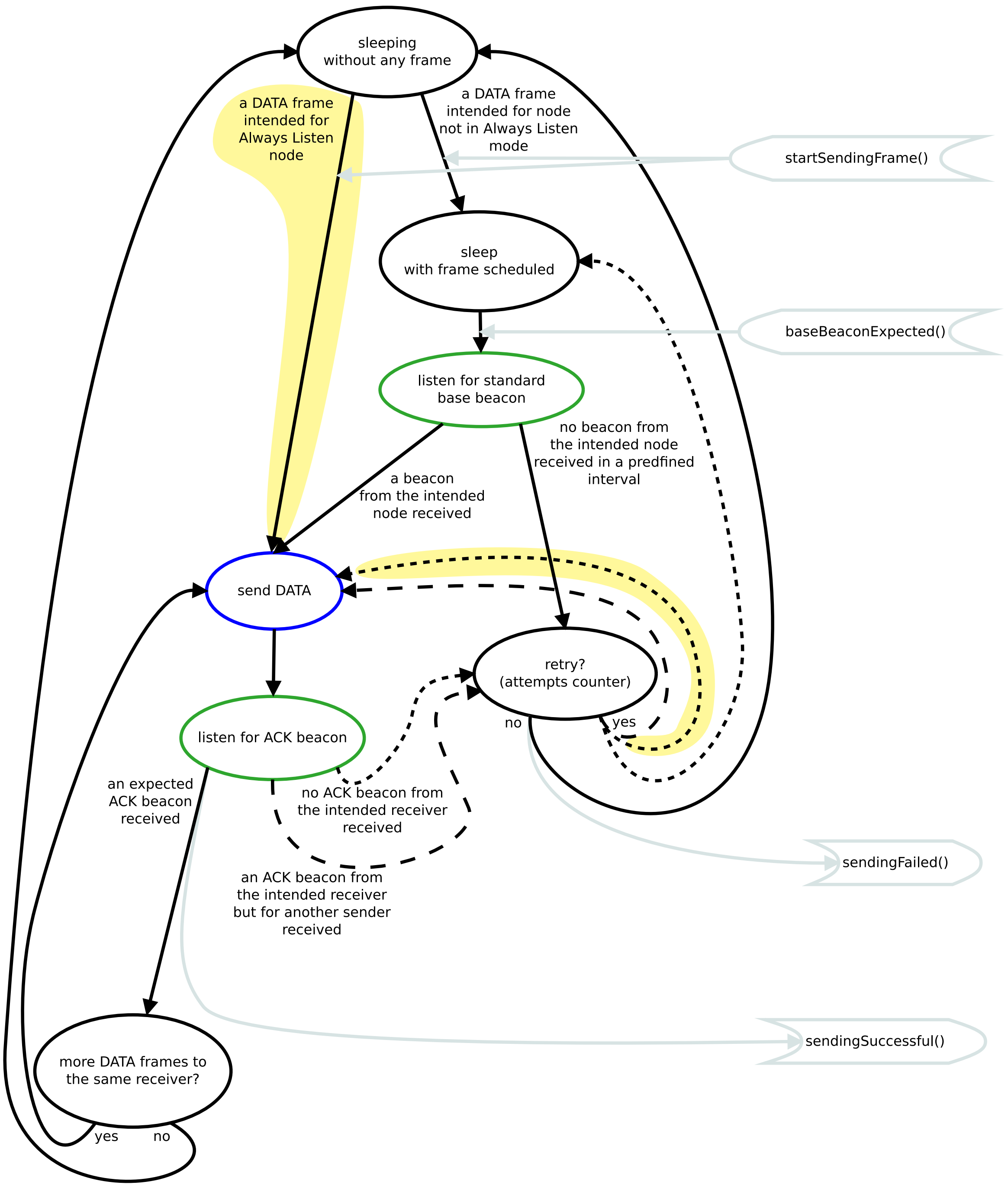
BB – standard base beacon, * – Always Listen tag set, AB – ack beacon, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

Figure 3.14: CherryRiMAC: Sender with Always Listen mode enabled

its own schedule (moment J). Note that in this example node N1, being in the Always Listen mode, is able to send one and receive one data frame even before the end of its current cycle.

Sending and receiving in the Always Listen mode follows the same steps as pictured in Figure 3.4 and Figure 3.3 respectively, but instead of sleeping the node is listening and when it receives a frame it jumps to a proper receiving state. Special modes like Neighbors Scan and Broadcast work the same way as previously. Note that a node with the Always Listen mode enabled but not ready to receive frames (i.e., having no free buffers) does not perform extra listening. Therefore, when it wants to send a unicast frame, the process of sending follows exactly the same steps as with the mode disabled.

Figure 3.15 presents internal states of a sending node (not necessarily always listening) highlighting actions executed when it sends a data frame intended for a node with the Always Listen mode enabled (this is an enhanced diagram from Figure 3.4).



States: blue – transmitting, green – listening, black – radio not in use
Yellow background highlights actions executed when sending to an always listening node.

Figure 3.15: CherryRiMAC: Sender – states

3.7. Beacon frames

CherryRiMAC is designed to be compliant with IEEE802.15.4-2015 at the physical layer. This decision allows to implement the new protocol on chips that are equipped with radios that fulfill physical requirements of the standard because such devices are popular in low-power wireless networks. Additionally, an actual implementation of CherryRiMAC may use already existing functionality, implemented both in hardware and software, to handle some basic tasks related to incoming and outgoing frames. Therefore, IEEE802.15.4 Data frames [2, chapter 7.3.2] are used as data frames in CherryRiMAC. For beacons, in turn, the Multipurpose frame format [2, chapter 7.3.5] is chosen as it is a flexible format that can be used for a variety of applications, according to the standard. The following descriptions present in detail how CherryRiMAC beacons are formatted.

Table 3.1 presents a base beacon in the context of the Multipurpose frame format [2] [3]. Individual fields have their values as follows:

Octets: 1	1	8	1	2
MP Frame Control	Sequence Number	Source Address	Frame Payload	FCS
MHR			MAC Payload	MFR

Table 3.1: Multipurpose frame format: Base beacon

1. **MP Frame Control** Format of this field in illustrated in Table 3.2. Values:

Bits: 0–2	3	4–5	6-7
Frame Type	Long Frame Control	Destination Addressing Mode	Source Addressing Mode

Table 3.2: MP Frame Control field

   1.1. **Frame Type** `0b101`: Multipurpose

   1.2. **Long Frame Control** `0b0`: MP Short Frame Control field (8 bit)

   1.3. **Destination Addressing Mode** `0b00`: PAN ID and address fields are not present.

   1.4. **Source Addressing Mode** `0b11`: Address field contains an extended address (64 bit).

2. **Sequence Number** `0xbe`: Constant value

3. **Source Address** IEEE802.15.4 extended address

4. **Frame Payload** Is used for CherryRiMAC-specific fields, formatted as illustrated in Table 3.3:

[2] All tables in this Section follow IEEE802.15.4 format conventions.

[3] When a frame is presented, the Synchronization header and PHY header are omitted, only the PHY payload is illustrated.

Bits: 0–3	4-7
Base Beacon Type	Interval Code

Table 3.3: Base beacon: Frame Payload

4.1. **Base Beacon Type** Specifies type of the base beacon:
   - `0`: standard base beacon
   - `1`: scan base beacon
   - `2`: broadcast base beacon

4.2. **Interval Code** Code describing node's cycle duration (actual values are incorporated into an implementation). `0` means that the node has the Always Listen mode enabled.

5. **FCS** 16-bit ITU-T CRC (Cyclic Redundancy Check) as described in IEEE802.15.4 [2, chapter 7.2.10].

Table 3.4 presents how the ack beacon is formatted according to the Multipurpose frame format. Individual fields have values as follows:

1. **MP Frame Control** Format of this field is illustrated in Table 3.2. Values:

Octets: 1	1	8	8	2
MP Frame Control	Sequence Number	Destination Address	Source Address	FCS
MHR				MFR

Table 3.4: Multipurpose frame format: ack beacon

1.1. **Frame Type** `0b101`: Multipurpose

1.2. **Long Frame Control** `0`: MP Short Frame Control field (8 bit)

1.3. **Destination Addressing Mode** `0b11`: Address field contains an extended address (64 bit).

1.4. **Source Addressing Mode** `0b11`: Address field contains an extended address (64 bit).

2. **Sequence Number** `0xbe`: Constant value

3. **Destination Address** IEEE802.15.4 extended address

4. **Source Address** IEEE802.15.4 extended address

5. **FCS** 16-bit ITU-T CRC (Cyclic Redundancy Check) as described in IEEE802.15.4 [2, chapter 7.2.10].

3.8. Interference from other networks

In a real-world deployment of a low-power wireless network that uses CherryRiMAC, it may happen that another network also using IEEE802.15.4 frame formats is located within the radio range. Therefore, the new protocol is designed in a way that minimizes the possibility of being disrupted by other networks.

First of all, an implementation should filter received frames and accept only data frames and beacon frames when they are expected. To verify that a received Multipurpose frame is a CherryRiMAC beacon, the node should check its length, value of the MP Frame Control field and the Sequence Number field whether they contain the expected values (as defined in Section 3.7). Using the constant value as a sequence number should additionally reduce the risk of a situation in which another protocol utilizes frames that look like CherryRiMAC beacons: if the field is incremented in the standard way, only 1 in 256 frames has the same value as CherryRiMAC expects.

Contrary to RI-MAC, CherryRiMAC uses IEEE802.15.4 extended addresses in its frames instead of the short ones. As a downside, transmitting each address requires six additional bytes. However, it guarantees that each device has a globally unique address. Therefore, all frames transmitted by another devices in another network are filtered out by the nodes in the network implementing CherryRiMAC when the frames' destination addresses are verified.

The only frame type used by CherryRiMAC that does not include a destination address is the base beacon. However, receiving such a frame from another network does not disrupt the protocol: a node may mistakenly add the sender of this beacon to its Neighbors List, but it should never get from a higher network layer a data frame intended for this node. A device may reply to such a beacon with its broadcast data frame, but this behavior does not validate any guarantee.

There is only one destination address that can be valid in both networks: the broadcast address. It may happen that a broadcast data frame, which has been transmitted in another network, is received and accepted by a node that implements CherryRiMAC. However, only a higher network layer can distinguish that such a frame comes from the other network (by analyzing its source address or payload). Moreover, as broadcast data frames are not acknowledged, receiving such a frame does not generate additional traffic (i.e., ack beacons).

Finally, another network within the radio range almost always leads to a lower performance as the medium is more congested (thus the likelihood of collisions increases). Nevertheless, a node implementing CherryRiMAC will never receive a misleading ack beacon: due to globally unique addresses only the genuine beacon has both source address (receiver of the data frame) and destination address (sender of the data frame) correct.

Although CherryRiMAC is resilient to interference from other networks, the current version of the protocol does not include any mechanisms to protect itself from intruders. They can, for example, destabilize communication in the network by transmitting fake beacons misguiding actual nodes. However, future versions of CherryRiMAC can be extended with cryptographic solutions which allow for verifying genuineness of received frames. Furthermore, embedding cryptography in the design should also enable to hide some information exchanged in beacons so that intruders cannot easily discover configuration of the network.

Chapter 4

Implementation

For evaluation of its correctness and performance CherryRiMAC has been implemented for actual low-power wireless devices: the CherryMote devices, which have been developed within the HENI project. They are equipped with a Texas Instruments CC2650 SimpleLink ultra-low power wireless microcontroller [1] (see Figure 4.1) and are supported by whip6 [6] which is the main operating system used within the project.

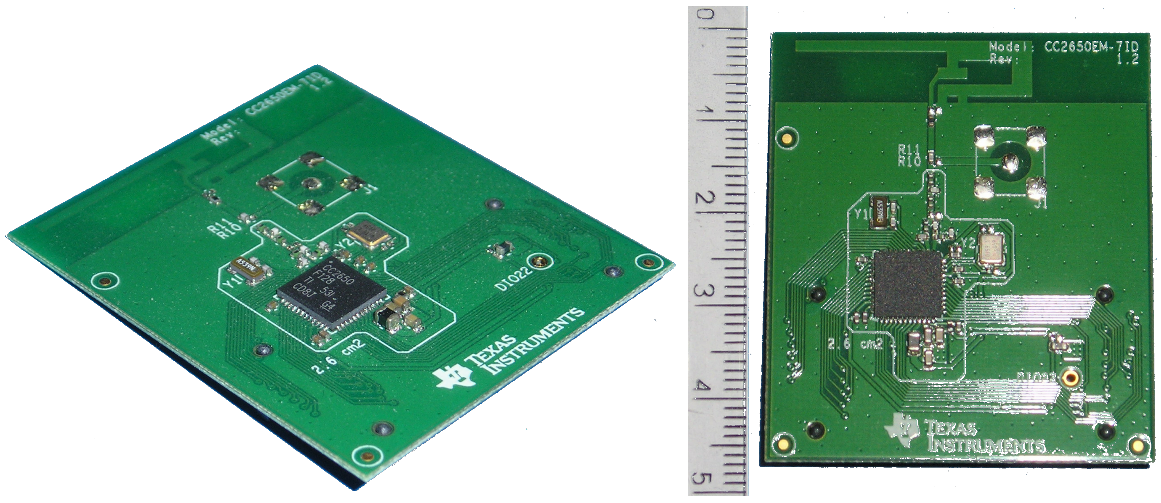


Figure 4.1: Texas Instruments CC2650 Evaluation Module Kit

This chapter discusses selected aspects of the implementation. The first sections provide a general description of the software while the following ones present a more detailed view of the most significant components. Additionally, some important implementation decisions are discussed in this chapter.

4.1. Overview of whip6

Whip6 is an operating system developed by the InviNets company [40] with contributions from the HENI project, designed for low-power devices. It is written in InviNets' fork [41] of the nesC [4, 3] programming language.

The software is divided into components called *modules*. Each module is described by two sets of interfaces: *provided* and *used* by the module. The former ones list which functions are implemented inside this component so that other modules can invoke them. The latter ones describe functions that are required by the component and need to be provided for it by other modules. A whip6 application is a set of components that are connected (*wired*) with each other in a way that fulfills requirements of all its modules. There are special components called *configurations* which do not implement any functionality but only set wiring between modules.

Functions that are described by interfaces are called *commands* and invoking them is called *calling*. Additionally, a component that uses another module may have to implement some functions itself: they are called *events*. A good example of an event is an interrupt handler: the used module *signals* an interrupt and the component that uses it should implement an event that handles it. Within each module, special functions called *tasks* can be *posted* to be executed later: they provide a simple form of concurrency in which posted tasks are executed when the current thread of control (i.e., "chain" of commands and events invoking each other) ends. A new thread of control starts from a posted task or a hardware interrupt.

4.2. CherryRiMAC components

Figure 4.2 pictures components that altogether implement the CherryRiMAC protocol and provide its functionality to higher layers. To simplify the illustration, interface names are omitted and only modules implementing them are shown. To learn about individual interfaces and wiring of CherryRiMAC modules, see Figure B.1 in Appendix B.

The following description presents an overview of newly created modules and configurations:

4.2.1. `CherryRiMACRadioPub`

`CherryRiMACRadioPub` is the main configuration of the implementation, providing all functionality of the new protocol: a higher network layer should use it if it wants to use the wireless radio. This configuration wires all components necessary to execute the CherryRiMAC protocol.

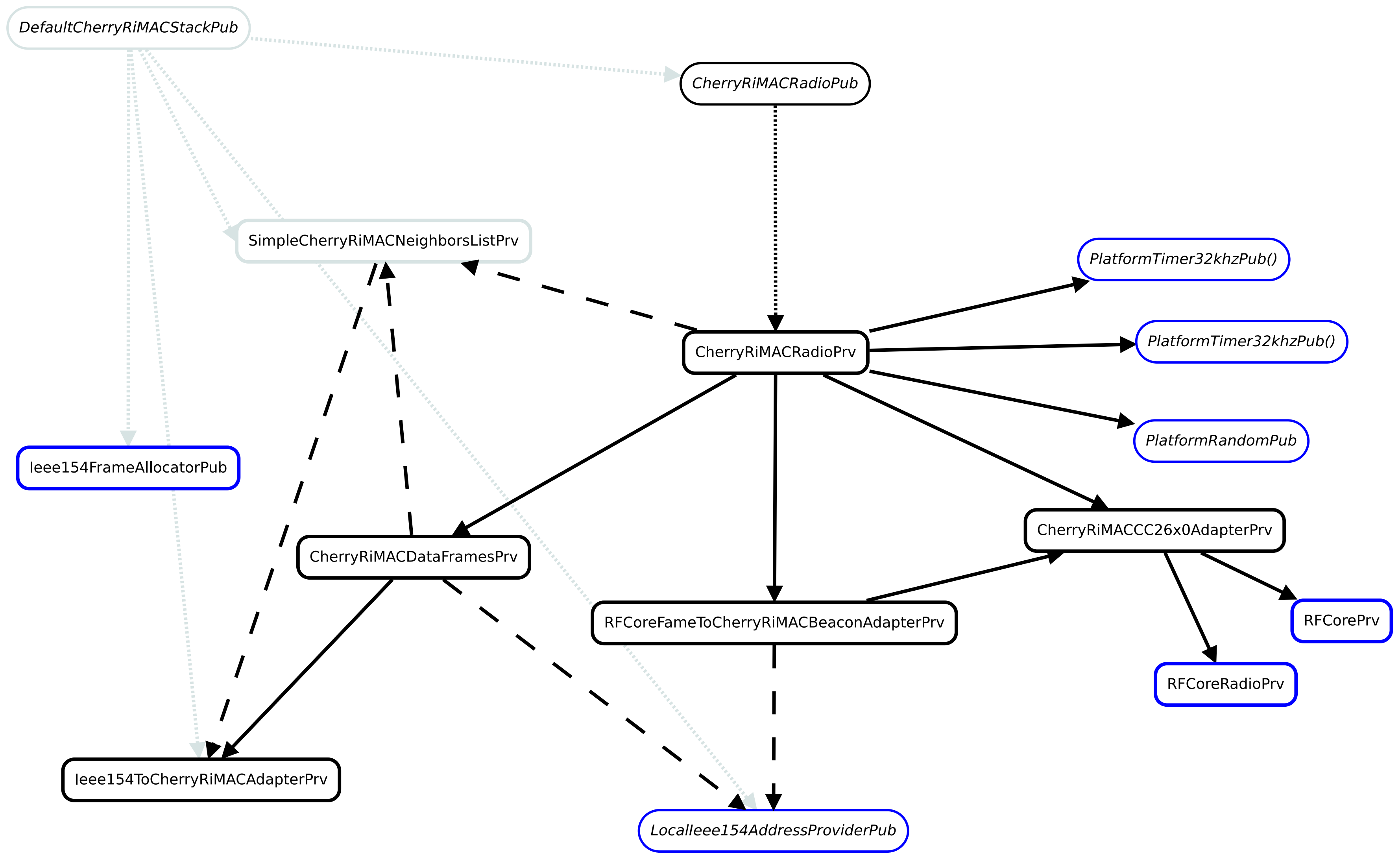
As an external dependency, it expects to be supplied by the user with an implementation of Neighbors List and a component that provides the IEEE802.15.4 extended address of the device.

4.2.2. `CherryRiMACRadioPrv`

`CherryRiMACRadioPrv` is the main module of CherryRiMAC in which the logic of the protocol is implemented. It also provides all necessary control of CherryRiMAC state and parameters, manages the radio and updates the Neighbors List. Most of the features described in this Chapter are implemented within this component.

4.2.3. `CherryRiMACCC26x0AdapterPrv`

`CherryRiMACCC26x0AdapterPrv` is an adapter that transforms generic commands ordered by the `CherryRiMACRadioPrv` module to actual commands expected by the CC2650 chip's radio [42, chapter 23: Radio]. It also interprets hardware interrupts issued by the radio and provides this feedback to the CherryRiMAC logic. Thereby, to port the protocol to another radio (one that uses different commands but implements similar hardware operations) one should only need to replace this module with a newly written adapter, without the necessity to modify other components.


DefaultCherryRiMACStackPub
CherryRiMACRadioPub
SimpleCherryRiMACNeighborsListPrv
PlatformTimer32khzPub()
CherryRiMACRadioPrv
PlatformTimer32khzPub()
PlatformRandomPub
Ieee154FrameAllocatorPub
CherryRiMACCC26x0AdapterPrv
CherryRiMACDataFramesPrv
RFCoreFameToCherryRiMACBeaconAdapterPrv
RFCorePrv
RFCoreRadioPrv
Ieee154ToCherryRiMACAdapterPrv
LocalIeee154AddressProviderPub

Font: Regular – module, Italic – configuration
Color: Black – CherryRiMAC component, Gray – additional component, Blue – already existing whip6's component
Arrows: Solid – dependency wired by CherryRiMAC, Dashed – dependency wired by a higher network layer, Dotted – configuration provides

Figure 4.2: CherryRiMAC implementation: Components and their dependencies

This module uses RFCoreRadioPrv and RFCorePrv, which are already existing components, to communicate with the radio hardware . They provide simple functions like: power up and down the radio, configure it, pass a command to the radio and notify about a hardware interrupt.

The adapter is stateless. It does not track commands issued to the radio. Therefore, the CherryRiMAC logic should monitor the states of pending operations. The module tracks itself only whether the radio is currently being used by CherryRiMAC to filter incoming interrupts because also external components can be wired with the radio driver at the same time.

Additionally, this component provides functions to manipulate frames formatted in the way expected by the radio.

4.2.4. CherryRiMACDataFramesPrv

CherryRiMACDataFramesPrv manages data frames that have been passed to be sent by CherryRiMAC and buffers for incoming frames. The way it handles them is described in detail in Section 4.7.

4.2.5. RFCoreFrameToCherryRiMACBeaconAdapterPrv

RFCoreFrameToCherryRiMACBeaconAdapterPrv is an adapter responsible for generating and handling beacon frames used by CherryRiMAC. It implements the frame format described in Section 3.7.

4.2.6. Ieee154ToCherryRiMACAdapter

Ieee154ToCherryRiMACAdapter handles addresses and data frames described by the IEEE802.15.4 standard. Since CherryRiMAC is compliant with the standard on a physical-layer, this adapter wraps already existing whip6 functions and adds missing ones to facilitate managing frames and addresses by the CherryRiMAC protocol.

This component can be also used as a library by software that needs to prepare frames for sending or interpret received frames.

4.2.7. SimpleCherryRiMACNeighborsListPrv

SimpleCherryRiMACNeighborsListPrv is an implementation of Neighbors List. This module provides all functionality required by CherryRiMAC to handle frame transmissions. It also offers simple functions intended for higher network layers to manage the contents of the list.

However, it is an exemplary implementation, which has been created with simplicity, not performance in mind. Therefore, professional-scale deployments should exchange it for their own realizations of Neighbors Lists, which optimize the required functionality.

Note that technically Neighbors List is not a part of CherryRiMAC itself. Therefore, the users need to wire on their own this or another implementation of Neighbors List when they use the CherryRiMACRadioPub configuration.

4.2.8. DefaultCherryRiMACStackPub

DefaultCherryRiMACStackPub is a configuration that wires an entire CherryRiMAC stack and additional library-like components that are helpful when using the protocol. It is thus a "batteries included" configuration, which has been prepared to facilitate rapid creation of simple applications that need to use the wireless radio.

In contrast, higher network layers in real-world deployments should wire the stack in a customized way only with components that they actually want to use.

4.3. CC2650's radio

The modern Texas Instruments CC2650 chip has a separate processor which manages the radio. This design allows software to benefit from extensive hardware support, which may automate many tasks related to wireless communication, such as executing time-critical parts of radio protocols, filtering incoming frames, preparing frames for transmission, sending acknowledgments, etc. This particular radio has, among others, an especially rich set of features to handle IEEE802.15.4 communication [42, chapter 23.5: IEEE 802.15.4]. Not only does such functionality simplify an implementation of a MAC protocol, but also offloads the main CPU.

This section discusses selected features of the CC2650 chip and elaborates on their applicability for CherryRiMAC.

4.3.1. Conditional execution of "chained" commands

The main CPU can prepare in advance a series of commands for the radio. They can be executed automatically not only simply one by one, but also the execution of a command can depend on the end status of the previous one. This approach, combined with a wide range of commands, allows for creating loops executed without any interaction with the main CPU [42, e.g. chapter 23.3.3.1.13: CMD_COUNT_BRANCH].

However, CherryRiMAC requires frames with a special layout that are not described in any standard: beacons. Therefore, it is not possible to fully automate the protocol as the hardware is not able to analyze a beacon and thus execute a proper action in response to it. For this reason, the current implementation issues only single commands and handles all decision-making in software. Moreover, this software-oriented approach also allows for easily introducing changes and bug fixes, which is an important advantage as this is a prototype implementation of CherryRiMAC. Nevertheless, if the protocol successfully passes the evaluation process, it may be worth to analyze whether at least parts of it can be performed fully automatically by the radio.

4.3.2. Command scheduling

Most of the radio commands can have a scheduled beginning and end of the corresponding operation. This means that a command can be executed by the radio not only immediately at the moment of passing it, but also after a given time: either absolute or relative (e.g. to the time of the command submission, to the start of the previous command, etc.). Similar triggers are also available to plan an end of an operation. To support this functionality, the radio is equipped with its own internal clock (called RAT).

As the design of CherryRiMAC requires strict time keeping, especially when transmissions of base beacons are considered, the implementation uses command scheduling to issue sending these frames. Consequently, base beacons are always transmitted on time, without the need for the main CPU to precisely monitor the current time, which would be particularly challenging because software timers provided by whip6 do not guarantee to fire exactly on time.

Command scheduling is also used on a sender's side: the sender can start listening for a base beacon exactly when it is expected to be received (plus some margins, of course). In effect, the overhearing time can be reduced to minimum, which should significantly reduce power consumption.

Moreover, this feature combined with the one described next allows for easily implementing Neighbors Scan and Broadcast. Since each of these two modes lasts for a specified time and requires listing to be active for this whole duration, by properly scheduling the end of the receiving operation the mode can be automatically finished, without a need to actively monitor its duration by the main CPU.

4.3.3. Foreground and background operations

When being in IEEE 802.15.4 mode, the radio provides two levels of operations: foreground (e.g. transmitting) and background (e.g. receiving) ones. Commands launching them can be passed to the radio independently, although some foreground-level operations require a specific background-level one to be running at the same time. When both operations cannot be executed simultaneously, like sending and listening, the radio automatically suspends the receiving running in the background and resumes it after the transmission is done. This particular feature is used by the implementation for two reasons.

First, it simplifies the software as it orders receiving only once at the beginning of an operation (i.e., sending, receiving, scanning neighbors, etc.) and when the following steps of the protocol are executed, the main CPU changes only a buffer to receive a next frame to (see Section 4.3.4 to learn about buffer management). Transmissions of beacons or data frames during these operations automatically suspend receiving and resume it afterward without any interaction from the software. Moreover, since listening need not be stopped manually before sending, a tighter timing of the protocol can be obtained, as canceling a radio operation and starting a new one lasts some additional time.

Second, issuing a receiving operation before sending a frame guarantees that listening for a frame is resumed afterward as quickly as possible, without any need for an action of the main CPU. In effect the node is able to receive a data frame after transmitting its base beacon without any surplus delay. Equally fast the receiving of an ack beacon is resumed after transmitting a data frame. Since the use of this radio feature eliminates the need for extra delays in the protocol, it should increase the possible throughput and decrease power consumption.

4.3.4. Queues of frames

The radio's receive command requires to be provided not with a buffer to receive a frame to, but with a queue of buffers or even a queue of pointers to buffers. This way the radio can be supplied with multiple buffers at once, and it will automatically store received frames in successive buffers from the queue without the need to issue a new operation each time a frame is received.

CherryRiMAC uses this feature in a specific way: it passes to the radio a circular queue with only one element in it. This one entry contains a pointer to a buffer to receive a frame to. When subsequent steps of the protocol are executed and a next frame should be received, the main CPU exchanges the pointer with an address of a new free buffer and changes its status field. There is thus no need to stop a receiving operation and launch a next one after each reception, even if it is impossible to know in advanced how many buffers will be required (CherryRiMAC does not limit the number of frames that can be sent in one cycle). This approach also allows the implementation to write the received frames directly into the buffers provided by a user, without copying them.

Although the CC2650 Technical Reference Manual [42] does not describe clearly how the radio should work when the receive queue is handled in such a way, tests have shown that the hardware behaves as expected. Therefore, this approach has been chosen to be used when implementing CherryRiMAC as it significantly simplifies the software.

4.3.5. Support for IEEE 802.15.4 frames

When the radio is configured to work in the IEEE802.15.4 mode, it provides support for handling frames described by the standard.

An interesting feature used by CherryRiMAC for outgoing frames is automatic calculation of CRC and appending it as the FCS field to a frame. Also a proper PHY header can be added to the frame by the radio: the software needs to only prepare the MAC header and payload. When a frame is received, the PHY header can be automatically removed from the buffer (as it is needless for the protocol) and the frame can be extended with additional information. The most important one for CherryRiMAC is a timestamp of the frame: a node can know precisely when a base beacon has been received. It can thus track its neighbors' cycles. Moreover, although the radio captures the timestamp when it receives a SFD, it automatically adjusts the value to be equal to the moment when the sender started the transmission, which greatly simplifies calculations of neighbors' expected activity. The adjustment is defined as a parameter in the radio's firmware and may be overridden if needed.

Furthermore, the radio can actively filter received frames according to some simple rules which require the awareness of the IEEE802.15.4 standard, for example, checking type of a frame, rejecting frames with a non-matching destination address, validating frame version, etc. However, it seems that the CC2650 chip does not implement the newest version (2015) of the standard because the Multipurpose frame type (used for CherryRiMAC beacons), which was not present in the earlier versions of the standard, is not recognized correctly. Experiments have shown that a beacon's MAC header is not interpreted properly because, for example, a part of the source address is considered to be the Frame Version field. Additionally, data frames prepared by whip6's libraries are invalidly filtered out without any apparent reason. Therefore, the implementation disables hardware frames filtering and the main CPU performs all necessary checks itself.

The only feature used by CherryRiMAC which may automatically drop frames is validating the CRC of received frames. Consequently, if there is interference and, as a result, a frame is not received correctly, such a frame is removed from the queue by the radio itself and listening continues without requiring any action from the main CPU. This hardware approach should be faster than analyzing the frame in software and issuing subsequent receiving afterward. It also simplifies the implementation as it does not need to include handling of incorrectly received frames.

4.4. Clocks

Software running on the main CPU of the CC2650 chip can use in its computations the current time or a timer functionality by accessing a 32 KiHz [1] clock via an interface provided by whip6. Therefore, all timing and their calculations (e.g. when the next base beacon is expected) are expressed and performed by CherryRiMAC in 32 KiHz ticks as an unit.

The radio is additionally equipped with its own clock (called RAT) which enables it to, among others, execute command scheduling (see Section 4.3.2). However, since the RAT is a 4 MHz [2] clock, all radio commands and all frames' timestamps use this frequency as the unit.

[1] 1 Ki = 1024

[2] 1 M = 1000000

Thereby each interaction with the radio requires conversions between 32 KiHz and 4 MHz ticks.

To convert a duration (a relative time) a value is multiplied (or divided) by 122: the result is not exact, but accurate enough. To convert a 32 KiHz time of an event (an absolute time) to a 4 MHz time the following technique is used: the current wall-clock time is read from both clocks. Then the difference between the event and the current 32 KiHz time is calculated, the result is converted as a relative value by multiplying it by 122 and it is added to the current 4 MHz time read from the other clock. This approach, which requires reading both clocks each time instead of synchronizing them only once at the beginning, guarantees that the conversion is always exact despite a potential clock drift. CherryRiMAC performs such calculations when, for example, it passes to the radio a command scheduling a transmission of a base beacon (as it needs to be performed exactly on time). The time conversion in the reverse way is performed to get a timestamp of a received beacon to update a Neighbors List.

4.5. Event-driven programming and split-phase interfaces

The implementation of CherryRiMAC follows an *even-driven* approach: an action triggers execution of functions which respond to the event. It is a natural way of handling interactions with the radio: the software passes a command to the radio, then the hardware executes the operation and signals its end with a hardware interrupt afterward. It is also a convenient design to implement the MAC protocol: an action of a receiver (or a sender) triggers a transition to the next state of CherryRiMAC and a proper step is executed. What is more, whip6 supports even-driving programming in a straightforward way: nesC events are used to implement actions triggered by hardware, nesC commands implement actions triggered by software.

An advantage to the even-driven approach is a design pattern called *split-phase interface*, in which invoking a lengthy operation is divided into two steps: initiating the operation and, later, signaling its result. A command launches the requested operation and, instead of waiting for its completion, ends with a status informing the caller whether the initiation has been successful. However, the operation may be then still in progress. Only when it finishes, is the caller notified about the competition and gets a feedback whether the action has been performed successfully. This approach is used by CherryRiMAC to provide, for example, the functionality of sending a data frame: a higher layer calls a command that starts the transmission and an event is signaled when the frame has been sent (successfully or not). Receiving, Neighbors Scan and internal interfaces for managing frames and buffers are implemented in a similar manner.

4.6. Global state

The main way of managing control flow in the implementation, so as to ensure that the implementation always executes the right step of CherryRiMAC, is usage of a global state, which represents the current phase of the protocol. Each operation (like sending, receiving, Neighbors Scan, etc.) is divided into consecutive steps describing which action should be performed or which event is expected to occur at the moment. This approach is useful as the implementation follows the event-driven design and the same events happen in different contexts: for example, when the radio signals a reception of a frame, it can be a phase of CherryRiMAC when a base beacon is expected, or when an ack beacon should be received. The global state is the only way for the interrupt handler to decide whether the frame matches the required one and to choose which step should be executed next. Therefore, the whole implementation follows the rule that all commands and events that respond to external actions should at first check the current global state and then decide based on it which functions to invoke next.

The global state of the protocol serves also another purpose: it allows for verifying whether an intended action is allowed to be performed at a given moment. For example, receiving and sending a data frame cannot be executed simultaneously because they consist of different steps which cannot be interrupted by each other. By dividing these operations into disjoint sets of states, the implementation can ensure that sending and receiving are mutually exclusive. This approach is important since CherryRiMAC offers asynchronous interfaces, commands can be called at any time. Therefore, functions that can be executed only in certain modes or only during some phases of the protocol start their executions with a verification of the global state. Such checks are also used within tasks (defined in Section 4.1). They are posted to initiate a new thread of control sometime in the future. Consequently, when invoked, they should verify whether they are allowed to be executed in the existing phase of CherryRiMAC.

4.7. Frames queues

CherryRiMAC allows to be passed multiple free buffers for incoming frames. Similarly, there can be multiple sending operations of different data frames initiated at any time. To handle this functionality, the implementation includes the `CherryRiMACDataFramesPrv` component which stores and manages both the receive and the transmit queues.

4.7.1. Receive queue

Buffers to receive data frames into are kept in a circular queue implemented on an array with a constant number of slots, which follow a FIFO policy (as illustrated in Figure 4.3).

When a higher network layer issues receiving, the buffer which has been passed in the command is added at the end of the queue. It waits there until it becomes the oldest one and CherryRiMAC launches the receiving mode: then it is locked and passed to the radio (see Section 4.3.4 for technical details). Further steps depend on the result of the listening: if a frame has been successfully received to the buffer, it is marked as full and the next one is used for further receiving. It may also happen that the listening ends with an error (e.g., a problem with the radio has occurred): then the buffer is marked with the error and the next one is used for the following attempt. In both cases a task is posted. It processes the already used buffers and signals to the higher network layer whether they contain successfully received frames freeing these slots of the queue for further use. However, a third case is also possible: nothing has been received by the radio when it has been listening. Then the buffer is marked back as a free one and it is used again during the following receiving. Therefore, a buffer is kept in the CherryRiMAC layer as long as a data frame is received or an error occurs.

This approach eliminates the need for restarting receiving by the higher layer when there are no messages intended for the node. However, to be usable, CherryRiMAC should provide a way to cancel listening for incoming frames on the user's demand: the higher layer can recall a buffer. If such a command is issued and the requested buffer has not been filled with a received frame yet, the implementation rearranges the queue by moving the buffer at the head of the queue (to already received buffers) marking that it has been canceled and swaps another free buffer to this slot. Following the split-phase design (see Section 4.5), the command ends then with success but the buffer is actually "returned" to the higher layer later

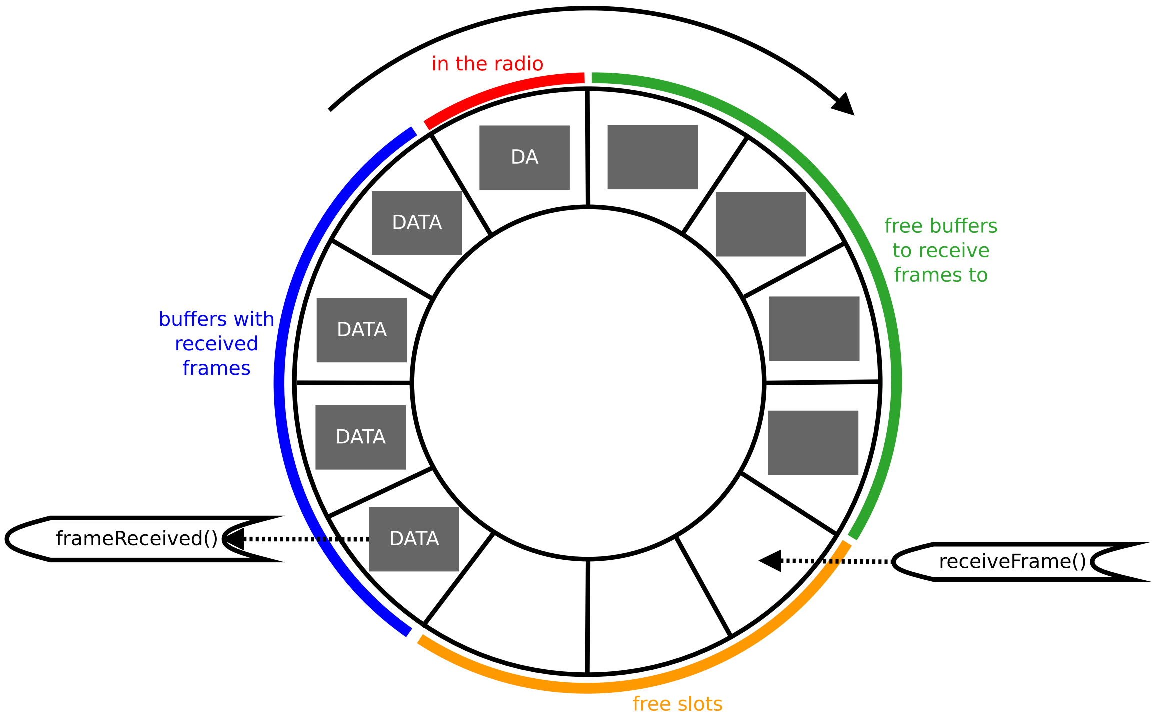


Figure 4.3: CherryRiMAC implementation: Receive queue

when a proper event is signaled. Note that a buffer that has already been passed to the radio cannot be retracted at this moment and the user is notified to try again (when the receiving finishes).

4.7.2. Transmit queue

Handling data frames that have been passed to be transmitted is a more complicated task. As described in Section 2.6, the FIFO strategy for choosing the first frame to be sent is a fair algorithm, but can lead to high latencies. Conversely, calculating which beacon is expected the earliest and transmitting a frame intended for that node may lead to starvation, when two potential receivers have their cycles so close to each other that after replying to the first one it is too late to prepare the other transmission. Therefore, a hybrid approach has been used in CherryRiMAC.

The module implements four lists keeping statically allocated slots (as illustrated in Figure 4.4). One list is used for free slots, one for slots which contain already sent data frames (and those which have failed to be transmitted) while the other two implement a two-tier queue. A new frame passed to be sent is appended to the *Minor* queue which sorts frames with the FIFO strategy. However, when choosing a data frame to be sent next, the Neighbors List is queried and a frame that is expected to have the earliest possible transmission is picked. If there are two such frames, a FIFO policy is used. Packets intended for Always Listen nodes (see Section 3.6) and broadcast data frames are considered to be sendable immediately. When the chosen frame is loaded to the radio, all other frames which have been placed in the Minor queue ahead of the selected one have their *overtake* counters increased. If the counter reaches a preset limit, such a data frame is moved to the other tier: the *Major* queue in which the strict FIFO policy is used.

Therefore, to prevent starvation and minimize frame transmission latencies at the same

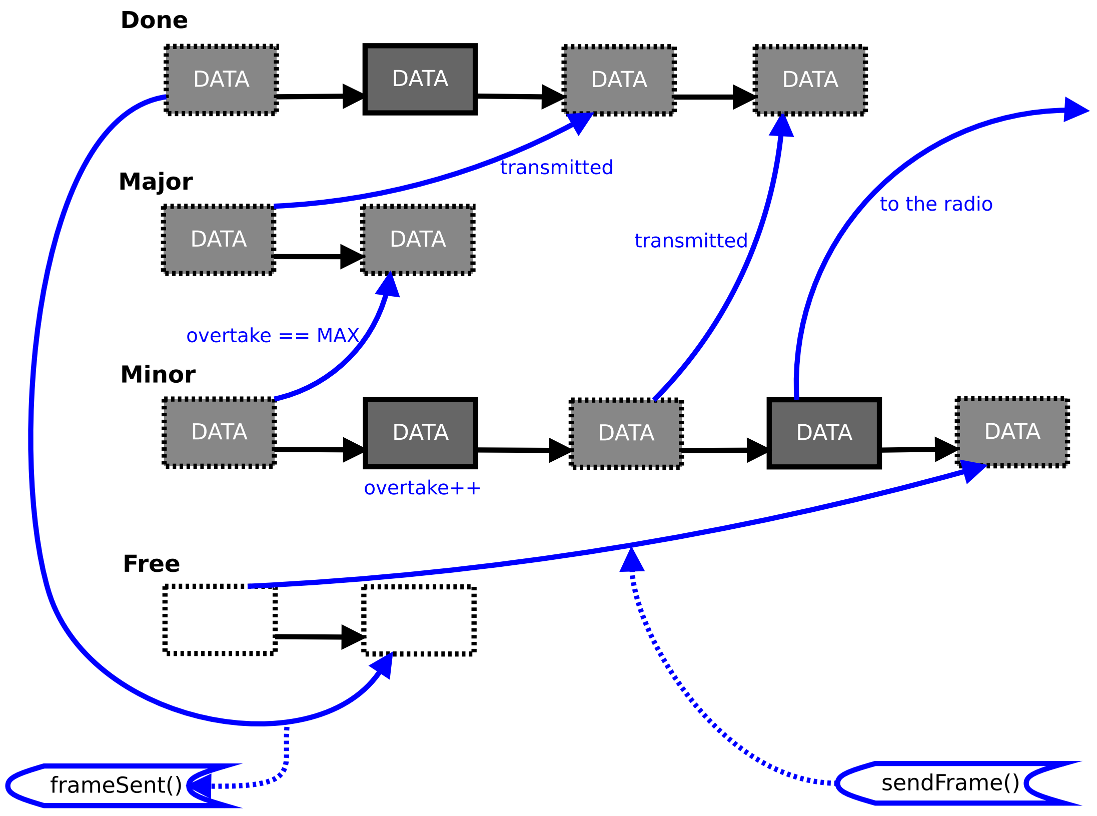


Figure 4.4: CherryRiMAC implementation: Transmit queue

time, the module `CherryRiMACDataFramesPrv` executes the following algorithm when it is asked to choose a data frame to be sent now. If the Major queue is not empty, pick the first frame from it. Otherwise, scan through the Minor queue querying the Neighbors List and choose the earliest transmission. When this transmission is being performed, update overtake counters of older frames from the queue and move them to the Major one if necessary.

Frames that have been successfully sent or have not been sent because of errors are moved to the *Done* queue and their completion is signaled to the higher network layer by a proper task later.

4.8. Data frame counters

Each data frame passed to be sent has a set of associated counters. The *overtake* counter, described in Section 4.7.2, is one of them. The others are used to monitor transmission failures and thus allow a higher network layer to estimate the link quality between the device and its neighbors.

A *noroute* counter is increased each time the node has not received an expected base beacon from the receiver. It may mean that the other device is inactive (as a receiver) or that a collision with the beacon has occurred, or that the entry in the node's Neighbors List is not up-to-date and a next Neighbors Scan should be performed.

A *noack* counter indicates how many times a corresponding ack beacon has not been received. The most common reason for such a situation is simultaneous transmission of data frames by two, or more, nodes: because of frames collision or the capture effect (cf. Section 2.4), the other device has not received this frame. In such a situation, the transmission should be retried and if the problem keeps occurring, it may mean that there is a heavy traffic to the receiver whereas this node has a too weak signal to deliver the frame.

It may seem that this counter should be a sub-counter of the noroute one. In effect, transmission failures would be counted during a successful ”contact” with the receiver. However, it has been decided to operate both counters independently to avoid a possibly endless loop of incrementing it and then resetting at the beginning of the next ”contact”.

There is also a *consider* counter, which describes how many times sending this particular frame has been considered by a dispatcher (see Section 4.11 for details). It is a ”last chance” counter which should prevent the frame from being kept by CherryRiMAC forever when this node and another one have colliding cycles, whereas the dispatcher does not implement a fair algorithm.

When any of these counters reaches a preset limit, sending the data frame is canceled, the frame is moved from the Major or the Minor to the Done queue, and an appropriate error is reported to a higher network layer. In other words, all frames passed to CherryRiMAC are guaranteed to be returned back to the user regardless of their transmission status.

Technically, all counters are implemented as decreasing counters, starting with preset values which are decremented by each failure until 0 is reached. This approach allows an user to set different limits for individual data frames that are passed to be sent by providing both a pointer to a frame and a pointer to the settings. Since counters are updated by the implementation also when a frame is sent before they reach their limits, the link quality can be monitored even when a transmission is successful. If no special settings are provided by the user, CherryRiMAC uses default values.

4.9. Concurrency

The CC2650 microcontroller includes only one core. Therefore, no actually simultaneous executions can take place. However, handling a hardware interrupt can preempt the current computation. Consequently, special measures need to be taken to ensure correctness of the implementation in the face of such concurrency.

As far as a MAC protocol is considered, the concurrency correctness should be verified at four levels: two different steps of the protocol are not executed simultaneously, two different actions during one step are not executed simultaneously and there are no race conditions when accessing global variables and hardware. To explain how these issues are handled in the implementation of CherryRiMAC, an overall concurrency-oriented description of the platform is provided first.

4.9.1. Concurrency in the CherryMote platform

The model of concurrent computing implemented in whip6 on the CC2650 chip can be considered to have two contexts in which operations can be performed [3]. Threads of control (see Section 4.1) initiated by tasks belong to a *task context* [4]. Threads of control initiated by hardware interrupts belong to an *asynchronous context.* The only possible suspension of a thread is preemption of a task-context operation by an asynchronous-context operation. In the case of CherryRiMAC, this means that interrupts from the radio can come in the middle of operations initiated by tasks. However, one interrupt handler cannot preempt another one, and

[3]The approach and names used in this description of the concurrency model are intentionally different from those used in the nesC manual [3, chapter 10: Concurrency in nesC]. In my opinion, they are more suitable because of differences between the generic nesC and the CherryMote platform.

[4]Names of contexts are introduced only for the need of this thesis. They are not any official ones.

a task-context command cannot preempt another task-context command. Although in nesC events and commands handling hardware interrupts are marked as *async*, they can be also invoked from task-context operations. Therefore, a function executed as a task-context operation can be interrupted by the same function but initiated within an asynchronous context. Only tasks, as they are always task-context operations, are guaranteed not to be preempted by one another (or by themselves). To prevent any fragment of code from being interrupted, an `atomic` block can be used.

However, the compiler [41] that is used to build whip6 software does not warn about using events and commands not marked as async from those tagged as async. As a result, it may happen that a task is interrupted by a command which does not have any async annotation. Moreover, this compiler does not detect data races when the same variable is used from both task-context and asynchronous-context operations (as the original nesC compiler does). Therefore, a lot of additional effort has been put into ensuring concurrency correctness of the CherryRiMAC implementation.

4.9.2. Concurrency in CherryRiMAC implementation

Both contexts are used within the implementation of CherryRiMAC. Operations that are not time-sensitive are implemented as tasks or commands and events that are invoked only from task-context threads of control. In contrast, consecutive steps of the protocol which should be performed without any delay are implemented as radio interrupt handlers, thereby belonging to the asynchronous context. In effect, time-critical operations can always interrupt non-critical ones and cannot be interrupted by themselves. Additionally, the number of asynchronous-context functions is narrowed to a minimum. In particular, if subsequent steps of an interrupt-initiated operation are not time-critical, they are posted to be performed as tasks later. Altogether this should guarantee smooth transmissions of data frames.

A coarse-grained mechanism for ensuring concurrency correctness is the usage of the global state, as described in Section 4.6. Not only does it manage the execution of individual modes or subsequent steps of the protocol, but also controls access to the hardware. For example, a command `Init.init()` ensures, by setting a special state, that the initialization of the radio and required structures is performed exactly once, regardless of how many times it is invoked. During this state also other non hardware-related structures can be safely initialized. Moreover, the global state ensures that the transmit and the receive queues are used properly: the `CherryRiMACDataFramesPrv` component allows to add a new frame or remove an already passed one at any moment, but selecting a frame for transmission as well as supplying it to and reclaiming it from the radio is allowed to be done only in an established order which is assured by the proper transitions between the protocol's states.

A fine-grained way to guarantee concurrency correctness is an appropriate usage of both contexts. For example, to ensure that there is no data race when accessing the head of a queue where received base beacons are stored, processing them to update a Neighbors List is performed in a task and no asynchronous-context operations are touching this side of the queue. Therefore, there can be only one computation at any time that analyzes and removes stored beacons. A similar mechanism is used to manage activating and deactivating the radio, as described in Section 4.10. However, a lot of effort is required to ensure that all actions and data accesses invoked from both contexts do not lead to incorrect behavior of the protocol, especially as the compiler does not warn about races. Hence, all commands, events and functions in `CherryRiMACRadioPrv` have been manually verified and marked depending on whether they are allowed to be invoked only from the task context (`_TC` pseudo-annotation) or only from the asynchronous context (`_AC`) or both. It has been also checked by hand that threads of control initiated from tasks include only functions annotated as `_TC` and threads originating from hardware interrupt handlers consists only of those marked as `_AC`. The knowledge about the contexts allowed for a function is especially useful when in the current state of the protocol two different events are possible and only one of them is an asynchronous-context operation. For example, the `STATE_TX_ACK_BEACON` state indicates that the radio may signal successful reception of an ack beacon (an asynchronous-context event) or a timer may signal a timeout of listening for the beacon (a task-context event). Whereas the former event can simply handle the acknowledgment without a risk that the timer goes off during this computation, the latter one needs to additionally re-check and change the global state within an `atomic` block since it can be preempted at any moment.

To ensure that no data-races can occur in the implementation, all global variables of the `CherryRiMACRadioPrv` component have also been manually marked with pseudo-annotations `_TC` and `_AC` (in a similar fashion as functions) according to the contexts in which they are accessed. Based on these tags, it has been verified that all accesses from functions marked as task-context to variables that are used in both contexts have been enclosed in `atomic` blocks. Accesses from operations that have been previously marked as asynchronous-context only do not require atomic operations since they cannot be preempted.

The current implementation makes use of the fact that hardware initiated events (like interrupts from the radio) belong to asynchronous-context and events originating in software (e.g. timers) are task-context operations. However, if a future development of the platform invalidates this assumption, the component can be easily modified to still guarantee the correct behavior: hardware interrupt handlers should then use `atomic` blocks, software initiated events should post tasks that execute the intended actions.

To facilitate the manual process of verifying concurrency correctness of the CherryRiMAC implementation, diagrams illustrating the control flow have been created. They present changes of the global state and allowed contexts of functions, transitions between commands, events and tasks, so that everything can be easily reviewed. These figures are included in Appendix C.

4.10. Radio management

The highest demand for energy occurs when the radio listens or transmits something. Nonetheless, when wireless communication is not needed, the CC2650 chip provides an ability to entirely power down the radio circuit, which allows for reducing energy consumption even further. However, powering the hardware up again lasts some additional time as the proper radio mode is being configured, RAT is synced, frequency synthesizer is activated and calibrated, etc. Whip6 provides interfaces executing this procedure in `RFCoreRadioPrv` and `RFCorePrv` modules.

To minimize power consumption, the CherryRiMAC implementation follows the rule of powering the radio up only when it will be actually used and powering it down as quickly as it is possible, leading to a potentially longer delay when the radio is needed again. The Always Listen mode is an exception to this: see Section 4.13.

The `CherryRiMACRadioPrv` module is responsible for managing the hardware according to this rule. To do that, it tracks the current radio usage: it needs to be active for transmitting, listening and converting frame timestamps (see Section 4.4). These actions can be executed simultaneously (e.g. listening in the background, transmitting in the foreground) but only one of each of these types can be active at any moment. As a result, only three boolean variables are required to monitor the progress. When an operation wants to use the radio, a proper bit is set and the radio is powered up if it has not been already in use. When the operation signals its end, it is checked whether other actions are still using the hardware: if not, the radio is powered down. In this way, the radio is deactivated as quickly as it ends to be needed.

To ensure that the powering-up procedure is not invoked simultaneously multiple times and that it is not interrupted by the powering-down one, deactivating the radio is implemented as a nesC task and an activating function is obliged to be launched only as a task-context operation. However, transmissions are also initiated by asynchronous-context events (hardware interrupts) and should be executed immediately (e.g. replying with an ack beacon). To this end, the implementation also follows an assumption that a transmission can be performed only when receiving is already active in the background. The reason for this is related to the hardware: a frame can be sent then without a need to manually calibrate the frequency synthesizer as is has been done automatically when the receiving has been initiated. Therefore, when a transmission starts, the radio should already be active, and thus the power-up procedure is not performed again.

4.11. Dispatcher

A dispatcher is a central point of the CherryRiMAC implementation. It is a nesC task responsible for deciding which action should be performed next and then initiating it: receiving, unicast sending, broadcast sending, Neighbors Scan or nothing. The task is posted by all commands that issue a new action to CherryRiMAC and all events that end these actions. It is thus executed always when a new decision on what to do next should be made. Figure 4.5 presents this idea.

Since the dispatcher is a nesC task, which is executed sometime in the future after posting it, and as it is posted also from asynchronous events, to ensure that it is executed only when it should be a special global state is introduced (the role of the global state is described in Section 4.6). The task can be run only if this state is the current state and no other actions are allowed to be performed then. Therefore, the task can make a decision and execute it without any harmful consequences for the protocol.

Designing an optimal algorithm for the dispatcher is a difficult problem, as discussed in Section 2.6. Therefore, it has been decided that the current version of CherryRiMAC implements a simple greedy algorithm, which should offer low individual latencies and high device utilization. However, it may lead to starvation of some actions if the node's and its neighbor's cycles are close enough each other. Fortunately, the dispatcher of CherryRiMAC can be easily rewritten to meet particular requirements.

At present, the dispatcher starts with a check whether Neighbors Scan has been issued. If so, it sets the right state and initiates scanning. Otherwise the `CherryRiMACDataFramesPrv` component is queried to find out whether there are available buffers to receive frames into. It is also asked whether there are any data frames ready to be sent: the module responds with an expected time of a possible transmission which is chosen by the algorithm presented in Section 4.7.2. A time of receiving is calculated according to the node's schedule. If both sending and receiving are possible, the dispatcher picks the one that happens earlier: this is the decision which may lead to starvation. Then, the task initiates the action and the next decision will need to be made when the chosen operation ends (successfully or not). If no action is picked, because none has been available, the dispatcher will be triggered again by a command when the user issues a new operation.

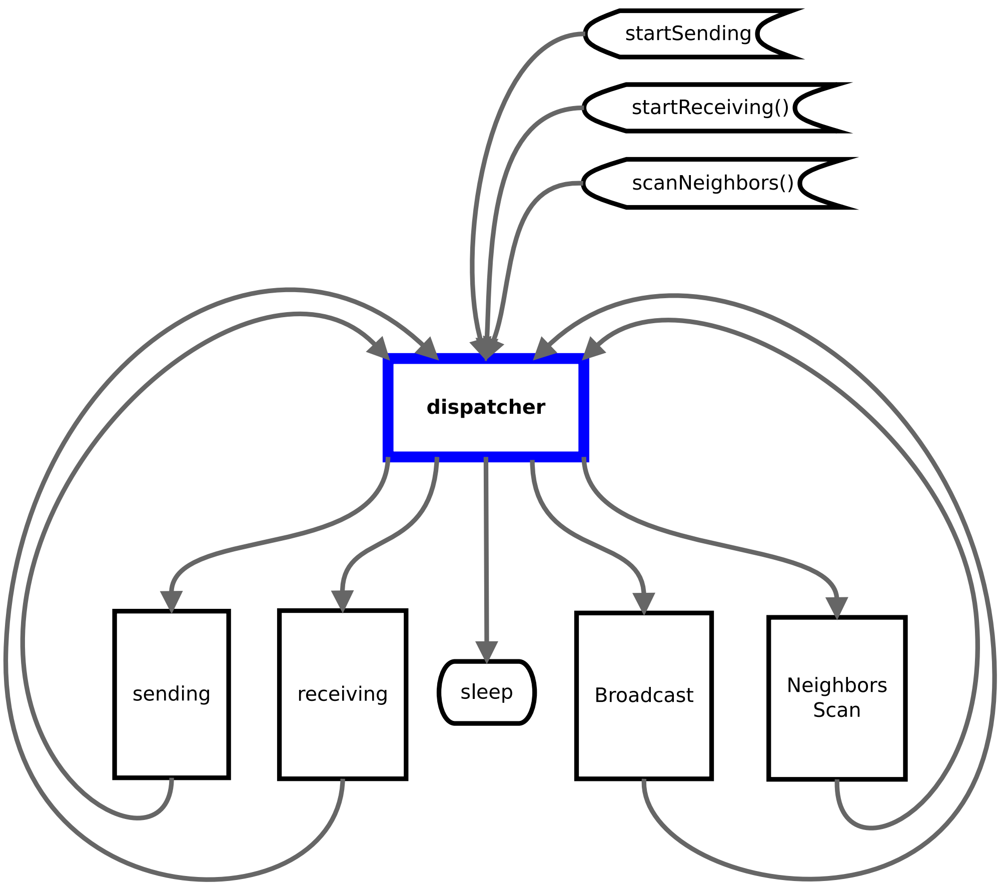


Figure 4.5: CherryRiMAC implementation: Dispatcher

4.12. CherryRiMAC operations

CherryRiMAC provides to a higher network layer four operations: receiving a frame, sending an unicast data frame, broadcasting a frame and scanning neighbors. They are different, mutually exclusive, modes the execution of which is launched by the dispatcher. The following descriptions explain how they are implemented, whereas possible control flows are illustrated in detail in the diagrams in Appendix C.

4.12.1. Receiving

The process of receiving a data frame is initiated by a user invoking the `startReceivingFrame()` command. Then, the new buffer is passed to the `CherryRiMACDataFramesPrv` component to be added to the receive queue and the `dispatcher()` task is posted. When it decides that the next operation is receiving, it calculates the time when the base beacon is scheduled and arms a software timer for a moment before this time. Which further steps are performed by the node, it depends on many variables. Therefore, to simplify the following description it is presented in two parts: nominal operation and failover paths.

Nominal operation

When the timer goes off, a free buffer is marked as locked and passed to the radio which starts listening. As this operation succeeds, the radio is commanded to transmit a standard base beacon at the right moment: it automatically suspends listening, sends the beacon at the scheduled time and resumes receiving afterward (as described in Section 4.3.3). Thereby, when a data frame is transmitted in the reply to the beacon, the radio can already receive it. When the hardware signals the successful reception, it is checked whether the received data frame is intended for the node. If it is, the following actions depend whether it is a broadcast or a unicast frame. In the first case, the buffer is marked as full and since CherryRiMAC does not require to acknowledge a reception of a broadcast data frame, the radio is commanded to finish listening. In contrast, if the received frame is a unicast one, an ack beacon is prepared and passed to the radio to be sent. However, before that, it is checked whether there is another free buffer available and if there is one, it is loaded to the radio (as described in Section 4.3.4). When another data frame is received to the buffer as the reply to the ack beacon, the above steps are repeated again. It may also happen that there are no more free buffers available after some data frames have been received. In this case, after the last ack beacon is successfully transmitted, the listening is finished instead of being continued. When the radio reports the end of receiving, the dispatcher is invoked to initiate the next operation. The buffers that have been filled with the received data frames are asynchronously returned to the user during or after the receiving process.

Failover paths

When the timer armed by the dispatcher goes off, before locking a free buffer it is verified at first that there is at least one available: a user may have revoked it in the meantime. If they have, the receiving operation is canceled and the dispatcher is posted again.

The radio listening operation is always launched before a transmission, even if an incoming frame is expected only as a reply to the frame which will be sent. However, it may happen that some frames arrive before the transmission is performed. Such frames are dropped so that the buffer is still free and the radio simply continues listening.

Durations of all steps of the protocol which include listening for incoming frames are limited (see Section 4.15 for exact values). It is achieved by arming a software timer at the beginning of such a phase for a time equal to the expected duration of this step. If a frame is received before the timer goes off, the timer is disarmed and the operation is continued normally. However, if no frames are received in the expected time, the timer signals a proper event which commands the radio to finish listening. When it is done, the still-free buffer is unlocked. The same steps are also executed when the received data frame is not intended for the node. Note that the buffer cannot be put back already when the timer goes off because it is then still used by the radio.

4.12.2. Sending a unicast data frame

A unicast transmission is initiated by a user invoking the `startSendingFrame()` command passing the data frame and, optionally, a structure with custom values of counters described in Section 4.8. The frame is moved to the `CherryRiMACDataFramesPrv` component where it is appended to the transmit queue (see Section 4.7.2) and the Neighbors List is queried to verify that the required information about the recipient is available. If it is not the case, an error is passed to the user. Otherwise the Neighbors List is requested not to remove this information until the transmission is finished. Then, the task `dispatcher()` is posted. When it decides that a unicast transmission will be performed now, it calculates when a base beacon from the intended receiver is expected and arms a software timer. The further steps can be divided into two descriptions: nominal operation and failover paths.

Nominal operation

As the armed timer goes off, the data frame is locked and the process of sending is being initiated. At first, the radio is activated and scheduled to start listening a moment before the expected beacon arrival (using command scheduling as described in Section 4.3.2). When a frame is received, it is stored in a special `CherryRiMACRadioPrv`'s buffer and is analyzed: it should be a standard base beacon from the intended receiver (as an optimization, ack beacons are also accepted). If it is, the beacon is saved for a now deferred update of the Neighbors List and the radio is commanded to transmit the data frame immediately. The listening is continued afterward since an ack beacon is now expected. When it arrives, the data frame is marked as successfully transmitted and the implementation tries to send another data frame: the `CherryRiMACDataFramesPrv` module is queried whether there is a frame which can be transmitted in response to the recently received beacon. If there is one, the above steps are repeated to send the next frame. Finally, when there are no more data frames which can be sent as the response to the received beacon, the radio is commanded to finish listening and the dispatcher is invoked to initiate a next operation. Frames that have been transmitted are returned to the user asynchronously in the meantime or during following operations.

The above description presents how a data frame is sent to a node which works in a regular cycle. However, since CherryRiMAC offers the Always Listen mode, transmissions to devices in this mode differ a bit: they can be performed immediately, without waiting for a base beacon. Therefore, if the `dispatcher()` initiates a transmission to such a device, instead of arming a timer it invokes directly further steps that open with activating the radio and launching listening (to be able to receive an ack beacon immediately afterward). Then the radio is commanded to transmit the data frame and the following steps (i.e., waiting for the acknowledgment and so on) are executed as during a standard transmission.

Failover paths

When the timer, armed by the dispatcher, goes off, it is at first verified that the data frame has not been revoked by a user. If it has, the operation is canceled and the dispatcher is invoked again.

The durations when the node waits for a standard base beacon or an ack beacon from the intended recipient are limited using a software timer. It is armed for the expected duration at the beginning of the corresponding phase and if the timer fires before the radio signals a reception, it is decided that the beacon has not been sent by the other device. In such a case, the data frame is unlocked and marked with a proper error: `NOROUTE` when no base beacon was received, `NOACK` when no ack beacon was got (see Section 4.8 to learn about errors and corresponding to them counters). In such a situation, the radio is commanded to finish the background listening and the dispatcher is posted to initiate a next operation.

The same `NOACK` error is reported when the received ack beacon is not the expected one. However, as an optimization, instead of finishing the operation, the implementation tries to send another data frame: the `CherryRiMACDataFramesPrv` module is queried whether there is a frame which can be transmitted in response to the received beacon. If there is one, it is transmitted and the operation is continued as normal. Note that if the sender got an ack beacon from the intended receiver but acknowledging another node's frame (as happens when there are contending nodes and the capture effect occurs), it may now choose to transmit again the same data frame.

Data frames, which error counter (see Section 4.8) reached a preset limit, are returned to the user with a proper status asynchronously in the meantime or during following operations.

4.12.3. Sending a broadcast data frame

Broadcast transmissions are provided by the implementation of CherryRiMAC via the same interface as unicast ones but the destination address of the outgoing frame needs to be the extended broadcast address. However, the process of sending such a data frame differs significantly from a unicast transmission. The implemented best-effort approach can be described as two independent event loops.

The first one processes incoming frames. It is initiated at the beginning of the broadcast operation by commanding the radio to start listening for a duration of k $(= 3)$ times the longest CherryRiMAC cycle (the command scheduling described in Section 4.3.2 is used here). Each time a frame is received, it is stored in the `CherryRiMACRadioPrv`'s internal buffer and processed according to its type. First, base beacons are kept in the buffer for deferred updates of the Neighbors List. Second, if a received frame is a standard base beacon, a broadcast base beacon or an ack beacon, the node replies to it by transmitting the broadcast data frame. Third, if a data frame is received, it is verified whether it is intended for the node (in particular it may be a broadcast data frame) and the frame is copied to a receive buffer if a free one is available. Note that this is the only place in the whole implementation of CherryRiMAC where a frame is being copied. This decision has been made because it is impossible to know in advance which type of frame will be received next.

The second event loop is responsible for transmitting the node's broadcast base beacon. When the broadcast is initiated a software timer is armed. As it goes off when a base beacon is scheduled, the current state is verified: if transmission of a frame is in progress, sending the beacon in this cycle is skipped. It is also skipped when the node is not ready to receive frames as it has no free buffers available. Otherwise, the hardware is commanded to transmit the broadcast base beacon on its schedule. This action takes advantage of two radio features: foreground and background operations (see Section 4.3.3) and command scheduling (see Section 4.3.2). When the end of the transmission is signaled by a hardware interrupt, the timer is armed for the next base beacon according to the schedule.

The broadcast operation ends when the radio finishes the scheduled listening. After making sure that all transmissions have also already ended, the frame is put back and the dispatcher is invoked to initiate a next operation. The broadcast data frame is asynchronously returned to the user afterward.

4.12.4. Neighbors Scan

The Neighbors Scan is initiated by a user invoking a `startScanningNeighbors()` command and then launched by the dispatcher. This operation can also be described as two event loops, similar to those implementing the broadcast transmission.

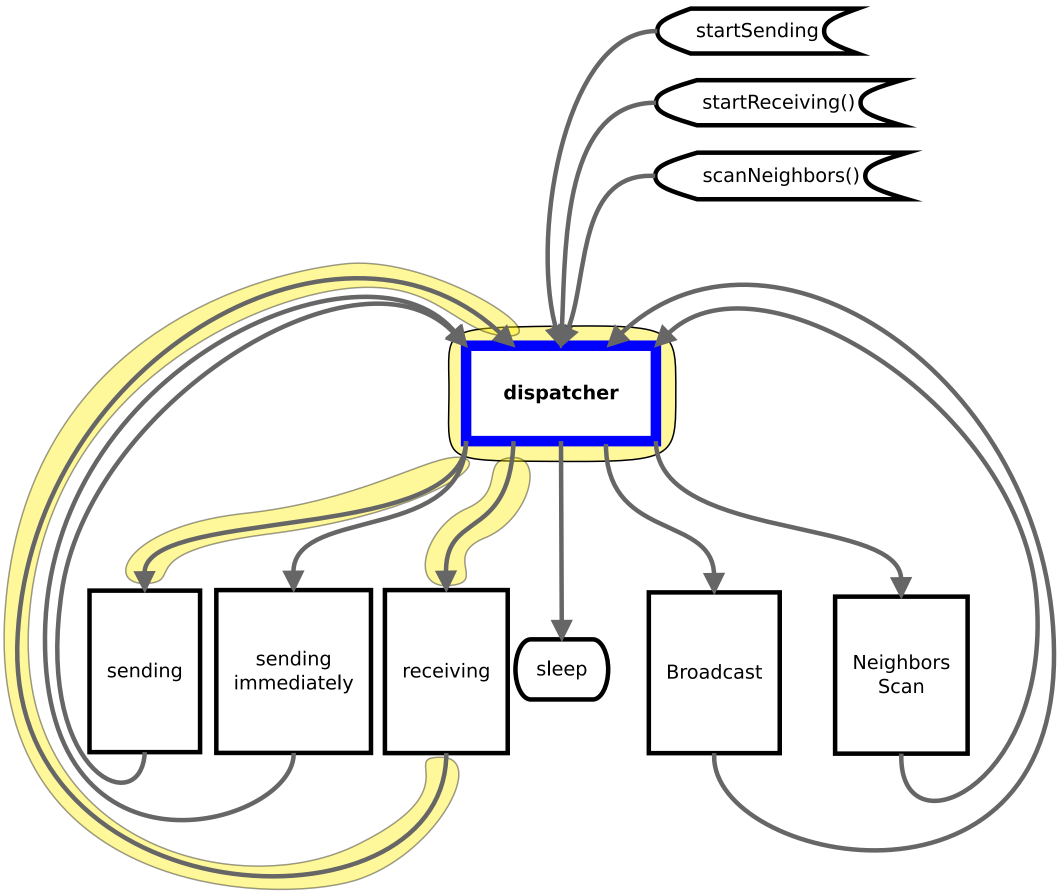
The first event loop is initiated at the beginning of the operation by commanding the radio to start listening for a duration of k times the longest CherryRiMAC cycle. Each received frame is stored in the `CherryRiMACRadioPrv`'s internal buffer and asynchronously processed: base beacons are used to update entries in the Neighbors List whereas other frames are ignored.

The second event loop transmits the node's scan base beacons. A software timer is first armed at the beginning of the operation. When it goes off, the radio is commanded to transmit the beacon at the scheduled time: two hardware's features (foreground and background operations as well as command scheduling) are employed in this process. The timer is armed again for the next base beacon when the end of the transmission is signaled.

The Neighbors Scan ends when the radio finishes the scheduled listening or on the user's request (a `stopScanningNeighbors()` command). In the latter case, if receiving has already been started, the radio is commanded to finish it. When the end is signaled by an interrupt, it is checked whether there is a need to wait for a pending transmission or some received beacons has not been processed yet. When all actions are eventually finished, the end of the Neighbors Scan is signaled to the user and the `dipatcher()` task is posted to launch a new operation.

4.13. Always Listen mode

The aim of the Always Listen mode is to maximize the time when the node is ready to receive incoming frames. The implementation of CherryRiMAC achieves it by launching extra listening when, during a normal operation, the node would be inactive (Figure 4.6 illustrates this approach). If there is a data frame received during such listening, it is handled the same



Yellow background highlights when extra listening can be active.

Figure 4.6: CherryRiMAC implementation: Extra listening in Always Listen mode

way as a frame captured during standard receiving. As a result, it is possible to use the same functions as during standard operations, which minimizes the required changes in the implementation.

A user can activate always listening using the `enableAlwaysListen()` command and when the mode is requested the dispatcher makes a decision which operation to perform next, in the same way as when the mode is disabled. However, it may need to launch additional steps before initiating the chosen operation.

4.13.1. Standard operations with extra listening

If the dispatcher's decision is to perform receiving and the Always Listen mode is requested, in addition to arming a timer for the scheduled base beacon, the `dispatcher()` task also activates the radio, locks a free buffer and passes it to the radio to initiate extra listening. If there has been no frame received when the timer goes off, the receive operation is performed further in the standard way: the standard base beacon is transmitted, the node listens for a data frame and so forth. The only exception is that there is no need to activate the radio as it is already listening. When incoming frames fill all available buffers, the operation is finished also as before: the listening is stopped and the dispatcher invoked. However, when there is at least one free buffer still available, the receiving process ends without interrupting the listening and another data frame can be received until the invoked `dispatcher()` launches the next operation. Note, that when the following operation is also the receiving, the radio is active and ready to receive a frame for the entire cycle.

When a transmission of a unicast data frame intended for a node not in the Always Listen mode is initiated by the dispatcher, it is checked whether extra listening is already active. If it is not, it is started (if there is a free buffer) but only when the duration until the base beacon from the intended receiver is expected is long enough. In all cases, the dispatcher arms a software timer to go off before the beacon is scheduled. When the time comes and no frames have been received so far, the extra listening has to be finished since beacons received during the process of sending the frame cannot be stored in the same buffer as incoming data frames. Therefore, when the timer goes off and extra listening is active, the radio is commanded to finish it and an interrupt handler signaling successful execution of the command initiates the process of sending the frame in the standard way. When it ends, the dispatcher is invoked and it decides which operation is performed next and whether extra listening should be restarted.

A common aspect of Neighbors Scan, a broadcast transmission and a unicast transmission to an always listening node is the possibility to start the operation immediately after the dispatcher launches it. Since during all these modes frames are not received to buffers for incoming data frames, the `dispatcher()` task, before executing proper steps, checks additionally whether the extra listening is active and if it is, the radio is commanded to finish it. The intended operation is performed further in the standard way when the hardware signals the end of extra listening.

Diagrams in Appendix C, presenting implementation of all CherryRiMAC operations, include also highlighted control flows followed when a node has the Always Listen mode activated.

4.13.2. Radio management

Since the Always Listen mode is oriented toward maximizing time when a node can receive frames, even at the cost of energy consumption, the implementation of CherryRiMAC uses another radio management policy when the mode is enabled to facilitate this goal. It differs significantly from the standard one described in Section 4.10: instead of switching the radio off as quickly as possible, the hardware is powered down only when it is no longer needed. When transmitting, listening or querying RAT finishes, the radio is kept active further. Consequently, when a next operation is launched by the `dispatcher()` task, there is no need to switch the hardware on again but it can be immediately passed the command to execute. The same principle is also followed when the extra listening has to be finished before the intended operation: as the radio is kept powered up the proper command can be issued without additional delay after the hardware interrupt signals the end of the extra receiving.

When this policy is active, the radio is powered down by the dispatcher only, when it ascertains that there are no more operations initiated by the user (in particular, no free buffers for incoming frames), so that the energy is not wasted during the node's inactivity. The radio is switched on again when the next command is passed to be executed.

4.13.3. Other differences

When extra listening is requested by a user, the node transmits in its base beacons a special code instead of its cycle duration. In this way, it notifies its neighbors that a frame can be sent to it immediately, not only as a response to a received base beacon. Note that if the node activates its Always Listen mode after it has been discovered by other devices, transmissions to it are still possible without additional scans. As long as it has free buffers for incoming frames, it is sending its base beacons according to the schedule. Therefore, another device sends its first data frame in a standard way and, when it processes the received base beacon, it updates its Neighbors List with information that further frames to that recipient can be sent immediately.

The dispatcher, when it is choosing which operation should be performed next, requires that no other action is being simultaneously executed (as described in Section 4.11). However, the Always Listen mode invalidates this assumption since the extra listening can be still active after a receiving operation ends with invoking the dispatcher. Therefore, additionally to the special global state, the `dispatcher()` task uses also an `atomic` block when it selects an operation to be executed next and initiates it. If a data frame is received during this time, the node will not handle it immediately. However, if the next operation continues the extra listening, the frame reception will be signaled after the dispatcher launches the new action and thus the node may be still able to respond with an ack beacon in time. If not, the sender should try to retransmit the data frame.

4.14. Watchdog

Implementing a watchdog is a common technique used especially when software interacts with hardware. Before initiating an operation a timer is set for a duration a bit longer than the action should take. If the device signals the completion of the action before the clock goes off, the timer is disarmed and further steps are executed normally. However, when the time elapses without any response from the hardware, it indicates a problem and the software should take a failover path (restart the device, rollback to the last known state, etc.).

A radio, especially an advanced one, is such an uncertain point in an implementation of a MAC protocol. Therefore, CherryRiMAC uses an additional software timer to implement a watchdog that is designed to not only detect hardware malfunctions, but also to descry deadlocks in the protocol. It employs a simple observation that the dispatcher (see Section 4.11) is executed regularly during a normal workflow. Therefore, the timer is armed when the `dispatcher()` task initiates a new action and it is disarmed when the dispatcher is invoked after the operation finishes. Because some operations do not have a specified duration (e.g. there is no limit on how many frames can be received at once), the timer is restarted in each internal cycle of these actions. The watchdog may thus not detect a livelock in CherryRiMAC.

An analysis of previously existing whip6 components and an interview with their authors indicate that the CC2650's radio is relatively reliable when working in the IEEE802.15.4 mode. Preliminary tests also support this observation. Therefore, the implemented watchdog only prints a message and panics when it fires without attempting to handle the error. The performed evaluation of the implementation has also backed this observation since no abnormal behavior of the radio was observed and the watchdog did not fired at all.

4.15. Timings

The implementation of CherryRiMAC can be tuned to the particular deployments by adjusting the timings of individual steps of the protocol. The following description explains the currently set values, which can be easily modified by editing the `CherryRiMACConf.h` file. They are also summarized in Table 4.1. All durations are expressed in 32 KiHz ticks [5].

The effective durations of selected steps of the protocol have been measured during actual transmissions between CherryMote devices. The values presented here have been calculated based on wall-clock times reported by devices when they were executing subsequent phases of CherryRiMAC. Experiments have shown that the duration of querying the timer and processing the received value is negligible.

Timing description	Current value (32 KiHz ticks)	Corre-sponding Section	Corresponding constants in `CherryRiMACConf.h`
Additional listening for a base beacon	79	4.15.1	`RECEIVE_BEACON_MARGIN`
Min. listening before the expected timestamp	10	4.15.1	`LISTEN_BEFORE_BASE_BEACON`
Min. listening after the expected timestamp	30	4.15.1	`LISTEN_FOR_BASE_BECON`
Listening for a data frame	215	4.15.1	`LISTEN_FOR_DATA`
Listening for an ack beacon	75	4.15.1	`LISTEN_FOR_ACK`
Initiating sending a base beacon	20	4.15.2	`AWOKEN_BEFORE_BEACON`, `AWOKEN_BEFORE_RX`
Initiating sending a data frame	17	4.15.2	`AWOKEN_BEFORE_TX_CLAIMED`, `AWOKEN_BEFORE_TX`
Powering up the radio and initiating listening	30	4.15.2	`AWOKEN_BEFORE_RX`
Powering up the radio	25	4.15.2	`AWOKEN_BEFORE_TX`
Scheduling timer ahead	43	4.15.3	`WAKE_UP_MARGIN`
Ending extra listening before sending	70	4.15.3	`WAKE_UP_BEFORE_TX_AL`
Dispatcher duration	90	4.15.3	`SCHEDULE_AHEAD_MARGIN`
Scheduling duration during Neighbors Scan and Broadcast	30	4.15.3	`SCHEDULE_BEACON_AHEAD`
Extra listening before sending	200	4.15.4	`AL_TX_DELTA_MIN`
Transmission delay	10	4.15.5	`TRANSMIT_DELAY`

Table 4.1: CherryRiMAC implementation: Timings

[5] 1 Ki = 1024

4.15.1. Listening durations

To be able to successfully receive a base beacon, a node should start listening 10 ticks before the expected timestamp of the frame and continue receiving for 40 ticks. However, because of clock drifts, the duration needs to be longer. Assuming the clock skew of about 20 ppm [43], the implementation employs additionally 79 ticks of listening before and after the required minimum. Consequently, in the worst case scenario, two nodes should be able to successfully transmit and receive a frame after 60 seconds of inactivity.

In most cases, 180 ticks of listening should be enough to receive the longest possible data frame sent in reply to a beacon. It is advisable to increase this value (the implementation uses 215 ticks) since the sender may need some additional time to scan through available frames to find one intended for the sender of the just received beacon.

An ack beacon should be received within 65 ticks of listening. The implementation employs an additional margin of 10 ticks.

4.15.2. Awoken before an operation

To facilitate tuning the protocol, when a scheduled timer goes off to initiate an operation, the implementation checks how much time remains to the moment when the intended action should be started. The aim of this verification is not to launch the operation if the node has been woken too late to be able to perform it on time. However, as these limits do not guarantee that all required steps will definitely finish before the scheduled moment, these values should be treated as a tool to monitor the behavior of the implementation rather than a way to guarantee the correctness of the protocol.

It has been measured that less than 20 ticks are required after a timer goes off to send a base beacon on schedule: this value was checked during Neighbors Scan, Broadcast or the receive operation within an active Always Listen mode. Additional 30 ticks are required when the node also needs to power up the radio and initiate listening before, as happens during standard receiving.

Sending a data frame involves verifying that it is prepared for the transmission and commanding the radio to start listening on time for the expected base beacon, which takes about 17 ticks. Without Always Listening active, the node additionally needs to power up and configure the radio which lasts up to 25 ticks.

4.15.3. Scheduling a timer

The aforementioned values are minimal durations required to successfully perform an operation after a timer goes off. Therefore, the timer should be armed for an earlier moment. Measurements have shown that the proper event is signaled 10 ticks after the scheduled time. Additionally, since the event belongs to the task context and thus it cannot preempt an already running computation, the implementation arms the timer 43 ticks earlier than the required awakening to increase the chance that the function is invoked early enough. An exception to this rule is a situation when the extra listening is active but it should be finished before sending a data frame can be performed (as described is Section 4.13.1). In this case, the timer is armed in addition 70 ticks earlier, accounting for commanding the radio to finish listening and handling an interrupt signaled then.

The dispatcher, deciding which action should be performed next, should not always consider the earliest moment when each beacon from a neighbor is expected (for sending) or the earliest moment when its own base beacon is scheduled (for receiving) as it may not be able to initiate the chosen action before this time comes. A duration of the `dispatcher()` task depends on how many computations it needs to perform. The measured values vary from 35 to 57 ticks. To be reliable even under high load, the implementation uses a duration of 90 ticks in its calculations.

Next to the dispatcher, the timer is also cyclically armed during Neighbors Scans or a Broadcast operation. This process lasts no longer than 30 ticks so this value is used when scheduling a transmission of the next base beacon.

4.15.4. Extra listening before sending

As described in Section 4.13, when the Always Listen mode is enabled, the node may decide to initiate sending a unicast data frame to a not always listening device and additionally launch some extra listening before the operation. When the dispatcher decides whether to issue this extra listening, it estimates how long it can last (initiating it requires about 30 ticks of time). The current setting of 200 ticks means that, if a frame is received, then, after handling it, it will be too late to transmit the node's data frame. Therefore, the device prioritizes receiving above sending, which can be accepted (or even preferred) in some deployments with a gateway device. However, if it is not the desired policy, the value can be easily increased as needed.

4.15.5. Transmission delay

Tests have shown that when a frame is sent as a response to a just received one, the transmission should be delayed a bit because the radio of the other device has to finish that transmitting operation and resume listening before it is ready to receive the frame. Therefore, the implementation sends frames 10 ticks after a command is passed to the hardware instead of transmitting them immediately when the protocol says to do so.

4.15.6. Duration of radio activity

Measurements have shown that when sending the largest possible data frame, the radio on the sender's side is powered up for about 460 ticks, on the receiver's side for about 365 ticks. The durations when the radio is transmitting or receiving are about 415 and 350 ticks respectively. These values are presented in Table 4.2 in reference to possible cycle lengths assuming that one sending or receiving operation is performed during a cycle.

	1 s	3 s	5 s	7 s
sender (460 ticks)	1.39%	0.47%	0.28%	0.20%
receiver (365 ticks)	1.11%	0.37%	0.22%	0.16%

(a) Duration when the radio is powered up

	1 s	3 s	5 s	7 s
sender (415 ticks)	1.26%	0.42%	0.25%	0.18%
receiver (350 ticks)	1.06%	0.36%	0.21%	0.15%

(b) Duration when the radio is transmitting or listening

Table 4.2: CherryRiMAC implementation: Radio activity in reference to cycle duration when transmitting the largest possible data frame

Experiments with a small data frame (4 bytes of payload) indicates that in an average case the radio is active about 350 ticks on the sender's side and 255 ticks on the receiver's, whereas it performs transmitting or listening for about 310 and 245 ticks respectively. These values are presented in Table 4.3 in reference to possible cycle lengths assuming that one sending or receiving operation is performed during a cycle.

	1 s	3 s	5 s	7 s
sender (350 ticks)	1.06%	0.36%	0.21%	0.15%
receiver (255 ticks)	0.77%	0.26%	0.16%	0.11%

(a) Duration when the radio is powered up

	1 s	3 s	5 s	7 s
sender (310 ticks)	0.94%	0.32%	0.19%	0.14%
receiver (245 ticks)	0.74%	0.25%	0.15%	0.11%

(b) Duration when the radio is transmitting or listening

Table 4.3: CherryRiMAC implementation: Radio activity in reference to cycle duration when transmitting a small data frame

The duration of sender's activity can be shortened when there is frequent communication with its neighbors as less additional listening for a base beacon is required then (see Section 4.15.1). However, a close analysis of aforementioned values, divided into subsequent steps of the protocol indicates that listening times can be, in the future, decreased even further by improving how the implementation handles the end of both operations. Moreover, it may be worth to reimplement the beginning of the process of receiving a data frame, introducing the radio's command scheduling. It is projected that such changes should altogether additionally improve the activity durations.

Note that CherryRiMAC does not activate the radio when no sending or receiving is requested. It also does not limit the number of frames transmitted during one cycle. Therefore, energy consumption is directly correlated with the volume of traffic in the network: power demand is low when the node is idle and increases as the device needs to handle increasing traffic.

Chapter 5

Evaluation

The implementation of CherryRiMAC has been preliminary tested on actual low-power wireless devices. The aim of the evaluation is threefold. First, it is tested whether all design goals have been met with respect to performance and whether the theoretical assumptions incorporated into the design hold in the real world, so that the protocol operates correctly. Second, the quality of the implementation itself is evaluated for the targeted devices. Third, possible future improvements, enhancing the usability of the protocol, are identified.

5.1. CherryMote device

The evaluation employed CherryMote devices (see Figure 5.1), developed within the HENI project. Each of them consists of a Texas Instruments CC2650 Evaluation Module Kit [1] (the

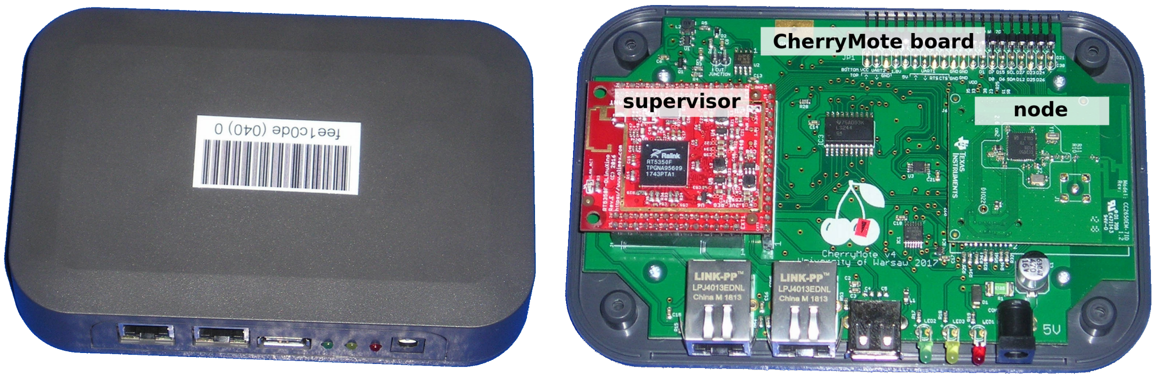


Figure 5.1: CherryMote device

small green board on the right side in Figure 5.1) which is a node of the low-power wireless network running the tested MAC protocol. The CherryMote device includes also an Olimex RT5350F-OLinuXino [44] (the red board on the left side in Figure 5.1) running a HENI's fork [45] of the Linux based OpenWrt operating system [46]. This single board computer is called *supervisor* as its main task is to program the node with an intended application and collect logs generated during an experiment. Both components of the CherryMote device are connected with each other through a CherryMote board (the largest green board in the Figure 5.1), which additionally provides them with a power supply and eases access to different ports of the device for an operator.

The evaluation of CherryRiMAC has been performed on a network consisting of only four such devices since they are still being developed and only a few prototypes have been produced thus far. However, the prepared test scenarios should provide reliable feedback on the correctness and performance of the newly created MAC protocol. Three of the used CherryMote devices (named A, B and C) are the second version prototypes and one (named D) is the fourth prototype version. The most significant difference between them, from the testing perspective is that the low-power nodes for the newer series were produced by another company than the previous ones. Therefore, device D may be built from components supplied by other manufacturers than the older three devices. As a result, it may have non-identical characteristics: for example, a slightly different clock rate (see Section 5.11 for actual measurements). Although the available prototypes do not include improvements developed for their final version, none of known problems should influence the results of the evaluation.

5.2. Test method

During the tests the devices were placed on the same level above the ground within each other's radio range, without any obstacles between them. Different scenarios were realized by specially prepared whip6 applications, which used the wireless communication in an intended way. Note that no additional modifications of the CherryRiMAC implementation were required to perform these evaluations. The prepared test programs were loaded on the nodes remotely through supervisors, which were also collecting logs and statistics during each experiment. This information was uploaded to a server via the supervisors' Wi-Fi connections and analyzed afterward.

The number and types of occurring transmission errors were monitored by analyzing the counters that are passed to CherryRiMAC together with each data frame (see Section 4.8). If not specified differently, the default values for the counters were used (i.e., 5 attempts for `noroute` and `noack` counters, 3 for `overtake` and 10 for the `consider` counter). The transmit and the receive queues were able to store during a test up to 5 outgoing frames and 5 buffers for incoming frames, respectively.

The transmissions times were measured within a test application by calculating how much time had elapsed since the `startSendingFrame()` command was invoked until the `frameSendingFinished()` event was signaled. All latencies presented in this chapter concern only successfully sent frames.

All data frames transmitted during tests included, as a part of their payload, a consecutive number of the frame. Thereby, a recipient was able to track which frames had arrived and detect any multiple receptions of the same frame. A full data frame was 125 bytes long in total and allowed for 104 bytes of payload.

After all nodes used in a particular experiment had been programmed with an intended application, they were reset one by one to initiate the start of the test. In its first phase, all devices were performing Neighbors Scan and the intended evaluation scenario was launched directly afterward. At the end, detailed information concerning all data frame passed to be sent and all received frames was appended to the experiment log.

Each test scenario analyzed in this chapter presents statistic of an actual transmission logged during the evaluation process. Note that the behavior of nodes is not fully deterministic, partly because devices were not synchronized precisely for each test, partly since the medium was not isolated from external interference.

5.3. Single flow

To verify that CherryRiMAC is able to reliably send a data frame and that the implementation does not include any obvious bugs, a simple test was conducted initially: one-way transmissions between two devices.

Device A acting as the sender and device B as the receiver were placed about 75 centimeters apart, without any obstacles between them (Figure 5.2 illustrates this setup). After executing

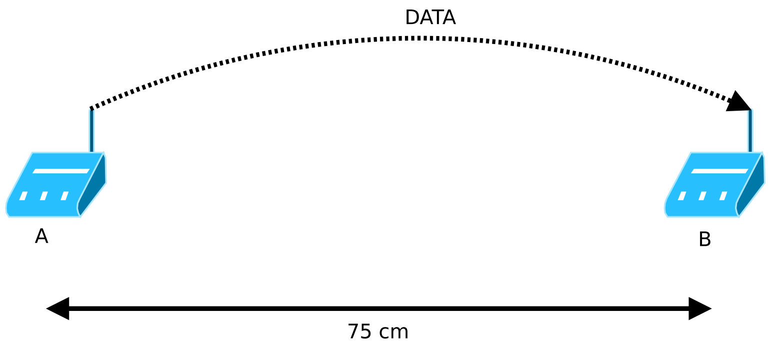


Figure 5.2: Single flow: Test setup

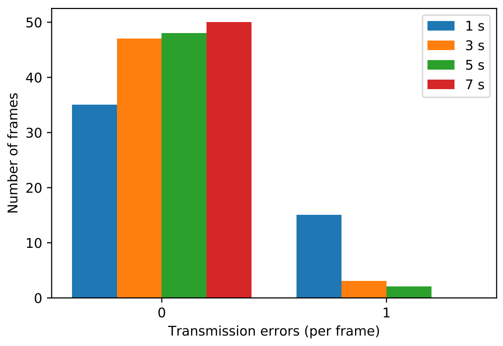
the Neighbors Scan phase, the test application worked in a cycle which lasted twice the duration of the configured CherryRiMAC interval (the test was repeated for all available cycle durations). Each time the sender initiated sending a full data frame, the receiver issued a single receive operation. The test ended when 50 transmissions were reported to be completed.

As presented in Table 5.1, in each setting, all 50 frames were signaled to be successfully transmitted and they all were actually received by the other device. No data frame was delivered to the receiver more than once. An average transmission lasted below the set cycle length: around 60% of the interval duration (except the 1 second cycle).

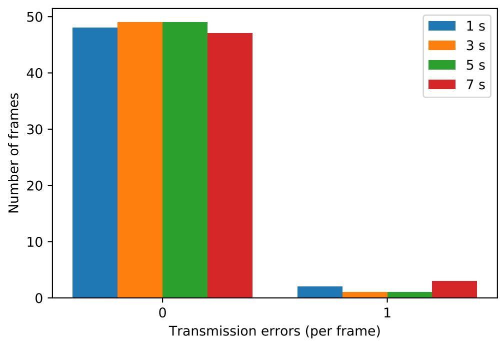
CherryRiMAC cycle	1 s	3 s	5 s	7 s
Test cycle	2 s	6 s	10 s	14 s
Issued transmissions/receptions	50	50	50	50
Frames sent successfully	50	50	50	50
Frames received successfully	50	50	50	50
Frames received multiple times	0	0	0	0
Average transmission time	0.85 s	1.74 s	2.96 s	3.86 s

Table 5.1: Single flow: Test summary

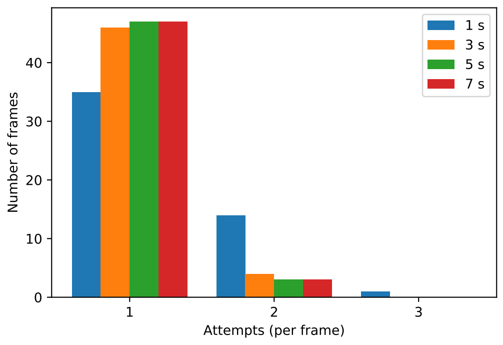
More detailed results, presented in Figure 5.3, indicate that almost all frames were sent in the first attempt. The higher average latency and the higher rate of `NOROUTE` errors, especially during the test with 1 second CherryRiMAC interval, are an effect of lack of synchronization of the nodes. The sender was ready to transmit a data frame before the receiver initiated the listening operation in the same test cycle. In effect, the receiver was not able to get a frame during the sender's first transmission attempt. Additionally, a few data frames did not arrived successfully at their destination resulting in `NOACK` errors. However, CherryRiMAC includes retransmission mechanisms, which were activated in those situations, and thus all data frames were eventually successfully transmitted. 90% of them were sent below 1.3 s, 3 s, 5 s and 7 s for 1 s, 3 s, 5 s and 7 s CherryRiMAC interval respectively.



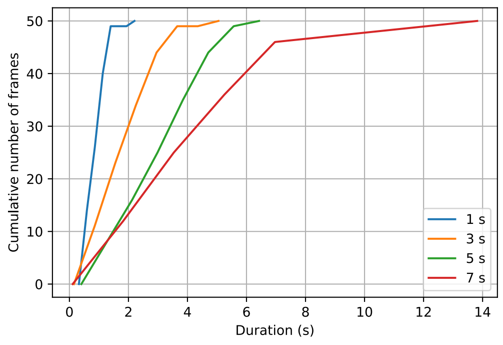
(a) `NOUROUTE` transmission errors



(b) `NOACK` transmission errors



(c) Attempts to transmit a frame



(d) Transmission time

Figure 5.3: Single flow: Test results

5.4. Single flow, multiple transmissions

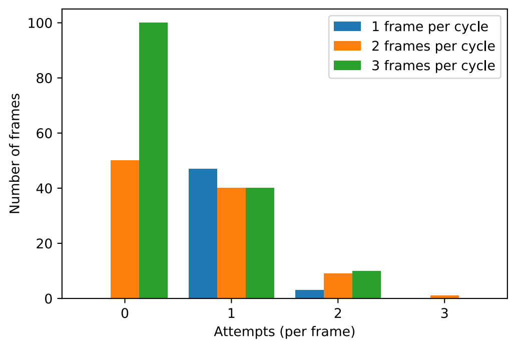
CherryRiMAC allows a node to send its data frame as a reply to both a base beacon and an ack beacon. As a result, multiple frames can be transmitted in a single cycle. To verify this functionality, a test with the same setup as the one described in the previous Section 5.3 was performed, but this time in each cycle the sender issued multiple transmissions. Similarly, the receiver each time passed multiple buffers for incoming data frames. The CherryRiMAC cycle duration was set to 5 seconds.

In the compared scenarios 1, 2 or 3 transmissions were issued, all at once in each test cycle. In effect, 50, 100 and 150 data frames in total were sent, respectively. Table 5.2 summarizes the test. In all cases, all frames were successfully transmitted and received. As presented in Figure 5.4a, data frames were actually sent as a response to an ack beacon: exactly 50 in the test with double transmissions and 100 in the test with triple. Figure 5.4b indicates that in the scenarios with multiple transmissions latencies for most of the frames were only slightly higher than in the one with single ones. The values were not the same since sending the second (and the third) frame in the row must last longer than the first one and a failure to send the first frame results in higher latency for the succeeding one.

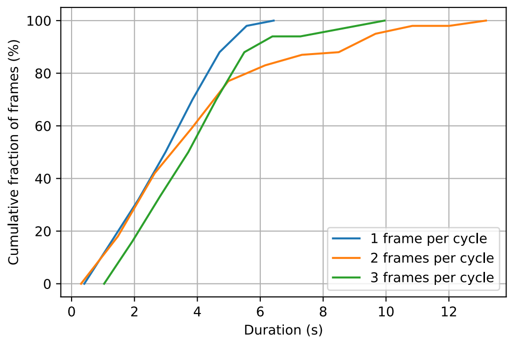
Note that the ability to transmit multiple data frames in one cycle greatly increases the potential throughput of a network. In practice, because a single transmission takes as little as 15 ms, the number of frames that can be sent during a single cycle is limited by the number of available free buffers on the receiver's side.

Transmissions per test cycle	1	2	3
CherryRiMAC cycle	5 s	5 s	5 s
Test cycle	10 s	10 s	10 s
Issued transmissions/receptions	50	100	150
Frames sent successfully	50	100	150
Frames received successfully	50	100	150
Frames received multiple times	0	0	0
Average transmission time	2.96 s	3.83 s	3.80 s
Test duration	508 s	586 s	508 s

Table 5.2: Single flow, multiple transmissions: Test summary



(a) Attempts to transmit a frame. 0 means that a frame was transmitted as a reply to an ack beacon.



(b) Transmission time

Figure 5.4: Single flow, multiple transmissions: Test results

5.5. Single flow with interference

The protocol was designed and implemented having in mind that it can be used in a deployment where there are also other active wireless networks within radio range. To verify whether CherryRiMAC tolerates such interference, a test similar to the one described in Section 5.3 was prepared, but a third device was introduced: it transmitted regardless of medium activity a data frame intended for itself each 3 to 6 seconds (the next interval was randomized after every transmission). The setup is illustrated in Figure 5.5.

The results of this test are summarized in Table 5.3 and Figure 5.6, and compared with a scenario without the third interfering device. An average latency and the total number of errors are noticeably higher. Moreover, one frame was not sent at all and one frame was received twice. An analysis of the experiment logs indicates that the following situation took place: after the receiver got a data frame, it transmitted an acknowledgment, as usual. However, the ack beacon did not arrived successfully at the sender: the CRC verification dropped the frame. In effect, the node transmitted this data frame one more time in the next cycle, and thus the receiver got it the second time. Note that in this experiment the receiver passed to CherryRiMAC exactly 50 buffers, only one in each test cycle. Therefore, since one frame was transmitted twice, the receiver did not have a free buffer for the last data frame. As a result, it did not transmit more base beacons and thus only 49 data frames were sent and received.

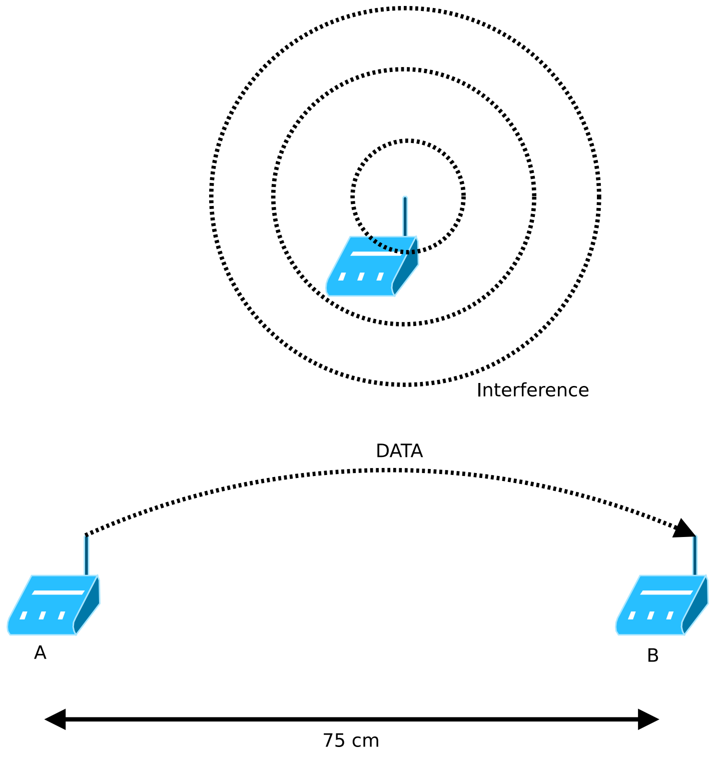


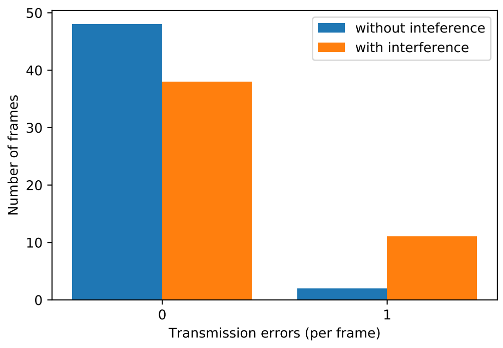
Figure 5.5: Single flow with interference: Test setup

The initiation of only one receiving operation per cycle by the node resulted also in the higher number of `NOACK` errors. Although the sender had two data frames prepared to be sent and actually transmitted the second one as a reply to an ack beacon, the receiver was not able to get it. Therefore, it is advisable for a receiver to provide at least one free buffer more than compared to the expected incoming frames.

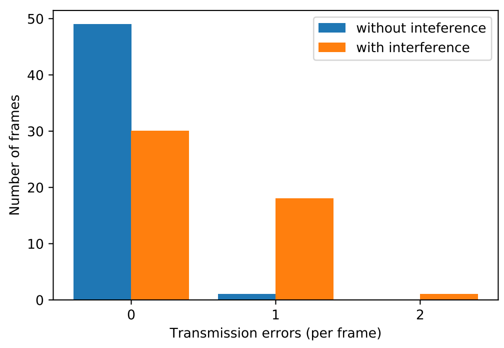
Interference	not present	present
CherryRiMAC cycle	5 s	5 s
Test cycle	10 s	10 s
Issued transmissions/receptions	50	50
Frames sent successfully	50	49
Frames received successfully	50	49
Frames received multiple times	0	1
Average transmission time	2.96 s	5.7 s
Test duration	508 s	508 s

Table 5.3: Single flow with interference: Test summary

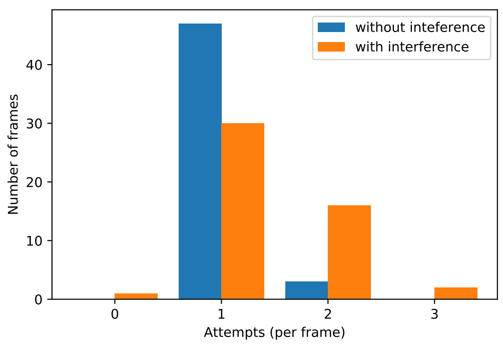
Nevertheless, it seems that CherryRiMAC can be safely used in deployments where it can interfere with other networks: in the test, not only did no abnormal behaviors take place (particularly, the node did not panicked), but also the device was still able to perform reliable transmissions. Furthermore, neither of nodes reported misguidedly a reception or a transmission because of the frame sent by the third device.



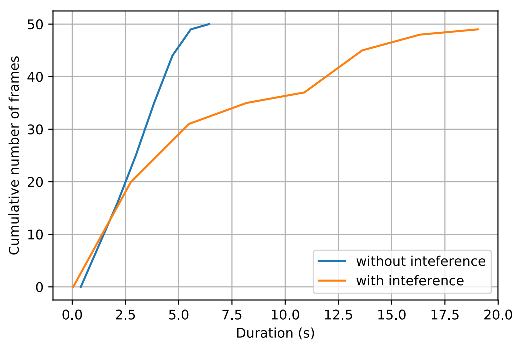
(a) `NOUROUTE` transmission errors



(b) `NOACK` transmission errors



(c) Attempts to transmit a frame. 0 means that a frame was transmitted as a reply to an ack beacon.



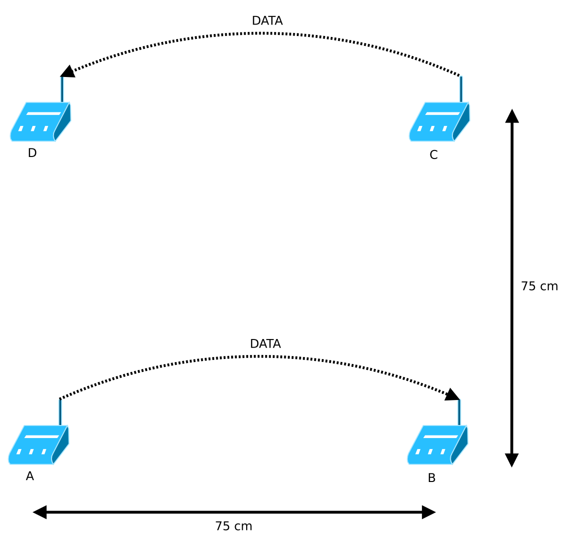
(d) Transmission time

Figure 5.6: Single flow with interference: Test results

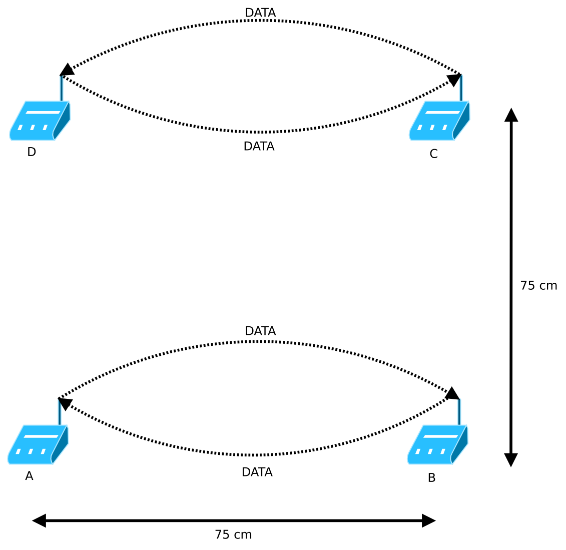
5.6. Multiple flows

Since CherryRiMAC was showed to be able to perform reliable transmissions between two devices, more realistic scenarios were tested, in which there was more than one flow of exchanged messages. To emulate such situations, all four available devices were used. They were placed as illustrated in Figure 5.7, all within of each other's radio range. All nodes had their CherryRiMAC cycle durations set to 5 seconds. Senders executed a similar program as in the previous tests: every 10 seconds they sent one data frame to their receiver. All data frames were filled with half the maximal possible amount of data. However, this time the receivers passed 2 buffers (to be able to use transmissions in replies to ack beacons) at the start of the test and each time a frame was received, a next free buffer was passed to CherryRiMAC: almost always two free buffers were available for the radio. Two setups were tested (see Figure 5.7). In the first one there were two flows of messages (device A to B and C to D), in the second one there were four flows (device A to B and B to A as well as C to D and D to C). In the latter case all devices executed simultaneously both the receiver's and the sender's programs. Each flow was independent from each other.

The results of the performed tests are compared in Table 5.4 and in Figure 5.8. In all cases all transmitted data frames were sent successfully and each one was received exactly once. The more flows, the more `NOROUTE` and `ENOACK` errors occurred (compare these values also with the single flow test presented in Section 5.3) and the greater the transmission times. Note that when a device acts both as a sender and as a receiver (i.e., the 4 flows test in



(a) 2 flows



(b) 4 flows

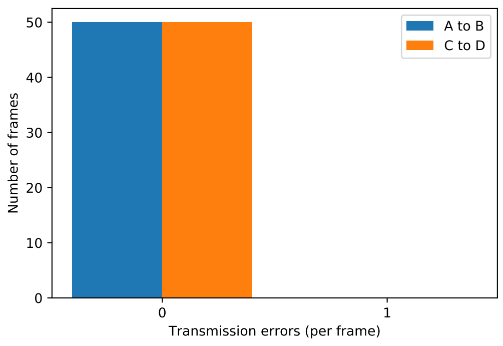
Figure 5.7: Multiple flows: Test setup

Figure 5.7a), most of measured latencies are above the cycle duration (i.e., 5 seconds in this test) since the node not always decides to perform a transmission in the earliest possible moment but chooses instead to send its own base beacon and listen for incoming frames.

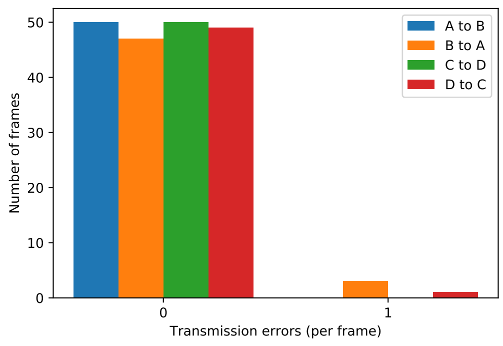
Although the dispatcher (see Section 4.11) does not implement a fair algorithm, no starvation occurred: no frames were considered to be transmitted more than 3 times.

Number of flows	2 flows	4 flows
CherryRiMAC cycle	5 s	5 s
Test cycle	10 s	10 s
Issued transmissions in each flow	50	50
Issued transmissions per test cycle	1	1
Free buffers for incoming frames	2	2
Flows	*A* to *B*, *C* to *D*	*A* to *B*, *B* to *A*, *C* to *D*, *D* to *C*
Frames sent successfully (per flow)	50, 50	50, 50, 50, 50
Frames received successfully (per flow)	50, 50	50, 50, 50, 50
Frames received multiple times (per flow)	0, 0	0, 0, 0, 0
Average transmission time (per flow)	3.39 s, 2.74 s	6.78 s, 4.17 s, 5.85 s, 4.72 s

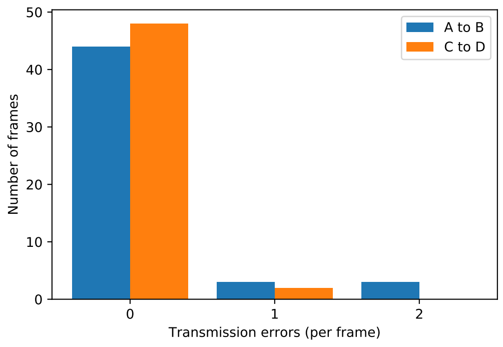
Table 5.4: Multiple flows: Test summary



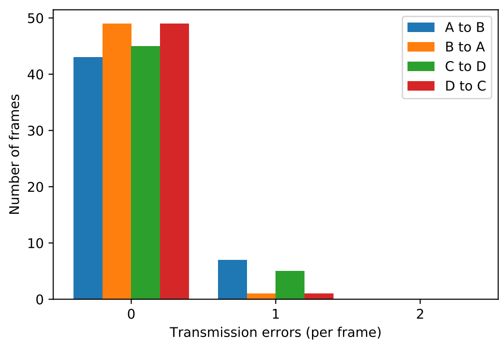
(a) 2 flows: `NOUROUTE` transmission errors



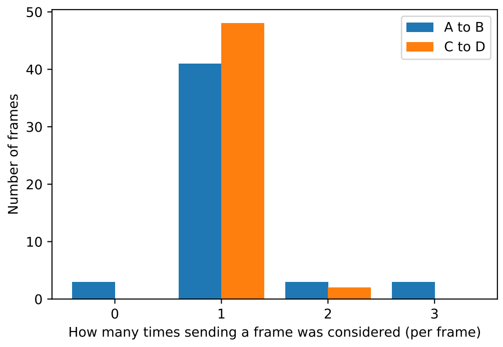
(b) 4 flows: `NOUROUTE` transmission errors



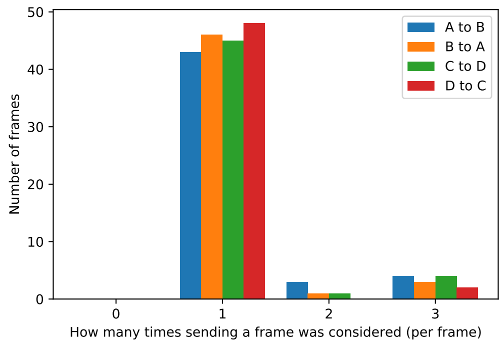
(c) 2 flows: `NOACK` transmission errors



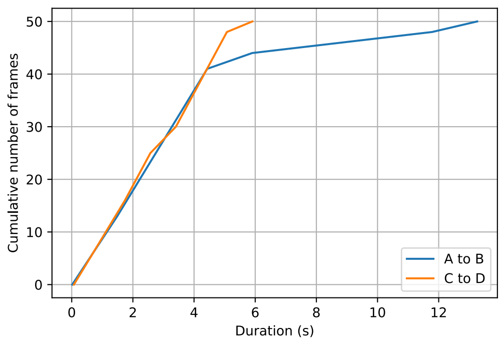
(d) 4 flows: `NOACK` transmission errors



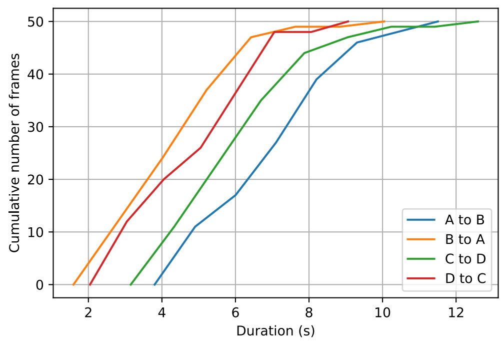
(e) 2 flows: How many times a frame transmission was considered by the dispatcher



(f) 4 flows: How many times a frame transmission was considered by the dispatcher



(g) 2 flows: Transmission time



(h) 4 flows: Transmission time

Figure 5.8: Multiple flows: Test results

5.7. Multiple flows, congested medium

Having four devices it is impossible to test the protocol in a dense network. Therefore, to study how CherryRiMAC copes within a congested medium, another test with two and four flows was prepared. Devices were placed as before (see Figure 5.7). However, this time the CherryRiMAC cycle length was set to 1 second and a new half-full data frame was generated each 1 and 2 seconds for setup with 2 and 4 flows respectively.

Table 5.5, Figure 5.9 and Figure 5.10 illustrate the results.

Number of flows	2 flows	4 flows
CherryRiMAC cycle	1 s	1 s
Test cycle	1 s	2 s
Issued transmissions in each flow	50	50
Issued transmissions per test cycle	1	1
Free buffers for incoming frames	2	2
Flows	A to B, C to D	A to B, B to A, C to D, D to C
Frames sent successfully (per flow)	50, 50	0, 50, 50, 15
Frames received successfully (per flow)	50, 50	20, 50, 50, 18
Frames received multiple times (per flow)	0, 0	20, 0, 0, 1
Average transmission time (per flow)	0.55 s, 0.57 s	–, 1 s, 0.62 s, 7.89 s

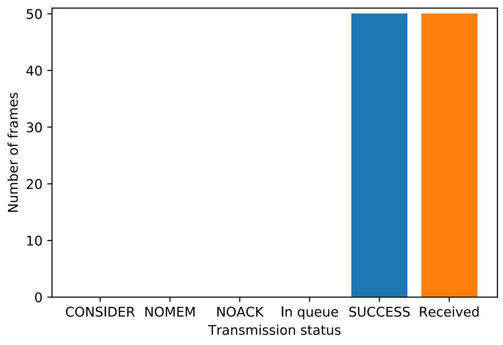
Table 5.5: Multiple flows: Test summary

When two flows were present, all data frames were delivered successfully and around half of these transmissions ended under 0.5 second. Each frame was received by its receiver exactly once. However, when four flows were introduced (and despite extending the test cycle to 2 seconds) only two flows managed to successfully exchange all messages. Especially peculiar are statistics for the flow from device A to device B, since none of these data frames was reported by the sender to be successfully transmitted whereas the receiver got 20 frames intended for it, each one multiple times. A close analysis of collected logs indicates that a coincidental synchronization of devices was the root of the situation: node B transmitted its base beacon and it was successfully got by node A. Therefore, the sender replied to it with its data frame which was also successfully received and thus device B prepared an ack beacon. However, just a moment before it transmitted the acknowledgment, node C woke up and sent its base beacon. Therefore, since the first frame that arrived at device A, after it had transmitted its data, was the base beacon from node C (not the ack beacon from device B), it concluded that its data frame had not been received (a `NOACK` error). When device C transmitted its base beacon and initiated listening for incoming data frames, before node D managed to send its data frame the ack beacon from device B was received. In effect, node C concluded that there were not any frames intended for it and finished listening without getting the data frame sent to it by device D. Note that all nodes were executing the same program with the same interval duration, thereby the aforementioned situation was repeated in consecutive cycles.

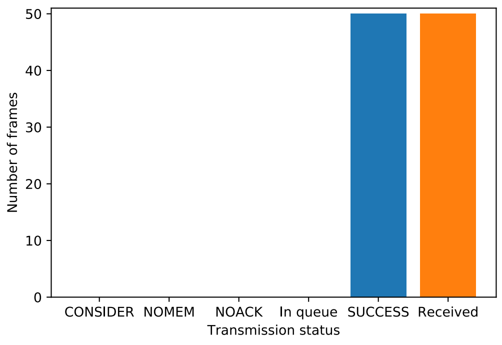
Problems of devices A and D with sending their data frames resulted in various transmission failures: since they were trying to retransmit frames, there was no place left in their transmit queues for succeeding ones, thus multiple `NOMEM` errors occurred. Moreover, some data frames were still in queues when the test ended 6 seconds after the last scheduled transmission. One frame (in device D) reached the maximal limit on the number of times it could be considered by the dispatcher.

As a future improvement counteracting such situations, it should be considered whether to implement choosing a device's cycle in an active way: not randomizing its initial time but monitoring a medium and selecting a moment when no transmissions are in progress. It is also worth analyzing whether before sending its base beacon the node should perform CCA and skip (postpone?) its transmission to avoid a potential collision. Currently, a higher network layer should monitor delivery rate and issue reinitialization of the node's cycle manually when the performance is lower than expected.

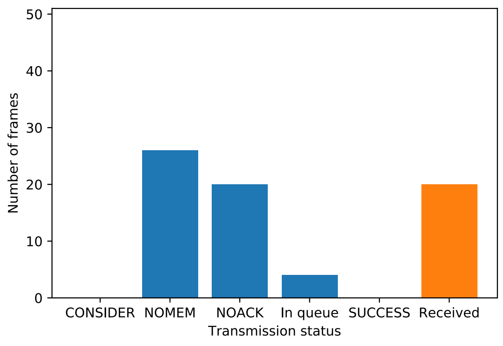
Note the robustness of the design. Despite so many unsuccessful transmissions to devices B and C, in the other two flows data frames were successfully communicated with latencies around 1 second. The same devices were involved in both successful and unsuccessful message exchanges.



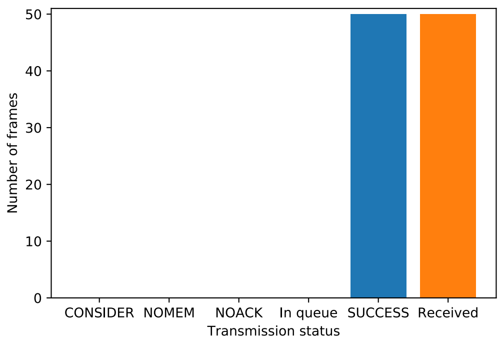
(a) 2 flows: A to B



(b) 2 flows: C to D

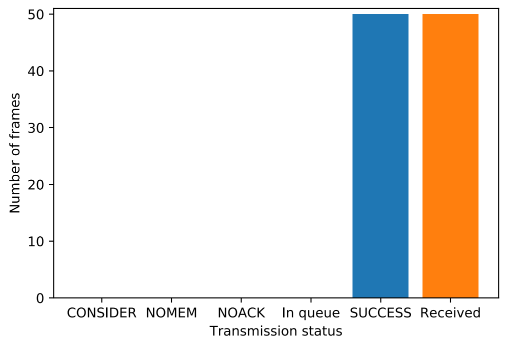


(c) 4 flows: A to B

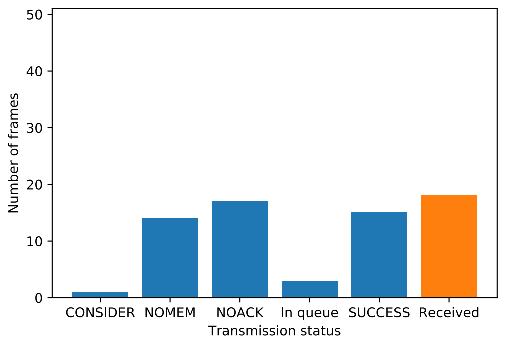


(d) 4 flows: B to A

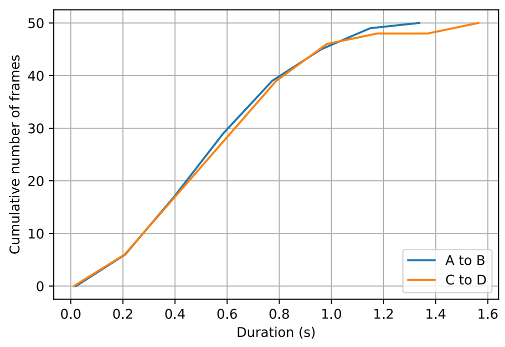
Figure 5.9: Multiple flows: Test results (part 1)



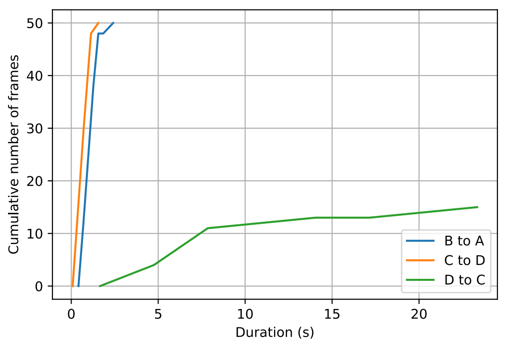
(a) 4 flows: C to D



(b) 4 flows: D to C



(c) 2 flows: Transmission time



(d) 4 flows: Transmission time

Figure 5.10: Multiple flows: Test results (part 2)

5.8. Broadcast

To evaluate the best-effort approach to broadcasting data frames implemented in CherryRiMAC, all four available devices were placed as illustrated in Figure 5.11 and four different scenarios were prepared. All devices were receivers (with 3 buffers) and one, two, three or four of them were also broadcasting their frames. CherryRiMAC cycle duration was set to 5 seconds and each broadcaster initiated transmissions of five half-full frames in 30-second intervals. Note that in cases with multiple broadcasters, they all were initiating the broadcast operation simultaneously since all devices executed the same test program. As a comparison, an all-to-all unicast transmissions scenario was also prepared.

The results of the broadcast test are summarized and compared to the all-to-all unicast transmissions in Table 5.6 and 5.7.

The best-effort approach performed the better, the fewer nodes were broadcasting simultaneously. 100%, 63%, 49% and 40% of data frames (counting in total) were delivered where there were one, two, three and four broadcasters respectively. In fact, presumably most of the transmissions with multiple senders were successful due to the capture effect and small timing differences. All broadcasters were responding to base beacons with their frames. Therefore, theoretically, collisions should have occurred. Furthermore, since broadcast data frames do not require acknowledgments, a frame could not have been retransmitted in the same cycle.

When using the implemented broadcast it is therefore advisable to randomize its start time so that it is not executed simultaneously by multiple nodes or, at least, it is not parallel to another broadcasting for their entire durations. Future improvements should consider employing

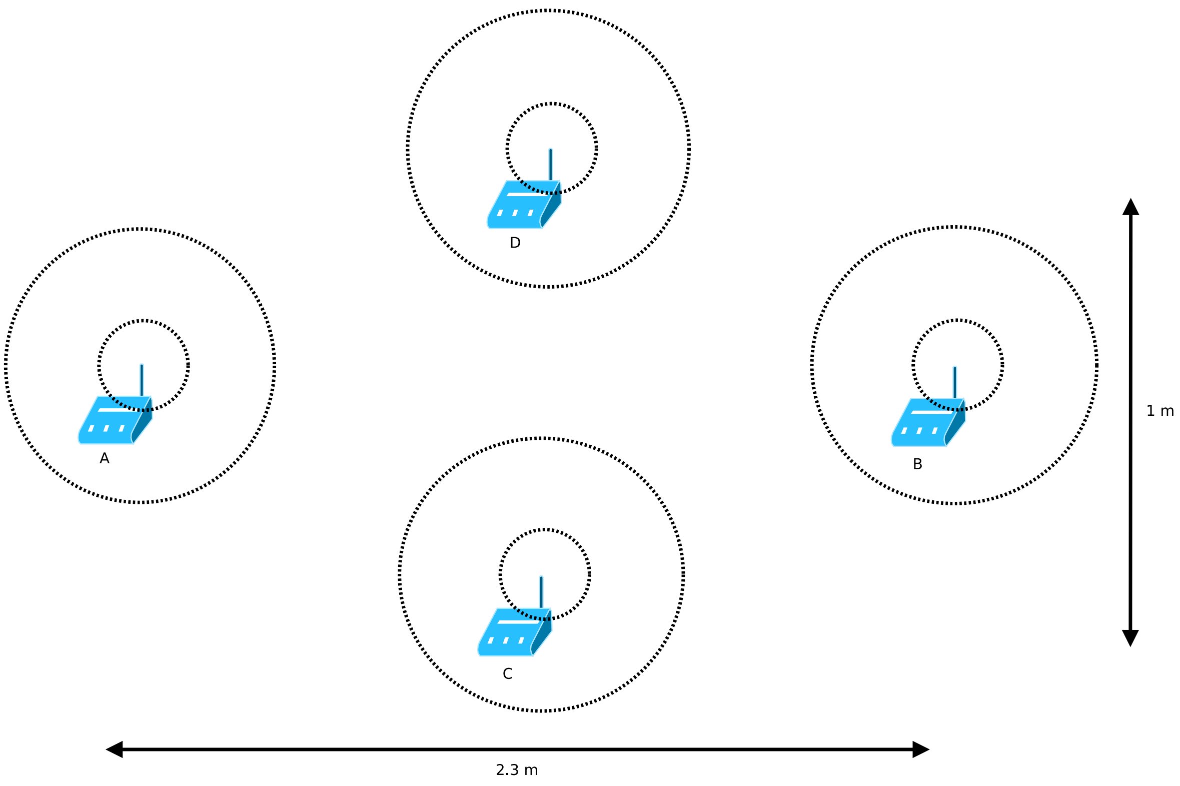


Figure 5.11: Broadcast: Test setup

some mechanisms counteracting potential collisions or managing contending broadcasters.

The unicast transmission, which also does not include any mechanism to deal with collisions, turned out to perform surprisingly well during all to all message exchanges: it completed 83% of all sending operations with success. All delivered data frames were received by their recipients exactly once. Therefore, a `NOACK` error means that a transmitted data frame did not reach its destination because of a collisions or a reception of a contending frame (not that the acknowledgment was lost). However, most of the data frames were actually delivered to their recipients due to the capture effect combined with retransmission mechanisms included in CherryRiMAC.

Number of broadcaster	1 broadcaster	2 broadcasters	3 broadcasters	4 broadcasters	4 senders
Transmission method	broadcast	broadcast	broadcast	broadcast	unicast
CherryRiMAC cycle	5 s	5 s	5 s	5 s	5 s
Test cycle	30 s	30 s	30 s	30 s	30 s
Issued transmissions in each flow	5	5	5	5	5 (per recipient)
Issued transmissions per test cycle	1	1	1	1	1 (per recipient)
Free buffers for incoming frames	3	3	3	3	3
Broadcasters/Senders	*A*	*A*, *B*	*A*, *B*, *C*	*A*, *B*, *C*, *D*	*A*, *B*, *C*, *D*
Frames received successfully / possible unique receptions (in total)	15 / 15	19 / 30	22 / 45	24 / 60	50 / 60

Table 5.6: Broadcast: Test summary

		Broadcaster
		A
Receiver	*A*	
	B	5 / 5
	C	5 / 5
	D	5 / 5

(a) 1 broadcaster: Number of received unique frames

		Broadcaster	
		A	*B*
Receiver	*A*		3 / 5
	B	2 / 5	
	C	5 / 5	2 / 5
	D	5 / 5	2 / 5

(b) 2 broadcaster: Number of received unique frames

		Broadcaster		
		A	*B*	*C*
Receiver	*A*		2 / 5	4 / 5
	B	4 / 5		2 / 5
	C	5 / 5	0 / 5	
	D	5 / 5	0 / 5	0 / 5

(c) 3 broadcaster: Number of received unique frames

		Broadcaster			
		A	*B*	*C*	*D*
Receiver	*A*		1 / 5	2 / 5	2 / 5
	B	0 / 5		4 / 5	0 / 5
	C	5 / 5	1 / 5		3 / 5
	D	5 / 5	0 / 5	1 / 5	

(d) 4 broadcaster: Number of received unique frames

		Sender			
		A	*B*	*C*	*D*
Receiver	*A*		4 / 5	4 / 5	5 / 5
	B	5 / 5		4 / 5	4 / 5
	C	3 / 5	3 / 5		3 / 5
	D	5 / 5	5 / 5	5 / 5	

(e) 4 unicast senders: Number of received unique frames

		Sender			
		A	*B*	*C*	*D*
Receiver	*A*		1.4	2.6	1.4
	B	1.8		2.8	1.6
	C	1.4	1.8		0
	D	1.6	2.4	2.2	

(f) 4 unicast senders: Average number of `NOACK` errors (limit: 5)

Table 5.7: Broadcast: Test results

5.9. Always Listen mode

The Always Listen mode in CherryRiMAC aims to take advantage of the unlimited power supply available to some nodes in a network by enabling extra listening on them: it should increase the potential throughout which can be handled by these devices. The mode is specially designed to be used on a node that acts as the network gateway. In other words, all other devices transmit their data frames to it when they want to send a message beyond the low-power wireless network.

To simulate such a situation a setup as presented in Figure 5.12 was prepared. Device A

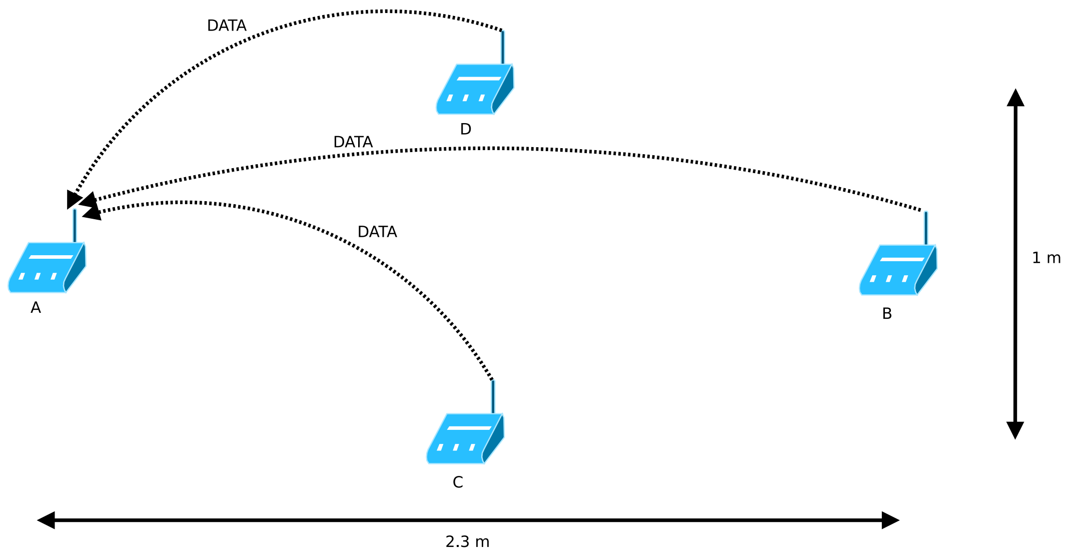


Figure 5.12: Always Listen mode: Test setup

played the role of the gateway device to which all other nodes sent 50 half-full data frames, each with an interval of 10 seconds between subsequent transmissions. The CherryRiMAC cycle duration was set to 5 seconds on all devices and the test compared scenarios in which node A had and had not the Always Listen mode enabled. In the scenario with the mode enabled, device A transmitted its standard cycle length during the Neighbors Scan phase of the test and activated the extra listening afterward. Therefore, first transmissions to it were those standard ones and only after receiving its base beacon other devices discovered that the Always Listen mode was enabled on node A.

The test results are summarized in Table 5.8. Figure 5.13 presents in turn the results with the Always Listen mode disabled on device A, while Figure 5.14 and Figure 5.14 illustrate the results when node A had the extra listening active.

Without the Always Listen mode enabled on the gateway device, only 30% of transmitted data frames reached the destination successfully. The reason for that were collisions which occurred because all three devices were sending simultaneously their frames in reply to node A's base beacons. Since devices were trying to retransmit those not acknowledged data frames, their transmit queues quickly became full and thus subsequent frames could not even be passed to CherryRiMAC. As a result, successfully performed transmissions had latencies even above 1 minute.

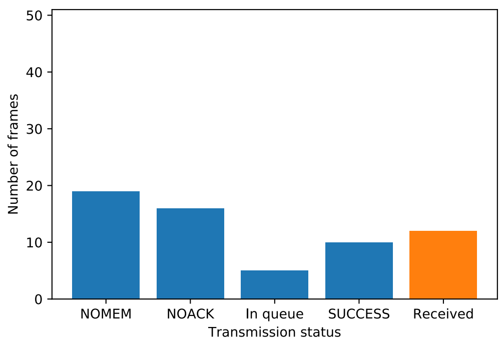
Activating the mode improved the situation a lot: in the optimistic scenario only the first frame from each device was not delivered (the standard, not the immediate transmission was used to send it) and average latencies were below 0.01 of a second. This indicates how

Always Listen mode	disabled	enabled	enabled
Synchronized senders	none	B and C	none
Always listening node	none	A	A
CherryRiMAC cycle	5 s	5 s	5 s
Test cycle	10 s	10 s	10 s
Issued transmissions in each flow	50	50	50
Issued transmissions per test cycle	1	1	1
Free buffers for incoming frames	3	3	3
Flows destination	B to A, C to A, D to A	B to A, C to A, D to A	B to A, C to A, D to A
Frames sent successfully (per flow)	10, 23, 9	14, 14, 49	49, 49, 49
Frames received successfully (per flow)	12, 24, 9	14, 14, 49	49, 49, 49
Average transmission duration (per flow)	39.86 s, 29.24 s, 36.97 s	0.035 s, 0.009, 0.009 s	0.035 s, 0.009, 0.010 s

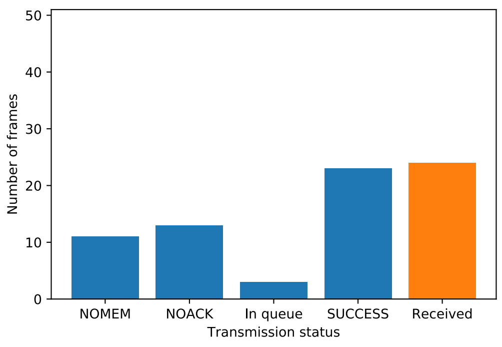
Table 5.8: Always Listen mode: Test summary

significantly the Always Listen mode can increase the potential throughput of a gateway and reduce the sender activity time (thereby decreasing power consumption).

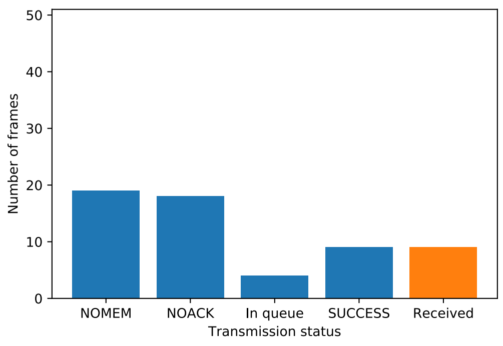
However, it may also happen, as during the pessimistic scenario with the Always Listen mode enabled, that two senders are coincidentally synchronized with each other. It this case, nodes B and C were executing the test program almost in parallel. In effect, they were sending their data frames simultaneously. As a result, collisions occurred and neither of frames was received by node A. Retransmissions also did not help since they also were performed simultaneously and only some data frames were delivered successfully at the end of the test when both devices got a bit out of sync. In contrast, transmissions from node C did not overlap with other flows. Therefore, they were successful. As a future improvement, it should be analyzed whether frames that can be sent immediately should be preceded with a random delay and a verification of medium inactivity.



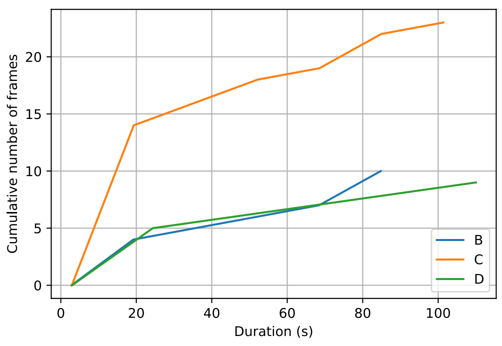
(a) B to A



(b) C to A

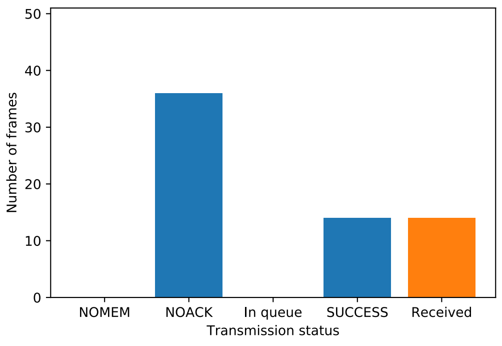


(c) D to A

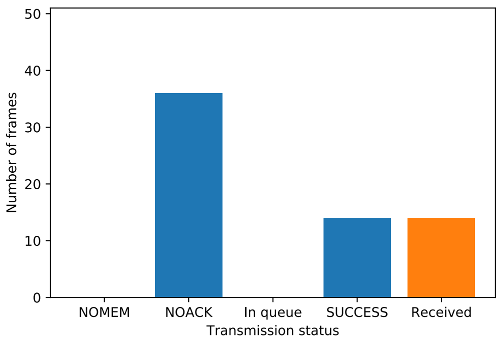


(d) Transmission time

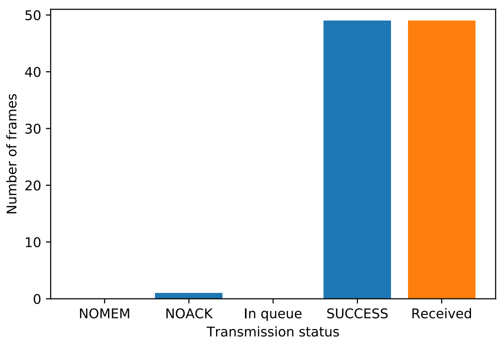
Figure 5.13: Always Listen mode: Test results (Always Listen mode disabled)



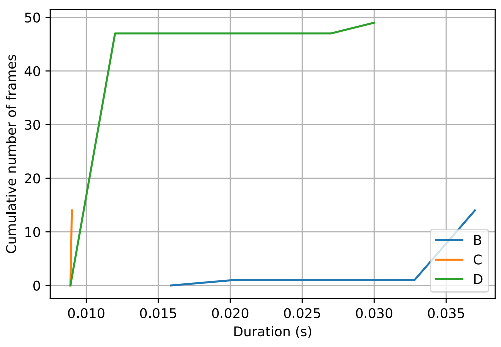
(a) B to A



(b) C to A

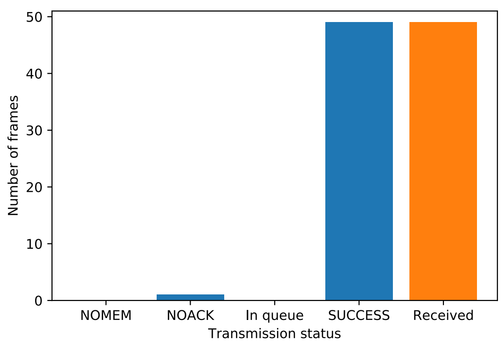


(c) D to A

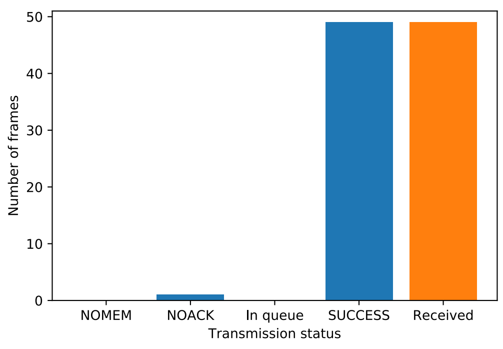


(d) Transmission time

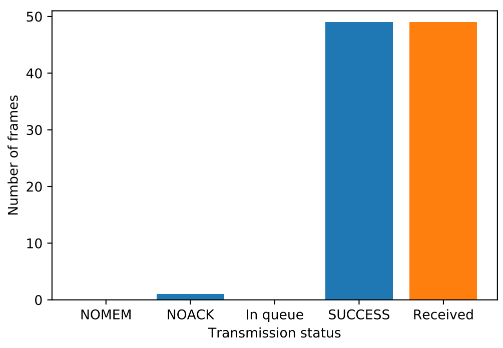
Figure 5.14: Always Listen mode: Test results (Always Listen mode enabled, synchronized senders)



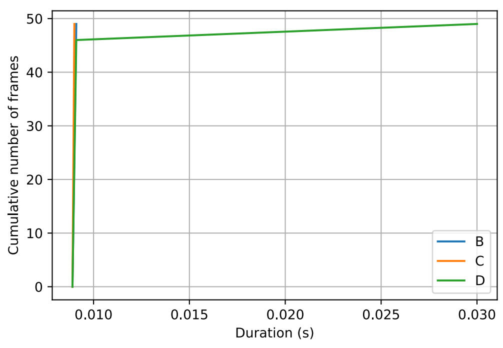
(a) B to A



(b) C to A



(c) D to A



(d) Transmission time

Figure 5.15: Always Listen mode: Test results (Always Listen mode enabled, desynchronized senders)

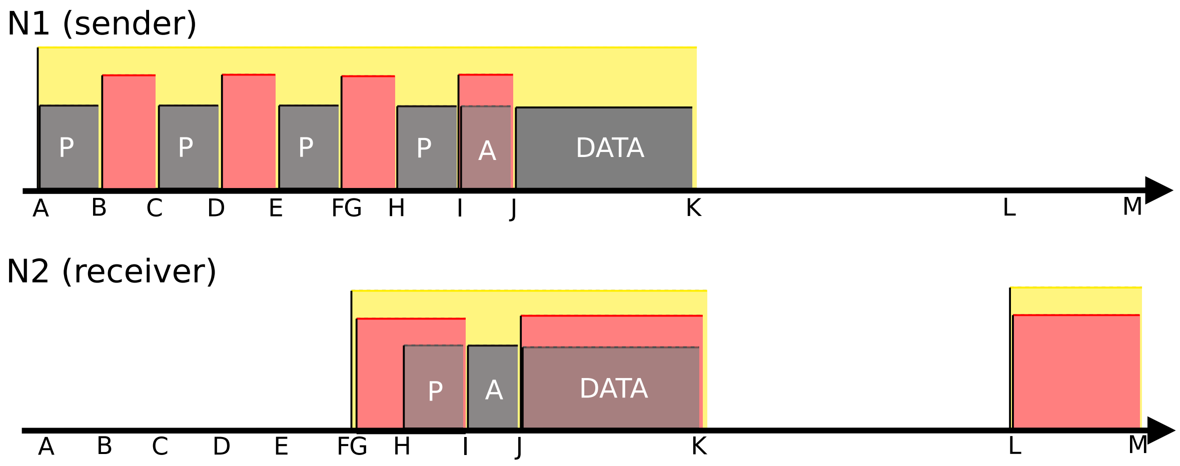
5.10. Comparison with X-MAC

To better assess the performance of CherryRiMAC, it was compared with the X-MAC protocol [12] which had been chosen since it had been the only already available MAC protocol in the whip6 operating system. The following description provides a short overview of the protocol which should help to interpret the results of the tests.

5.10.1. Overview of X-MAC

X-MAC is an asynchronous duty-cycle MAC protocol designed for low-power wireless networks in which a transmission is initiated by the sender. When a node has a data frame ready to be sent, it activates its radio and alternately transmits short frames called *preambles* and listens for incoming frames. Since the preamble includes the address of the data frame's recipient, when the latter wakes up and starts listening, it is able to quickly check whether there are any pending transmissions to it. If it gets a preamble with its address in it, it responds to it with an *acknowledgment*, which informs the sender, when the node gets it, that the intended receiver is ready to receive the data frame. The process of the transmission ends for the sender when it transmits its data frame, whereas the receiver goes to sleep when it gets the frame. The sender's preambles sending should span at least the duration of the receiver's duty cycle. The receiver, after waking up, should listen to the medium longer than a duration between two sender's preambles.

Figure 5.16 illustrates an exemplary transmission performed using the X-MAC protocol. When the sender (node N1) is passed a data frame to be sent at moment A, it transmits its



P – preamble, A – acknowledgment, DATA – data frame
yellow – node is active, red – listening, gray – transmitting

Figure 5.16: X-MAC: Overview

preamble (moment B) and listens to the medium (moment B). Since no frames are received, it repeats these steps (moments C – H). The receiver (node N2) wakes up at moment G and activates listening. When it gets the preamble sent by the receiver (moment H), it replies with an acknowledgment (moment J) and continues listening. The sender transmits its data frame (moment J) as soon as it receives the acknowledgment from the intended receiver (moment I). The sender goes to sleep when this transmission ends, the receiver — when it gets the frame (moment K). Since no preambles are detected when the receiver wakes up in its next cycle (moment L), it concludes that there are no pending transmissions to it and goes back to sleep again (moment M).

5.10.2. X-MAC in whip6

The implementation of X-MAC available in whip6 is a simple one: it accepts only one data frame and one buffer for incoming frame at any given moment. Moreover, it requires that receiving be active when sending is performed and always starts transmitting a frame as soon as one is passed. It does not include any phase awareness mechanisms and ways to deal with contending nodes.

A receiver, to detect a preamble, listens to a medium for 20 milliseconds in each cycle. Preambles are transmitted by a sender with an interval of 10 milliseconds for at least the duration of the receiver's cycle.

Unfortunately, this implementation does not appear to be production-ready since it contains a few ”TODO” comments and, during more complicated test scenarios, a node panicked frequently: it seems that it cannot handle some situations in which not the expected frame is received. However, to be able to perform the comparison, I modified the implementation according the best of my knowledge. It retransmits an acknowledgment when the next preamble is received instead of a data frame and simply continues listening in other unforeseen cases.

5.10.3. Single flow

At first, a simple test with one flow of messages was prepared. A sender (device A) was placed about 75 centimeters away from a receiver (device B). This setup is presented in Figure 5.17.

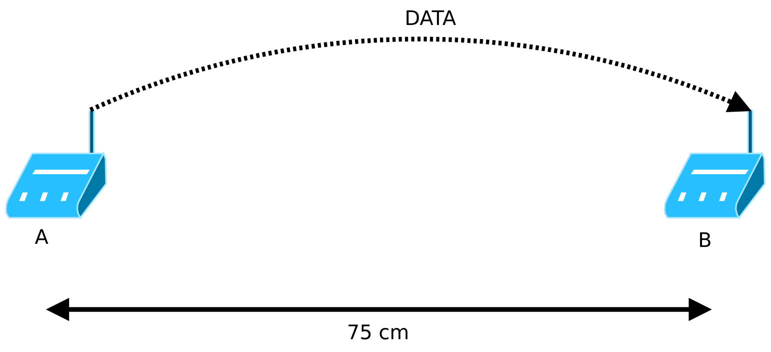


Figure 5.17: Comparison with X-MAC: Test setup (single flow)

50 data frames were sent, one every 2 seconds. Cycle duration of both protocols was set to 1 second. For a fair comparison, data frames passed to CherryRiMAC had counters (see Section 4.8) set so as to make only one attempt to send each frame because X-MAC does not implement retransmissions. Additionally, as it requires that receiving must be active when transmitting, the sender in the test was also passed a free buffer for incoming frames.

To measure how much activity of the radio each of these protocols required, the whip6's `RFCoreRadioPrv` component (which is the driver for the radio) was modified: when the radio was powered up, it saved the current wall-clock time. When the hardware was powered down, in turn, the module was able to calculate how long the radio had been switched on. Cumulative durations were appended to the test's logs.

The results of the comparison are presented in Table 5.9 and Figure 5.18.

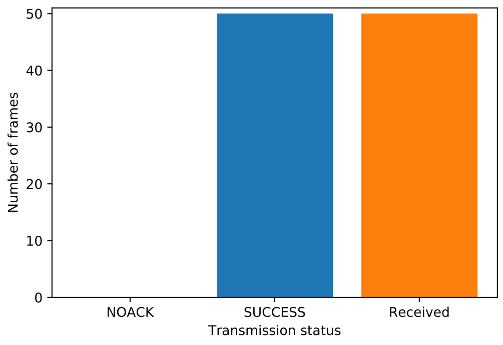
MAC protocol	CherryRiMAC	X-MAC
Protocol cycle	1 s	1 s
Test cycle	2 s	2 s
Flow	A to B	A to B
Issued transmissions	50	50
Issued transmissions per test cycle	1	1
Free buffers for incoming frames	1	1
Frames sent successfully	50	45
Frames received successfully	50	44
Average transmission time	0.87 s	0.46 s
Sender's radio powered up time	1.56 s	24.18 s
Receiver's radio powered up time	0.90 s	2.20 s
Neighbors Scan time	21 s	–

Table 5.9: Comparison with X-MAC: Test summary (single flow)

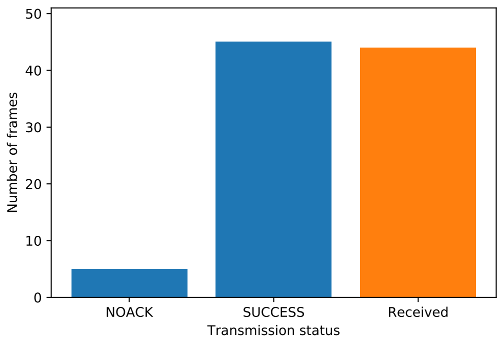
CherryRiMAC managed to deliver all passed data frames whereas X-MAC had six failures. However, more important is the fact that the X-MAC sender reported successful transmission of 45 frames. CherryRiMAC, by using the ack beacon as the acknowledgment of data reception, not as the acknowledgment of readiness to receive data, guarantees more reliable feedback: it signals `SUCCESS` only when the data frame has actually been received by the other device. The design of X-MAC does not offer such a guarantee.

Another significant advantage of CherryRiMAC is how it uses the radio: the hardware was active for almost sixteen times shorter on the sender's side and for almost two and half times on the receiver's. Therefore, a node exchanging messages according to the CherryRiMAC protocol should have much lower energy consumption which results in a longer life time with a limited power supply. This experiment indicates that introducing the phase awareness to the protocol has been a beneficial modification of the original RI-MAC design. Of course, to use it, nodes need to perform Neighbors Scan, which lasts about 21 seconds by default, before any transmission can take place. Contrarily, X-MAC does not requires this phase. However, in most cases the scan can be performed just once at the beginning since information in devices' Neighbors Lists are updated also during regular transmissions and measurements have shown that intervals between message exchanges can be relatively long (see Section 5.11 for exact values).

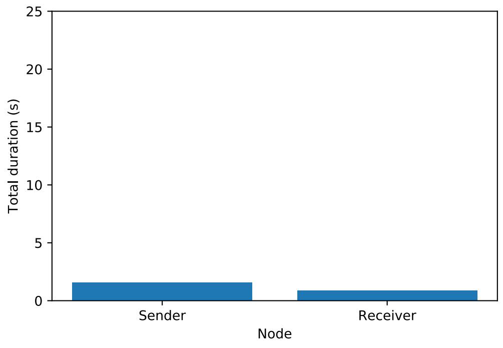
An average CherryRiMAC transmission lasted about two times longer than the duration required by X-MAC to send a data frame. The most likely reason for this it that the X-MAC sender was initiating a transmission of a data frame immediately when it was passed by the test application, whereas the CherryRiMAC sender was also regularly launching the receiving operation.



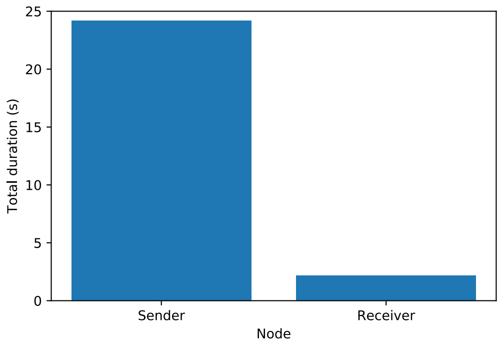
(a) CherryRiMAC: A to B



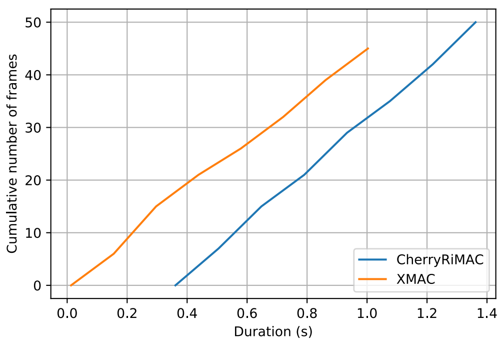
(b) X-MAC: A to B



(c) CherryRiMAC: Radio powered up



(d) X-MAC: Radio powered up



(e) Transmission time

Figure 5.18: Comparison with X-MAC: Test results (single flow)

5.10.4. Two flows

The two protocols were also compared in a test with two independent flows of data between four devices. The setup is presented in Figure 5.19. The senders were transmitting 50 data frames, with 2 second intervals. Both protocols had their cycle lengths set to 1 second. CherryRiMAC was not performing retransmissions.

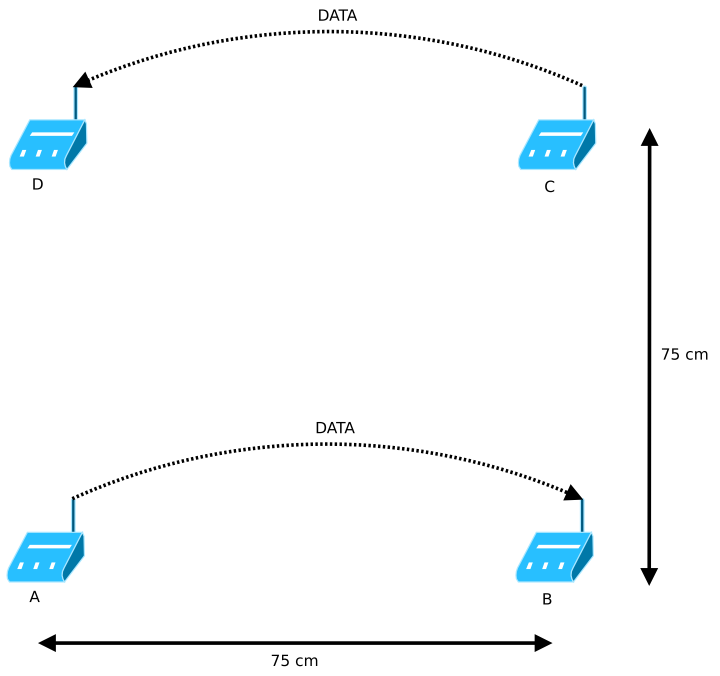


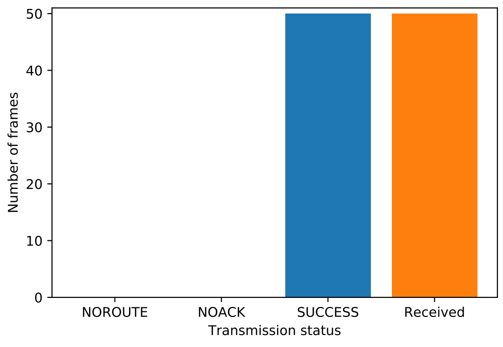
Figure 5.19: Comparison with X-MAC: Test setup (2 flows)

The results of the experiment are presented in Table 5.10 and Figure 5.20.

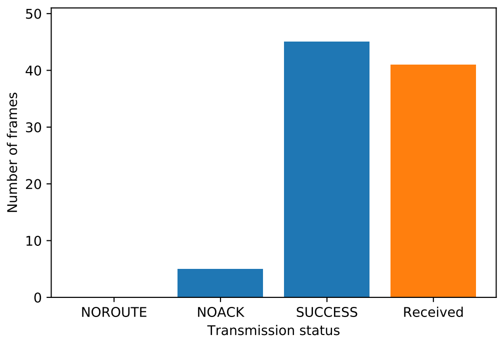
In this scenario, the difference in the role of the acknowledgment between both protocols was even more apparent. CherryRiMAC managed to deliver in total 99 data frames (one base beacon was not received), although it signaled successful transmission only 97 times (two ack beacons were not got). In contrast, the X-MAC senders reported SUCCESS 84 times, but only 75 frames were actually got by the receivers.

MAC protocol	CherryRiMAC	X-MAC
Protocol cycle	1 s	1 s
Test cycle	2 s	2 s
Issued transmissions in each flow	50	50
Issued transmissions per test cycle	1	1
Free buffers for incoming frames	1	1
Flows	A to B, C to D	A to B, C to D
Frames sent successfully (per flow)	50, 47	45, 39
Frames received successfully (per flow)	50, 49	41, 34
Average transmission duration (per flow)	0.59 s, 0.55 s	0.47 s, 0.48 s

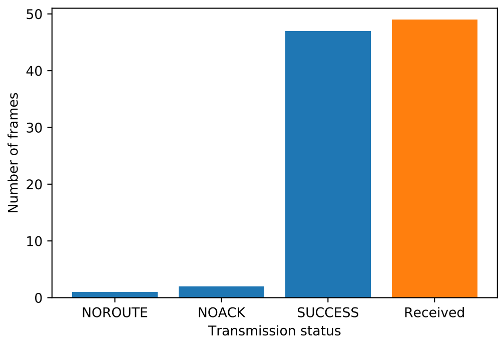
Table 5.10: Comparison with X-MAC: Test summary (2 flows)



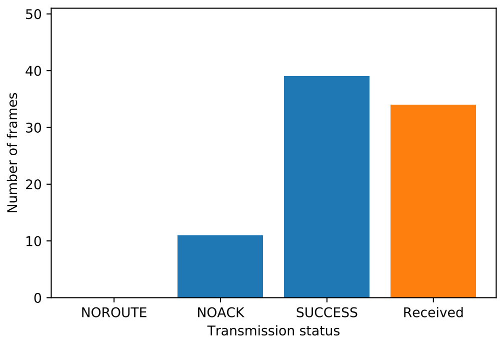
(a) CherryRiMAC: A to B



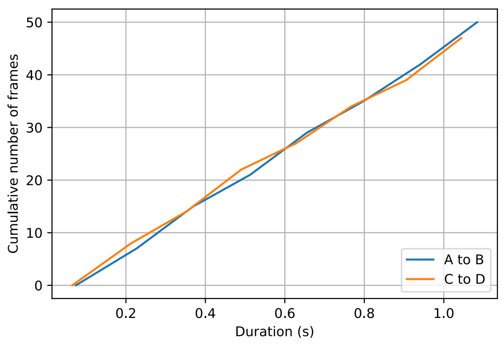
(b) X-MAC: A to B



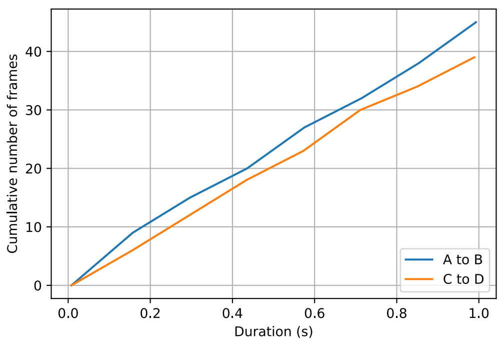
(c) CherryRiMAC: C to D



(d) X-MAC: C to D



(e) CherryRiMAC: Transmission time



(f) X-MAC: Transmission time

Figure 5.20: Comparison with X-MAC: Test results (2 flows)

5.11. Clock drift

As the last issue, let us consider the effect of clock drift in CherryRiMAC.

To make the clock drift evident and measure its actual rate on CherryMote devices, the following experiment was prepared. A node was transmitting data frames to another node with an increasing interval. Therefore, when the sender received a base beacon, it was able to calculate a difference between its timestamp and the expected arrival time based on the previous reception. The knowledge about the clock drift rate can be used to optimize timings of the CherryRiMAC protocol (see Section 4.15), especially the duration of additional listening before and after an expected beacon arrival time.

Figure 5.21 presents differences between actual and expected timestamps of base beacons measured when device *A* was sending data frames to device *B* and to device *D*. Note that devices *A* and *B* are both prototypes v2 whereas device *D* is the fourth CherryMote version.

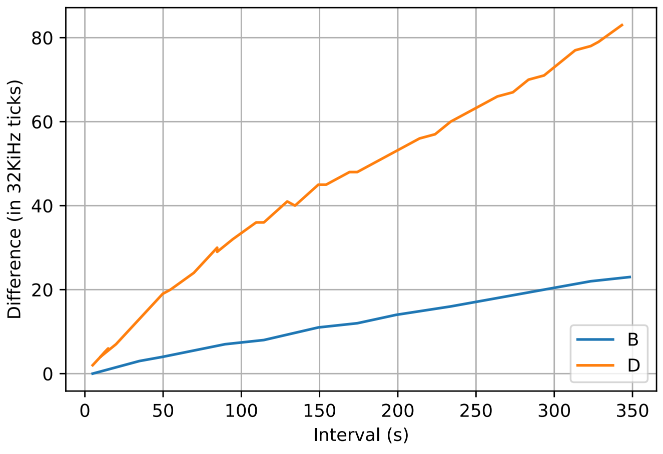


Figure 5.21: Clock drift: Difference between expected and actual base beacon timestamp

The measured values indicate that the additional listening which lasts 79 ticks in the current implementation should allow for a successful communication even when the previous exchange of messages took place 5 minutes earlier.

5.12. Conclusions

In conclusion, the performed preliminary evaluation has shown that CherryRiMAC worked as expected and provided the intended performance, especially when used under light traffic. Incorporating into the design features such as the phase awareness and the Always Listen mode seems to result in greatly reduced power consumption and should allow for successfully handling the targeted throughput level. In contrast, when used in a congested network, the performance was not so high, but still many frames were delivered in a reliable way. In none of these tests, did nodes running the implementation of CherryRiMAC stop to be responsive.

Furthermore, the validity of the guarantee that a sender reports successful transmission only when the data frame is received by the receiver was held in all performed tests. This indicates that the goal to offer reliable transmissions has been met both in the design and the implementation of CherryRiMAC.

As future improvements, the protocol can be enhanced with mechanisms to handle and prevent frames collisions. Incorporating such solutions into the design should improve the effectiveness not only of broadcast transmissions, but also of standard message exchanges in deployments with many contending nodes.

Chapter 6

Conclusions and Future Work

The aim of this thesis was to implement and evaluate a receiver-initiated MAC protocol for dependable low-power wireless networks. An ideal solution should manage communication between nodes in a way that guarantees a high frame delivery rate and minimizes power consumption of both senders and receivers. Transmissions should be performed in a reliable way and with low latencies. Also other functionality offered by the protocol is important: some higher network layers require an ability to broadcast a frame, some deployments feature special purpose gateway devices. Finally, a crucial attribute of a reliable MAC protocol is its robustness and resistance to interference from different surrounding networks.

6.1. Discussion of this thesis' contribution

To achieve the goal, the original RI-MAC protocol was thoroughly analyzed to explore advantages of receiver-initiated communication. However, also some shortcomings like excessive idle listening were identified. Knowing that there were different solutions already implemented in other protocols, which could improve a performance of a receiver-initiated design, it was decided to create a new solution: CherryRiMAC.

The newly designed protocol incorporates the same principle of a receiver-initiated communication as RI-MAC, but additionally enhances it with the phase awareness mechanisms. This approach allowed to greatly reduce the time when a node's radio needs to be powered up, which was later confirmed by a comparison with the X-MAC protocol using actual devices. Moreover, additional features like the best-effort broadcast or the Always Listen mode were introduced to the design to improve usability of the protocol and take advantage of special purpose devices in a network. It turned out that these changes required additional modifications to the original solution.

CherryRiMAC was then actually implemented for physical devices. The whip6 operating system was enhanced with the ability to exchange messages with other nodes using the newly created protocol. The implementation explored also the potential of a modern low-power radio, like the CC2650 chip: some of the required actions can be performed automatically by the hardware. This allowed to simplify the implementation and further minimize power consumption of a device by being able to precisely schedule radio operations. During the process of creating a usable protocol some crucial decisions had to be made: "which frame should be transmitted next?", "which operation should be performed now?". A lot of effort was also required to assure that the solution is free from concurrency problems which could lead to abnormal behaviors of a node.

Finally, CherryRiMAC was preliminarily evaluated in different scenarios set up with CherryMote devices. It was studied how the implementation handles communication during different levels of the medium congestion. Both unicast and broadcast transmissions were tested as well as devices simulated different configuration of senders and receivers. Finally, CherryRiMAC was compared with X-MAC to better access its performance.

The performed evaluation indicates that the newly created protocol is able to handle moderate-rate communication in a reliable way. During an excessive simultaneous traffic the delivery ratio was not so eminent, but the robustness of the design allowed for successfully exchanging messages. The effectiveness of the best-effort broadcast was estimated and the potential of the introduced Always Listen mode was demonstrated. Last but not least, the aimed reliability of both the design and the implementation was confirmed since even during stressed conditions the devices were still responsive and the guarantee that a successful transmission is reported only when the frame is actually received by the recipient was never violated.

6.2. Future work

The preliminary evaluation process, in addition to verifying the properties of the newly created protocol, allowed also for identifying possible enhancements which, when introduced to CherryRiMAC, should result in even better performance under different workloads.

As far as the design is concerned, it should be profitable to incorporate in it some mechanisms to cope better with a congested medium. On the one hand, detecting frame collisions and issuing then appropriate retransmissions should improve delivery rate in networks with contending nodes. On the other hand, some ways to prevent potential collisions should be considered since they can improve a lot the best-effort broadcast and increase reliability of the Always Listen mode.

Future improvements related to the implementation should concentrate on refining the way in which sending and receiving are ended, since precise time measurements of each phase of the transmission indicated that despite the low current radio activity duration, it can be minimized even further. Additionally, it is worth to analyze whether more of the actions executed by the protocol can be reimplemented using functionality provided by the hardware since the already used one proved to be reliable and incorporating it resulted in a cleaner implementation, which is able to follow much more precise timings.

An open issue that should be further studied is how CherryRiMAC handles message exchanges in a network build from many more nodes than those used during the preliminary evaluation process. It is recommended that when the final version of CherryMote devices is available, large-scale test setups should be prepared and experiments, similar to the ones described in this thesis, launched on them.

Nevertheless, I am strongly convinced that the current version of CherryRiMAC can certainly be used by the HENI project and an evaluation process of the routing algorithm developed within the project can reliably depend on the newly created MAC protocol.

Bibliography

[1] Texas Instruments. CC2650. `http://www.ti.com/product/cc2650`. SimpleLink multi-standard 2.4 GHz ultra-low power wireless MCU. 9, 11, 43, 69

[2] IEEE. *IEEE Standard for Low-Rate Wireless Networks*. IEEE 802.15.4-2015. 9, 15, 17, 22, 39, 40

[3] David Gay, Philip Levis, David Culler, and Eric Brewer. *nesC 1.3 Language Reference Manual*, 2009. 10, 43, 54

[4] nesC. `http://nescc.sourceforge.net/`. 10, 43

[5] Yanjun Sun, Omer Gurewitz, and David B. Johnson. RI-MAC: A receiver-initiated asynchronous duty cycle MAC protocol for dynamic traffic loads in wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, 2008. 10, 12, 15, 16

[6] whip6. `https://github.com/InviNets/whip6-pub`. 10, 12, 22, 43

[7] InviNets. Thermomesh. `http://invinets.com/thermomesh.html`. 11

[8] Texas Instruments. *CC2650 SimpleLink^{TM} Multistandard Wireless MCU datasheet (Rev. B)*, SWRS158B edition, July 2016. 11

[9] IEEE. *IEEE Standard for Ethernet*. IEEE 802.3-2015. 11

[10] IEEE. *IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE 802.11-2016. 12

[11] HENI project. `https://www.mimuw.edu.pl/~iwanicki/projects/heni/`. 12

[12] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, 2006. 13, 16, 88

[13] Koen Langendoen. Medium access control in wireless sensor networks. In *Medium Access Control In Wireless Sensor Networks*, volume 2: Practice and Standards. Nova Science Publishers, 2008. 15

[14] Rajesh Yadav, Shirshu Varma, and N. Malaviya. A survey of MAC protocols for wireless sensor networks. *UbiCC Journal*, 4(3), 2009. 15

[15] Anuradha Rai, Suman Deswal, and Parvinder Singh. MAC protocols in wireless sensor network: A survey. *International Journal of New Innovations in Engineering and Technology*, 5(1), 2016. 15

[16] Ilker Demirkol, Cem Ersoy, and Fatih Alagoz. MAC protocols for wireless sensor networks: a survey. *IEEE Communications Magazine*, 44(4), 2006. 15

[17] Wei Ye, John S. Heidemann, and Deborah Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002. 16

[18] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, 2003. 16

[19] Shu Du, Amit Kumar Saha, and David B. Johnson. RMAC: A routing-enhanced duty-cycle MAC protocol for wireless sensor networks. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, 2007. 16

[20] Yanjun Sun, Shu Du, Omer Gurewitz, and David B. Johnson. DW-MAC: A low latency, energy efficient demand-wakeup MAC protocol for wireless sensor networks. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2008. 16

[21] Amre El-Hoiydi. Aloha with preamble sampling for sporadic traffic in ad hoc wireless sensor networks. In *IEEE International Conference on Communications*, volume 5, 2002. 16

[22] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second International Conference On Embedded Networked Sensor Systems*, 2004. 16

[23] Amre El-Hoiydi and Jean-Dominique Decotignie. WiseMAC: An ultra low power MAC protocol for multi-hop wireless sensor networks. In *International symposium on algorithms and experiments for sensor systems, wireless networks and distributed robotics*, 2004. 16

[24] Gertjan P Halkes and KG Langendoen. Crankshaft: An energy-efficient MAC-protocol for dense wireless sensor networks. In *European Conference on Wireless Sensor Networks*, 2007. 16

[25] Philipp Hurni and Torsten Braun. An energy-efficient broadcasting scheme for unsynchronized wireless sensor MAC protocols. In *Seventh International Conference on Wireless On-demand Network Systems and Services*, 2010. 16

[26] Kevin Klues, Gregory Hackmann, Octav Chipara, and Chenyang Lu. A component-based architecture for power-efficient media access control in wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, 2007. 16

[27] TinyOS. `https://github.com/tinyos/tinyos-main`. 16, 22

[28] Jeongkeun Lee, Wonho Kim, Sung-Ju Lee, Daehyung Jo, Jiho Ryu, Taekyoung Kwon, and Yanghee Choi. An experimental study on the capture effect in 802.11a networks. In *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, 2007. 19

[29] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, 2004. 21

[30] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. RPL: IPv6 routing protocol for low-power and lossy networks. RFC 6550, Internet Engineering Task Force (IETF), March 2012. 21

[31] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, 2004. 21

[32] Contiki. `https://github.com/contiki-os/contiki`. 22

[33] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating systems for low-end devices in the internet of things: a survey. *IEEE Internet of Things Journal*, 3(5), October 2016. 22

[34] Adam Dunkels. The ContikiMAC radio duty cycling protocol. `http://dunkels.com/adam/dunkels11contikimac.pdf`. 22

[35] Contiki. Radio duty cycling. `https://github.com/contiki-os/contiki/wiki/Radio-duty-cycling`. 22

[36] Simon Duquennoy, Atis Elsts, Beshr Al Nahas, and George Oikonomou. TSCH and 6TiSCH for Contiki: Challenges, design and evaluation. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems*, 2017. 22

[37] David Moss and Philip Levis. BoX-MACs: Exploiting physical and link layer boundaries in lowpower networking. *Computer Systems Laboratory Stanford University*, 64(66), 2008. 22

[38] Contiki. Pull request: *remove xmac.c and lpp.c*. `https://github.com/contiki-os/contiki/pull/479`. 22

[39] Contiki. Contiki X-MAC. `https://github.com/contiki-os/contiki/blob/master/core/net/mac/cxmac/cxmac.c`. Source: *cxmac.c*. 23

[40] InviNets. GitHub repositories. `https://github.com/InviNets`. 43

[41] InviNets. whip6-nesc. `https://github.com/InviNets/whip6-nesc`. 43, 55

[42] Texas Instruments. *CC13x0, CC26x0 SimpleLink^{TM} Wireless MCU – Technical Reference Manual*, SWCU117H edition, August 2017. 44, 47, 48

[43] Seiko Epson Corporation. *kHz range crystal unit: FC-135R / FC-135*. 66

[44] Olimex. RT5350F-OLinuXino. `https://www.olimex.com/Products/OLinuXino/RT5350F/RT5350F-OLinuXino/`. 69

[45] HENI. Fork of the OpenWrt operating system. https://github.com/heni-project/openwrt. 69

[46] OpenWrt. https://openwrt.org/. 69

Appendix A

CD Contents

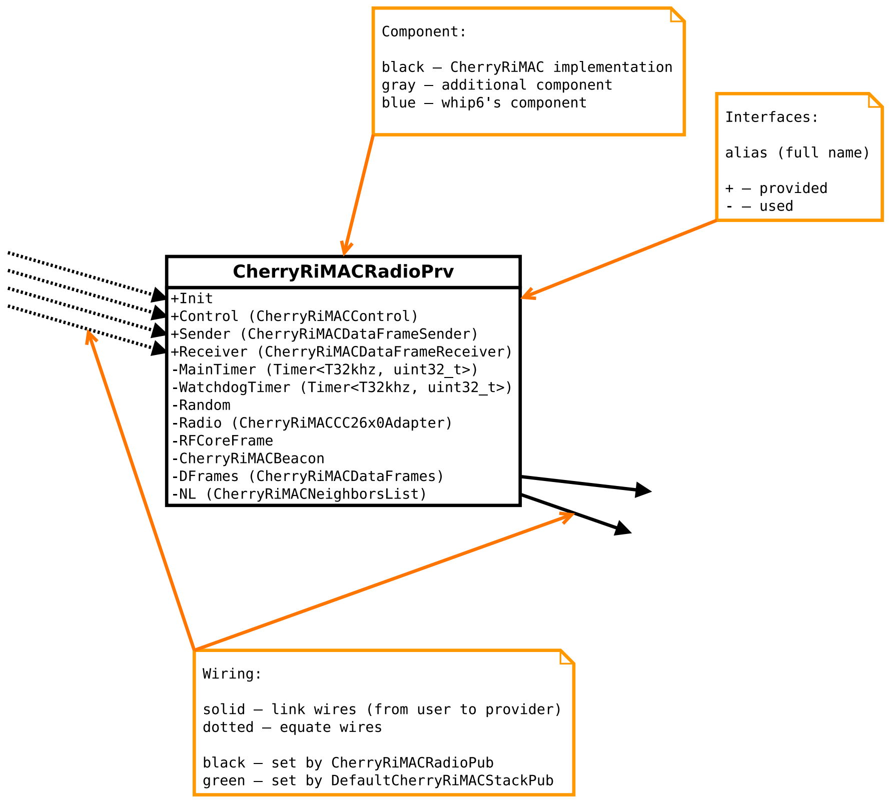
The CD attached to the thesis contains:

- `thesis.pdf`: this thesis in the PDF format,
- `implementation.zip`: a ZIP archive containing source of the whip6 operating system (acquired from `https://github.com/InviNets/whip6-pub` on 17 May 2018) enhanced with the implementation of CherryRiMAC and other works created within this thesis.

Appendix B

CherryRiMAC implementation: Components

A key to the diagram:


Component:
black – CherryRiMAC implementation
gray – additional component
blue – whip6's component
Interfaces:
alias (full name)
+ – provided
- – used
CherryRiMACRadioPrv
+Init
+Control (CherryRiMACControl)
+Sender (CherryRiMACDataFrameSender)
+Receiver (CherryRiMACDataFrameReceiver)
-MainTimer (Timer<T32khz, uint32_t>)
-WatchdogTimer (Timer<T32khz, uint32_t>)
-Random
-Radio (CherryRiMACCC26x0Adapter)
-RFCoreFrame
-CherryRiMACBeacon
-DFrames (CherryRiMACDataFrames)
-NL (CherryRiMACNeighborsList)
Wiring:
solid – link wires (from user to provider)
dotted – equate wires
black – set by CherryRiMACRadioPub
green – set by DefaultCherryRiMACStackPub

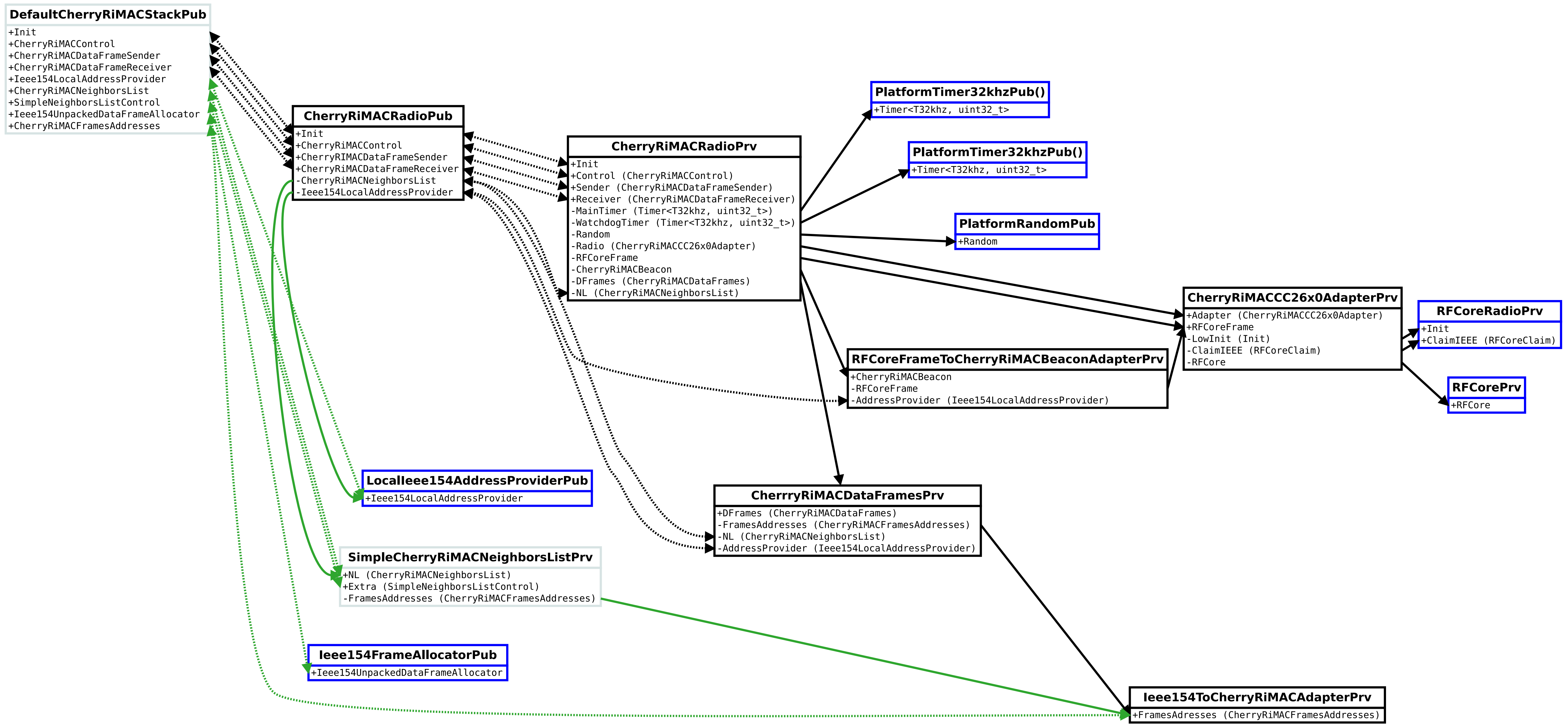
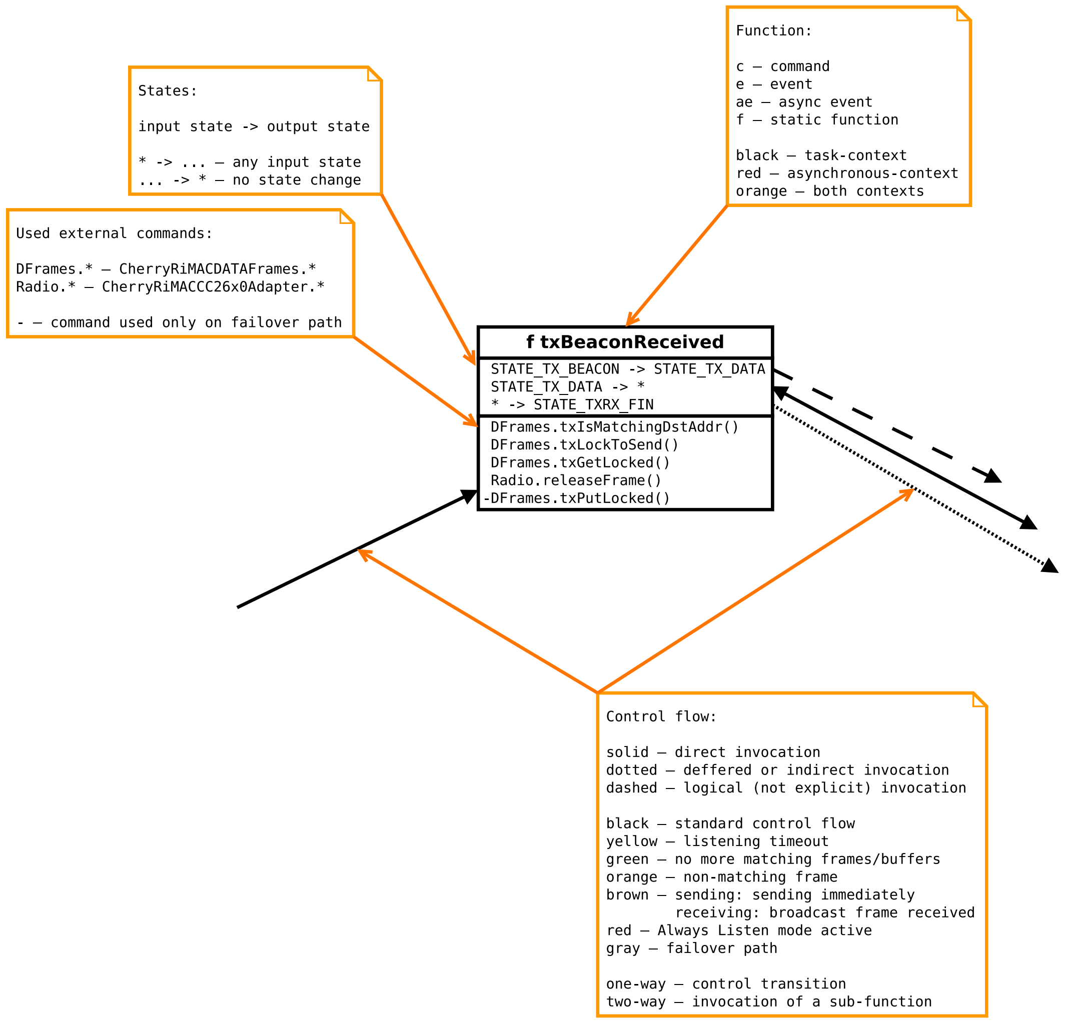

DefaultCherryRiMACStackPub
+Init
+CherryRiMACControl
+CherryRiMACDataFrameSender
+CherryRiMACDataFrameReceiver
+Ieee154LocalAddressProvider
+CherryRiMACNeighborsList
+SimpleNeighborsListControl
+Ieee154UnpackedDataFrameAllocator
+CherryRiMACFramesAddresses
CherryRiMACRadioPub
+Init
+CherryRiMACControl
+CherryRIMACDataFrameSender
+CherryRiMACDataFrameReceiver
-CherryRiMACNeighborsList
-Ieee154LocalAddressProvider
CherryRiMACRadioPrv
+Init
+Control (CherryRiMACControl)
+Sender (CherryRiMACDataFrameSender)
+Receiver (CherryRiMACDataFrameReceiver)
-MainTimer (Timer<T32khz, uint32_t>)
-WatchdogTimer (Timer<T32khz, uint32_t>)
-Random
-Radio (CherryRiMACCC26x0Adapter)
-RFCoreFrame
-CherryRiMACBeacon
-DFrames (CherryRiMACDataFrames)
-NL (CherryRiMACNeighborsList)
PlatformTimer32khzPub()
+Timer<T32khz, uint32_t>
PlatformTimer32khzPub()
+Timer<T32khz, uint32_t>
PlatformRandomPub
+Random
CherryRiMACCC26x0AdapterPrv
+Adapter (CherryRiMACCC26x0Adapter)
+RFCoreFrame
-LowInit (Init)
-ClaimIEEE (RFCoreClaim)
-RFCore
RFCoreRadioPrv
+Init
+ClaimIEEE (RFCoreClaim)
RFCorePrv
+RFCore
RFCoreFrameToCherryRiMACBeaconAdapterPrv
+CherryRiMACBeacon
-RFCoreFrame
-AddressProvider (Ieee154LocalAddressProvider)
LocalIeee154AddressProviderPub
+Ieee154LocalAddressProvider
CherrryRiMACDataFramesPrv
+DFrames (CherryRiMACDataFrames)
-FramesAddresses (CherryRiMACFramesAddresses)
-NL (CherryRiMACNeighborsList)
-AddressProvider (Ieee154LocalAddressProvider)
SimpleCherryRiMACNeighborsListPrv
+NL (CherryRiMACNeighborsList)
+Extra (SimpleNeighborsListControl)
-FramesAddresses (CherryRiMACFramesAddresses)
Ieee154FrameAllocatorPub
+Ieee154UnpackedDataFrameAllocator
Ieee154ToCherryRiMACAdapterPrv
+FramesAdresses (CherryRiMACFramesAddresses)

Figure 1: CherryRiMAC implementation: Wiring of components

Appendix C

CherryRiMAC implementation: Control flows

A key to diagrams:



States:
input state -> output state
* -> ... – any input state
... -> * – no state change
Function:
c – command
e – event
ae – async event
f – static function
black – task-context
red – asynchronous-context
orange – both contexts
Used external commands:
DFrames.* – CherryRiMACDATAFrames.*
Radio.* – CherryRiMACCC26x0Adapter.*
- – command used only on failover path
f txBeaconReceived
STATE_TX_BEACON -> STATE_TX_DATA
STATE_TX_DATA -> *
* -> STATE_TXRX_FIN
DFrames.txIsMatchingDstAddr()
DFrames.txLockToSend()
DFrames.txGetLocked()
Radio.releaseFrame()
-DFrames.txPutLocked()
Control flow:
solid – direct invocation
dotted – deffered or indirect invocation
dashed – logical (not explicit) invocation
black – standard control flow
yellow – listening timeout
green – no more matching frames/buffers
orange – non-matching frame
brown – sending: sending immediately
receiving: broadcast frame received
red – Always Listen mode active
gray – failover path
one-way – control transition
two-way – invocation of a sub-function

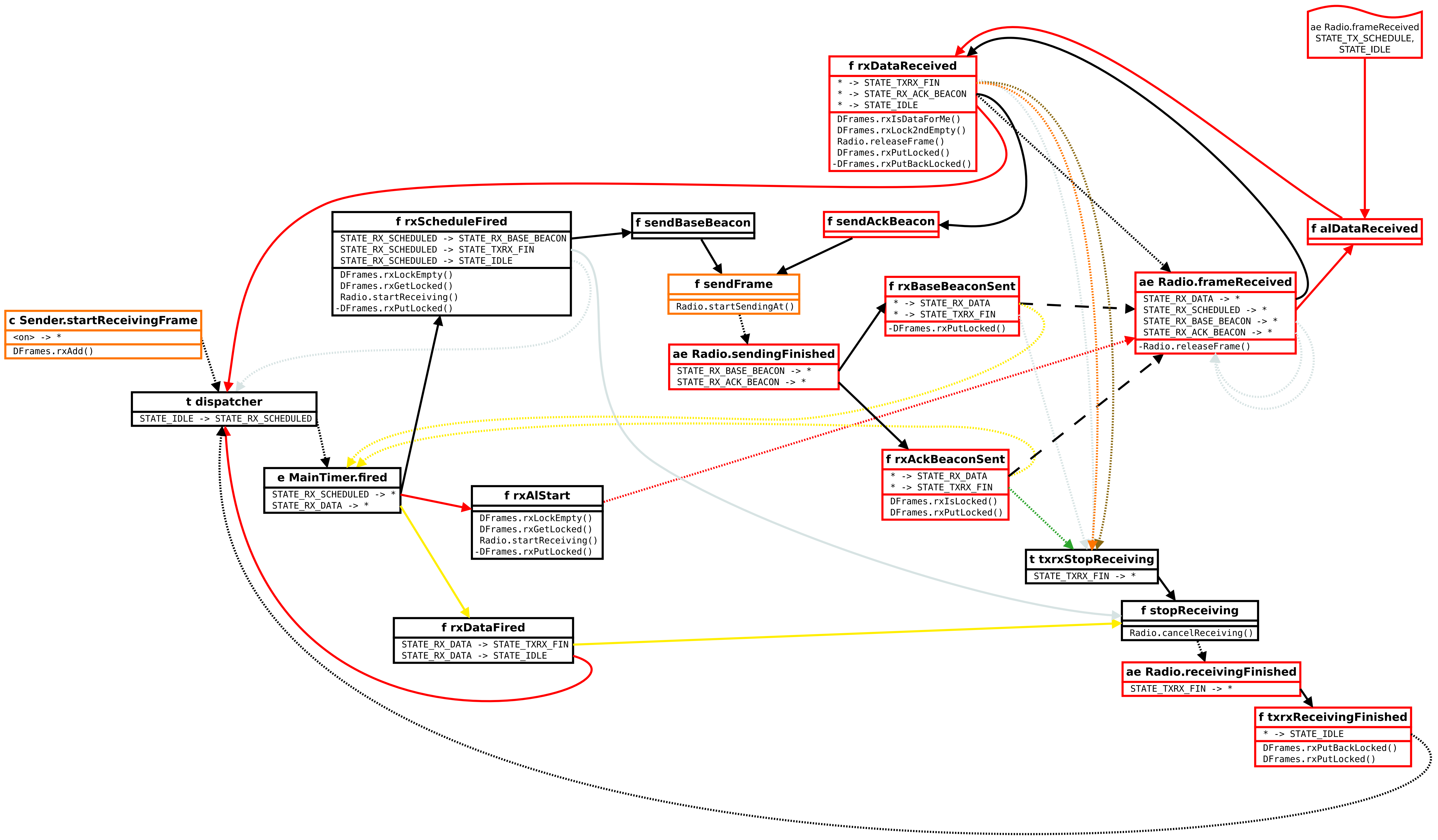


Figure 1: CherryRiMAC implementation: Control flow during the receiving operation

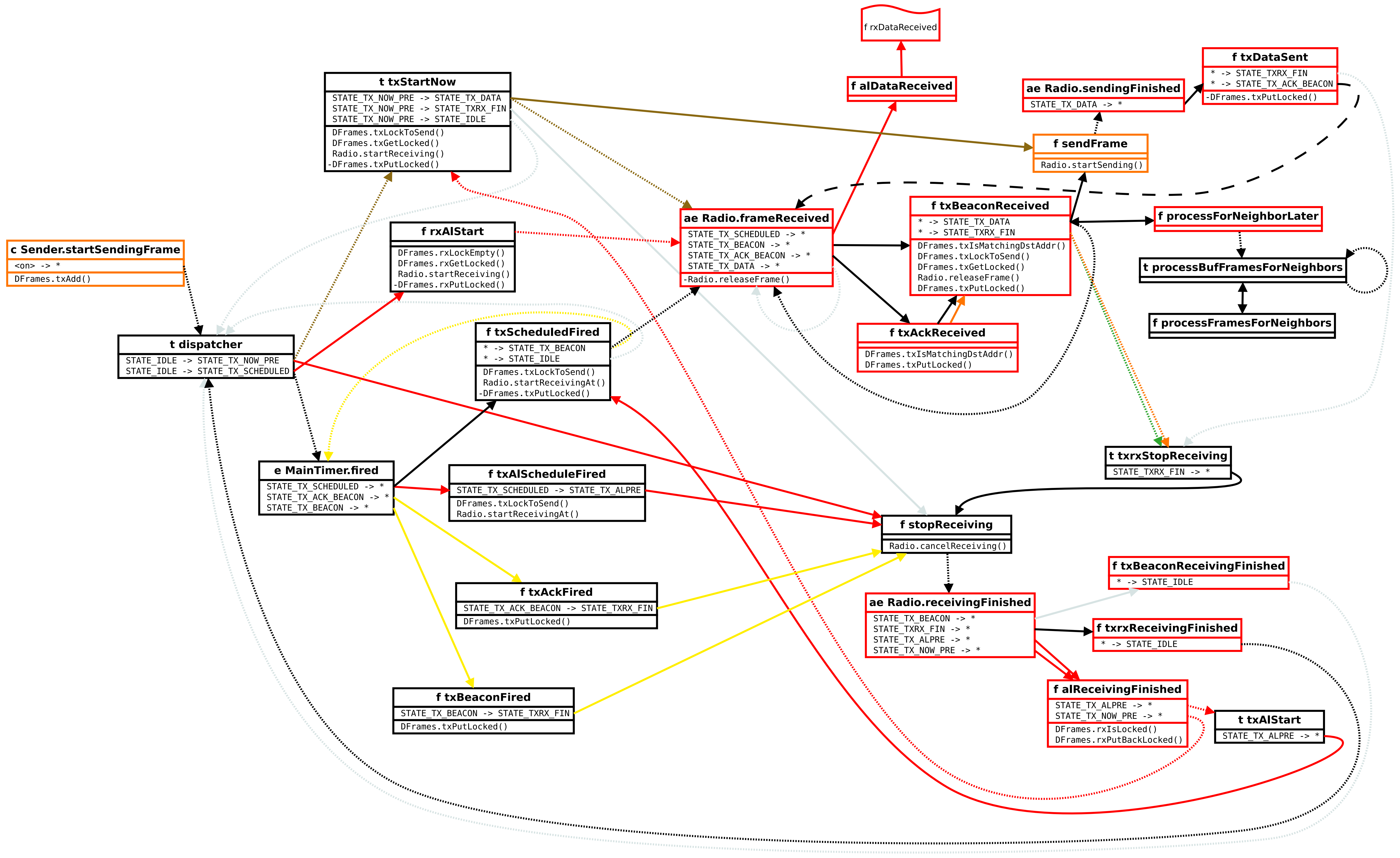


Figure 2: CherryRiMAC implementation: Control flow during the sending operation

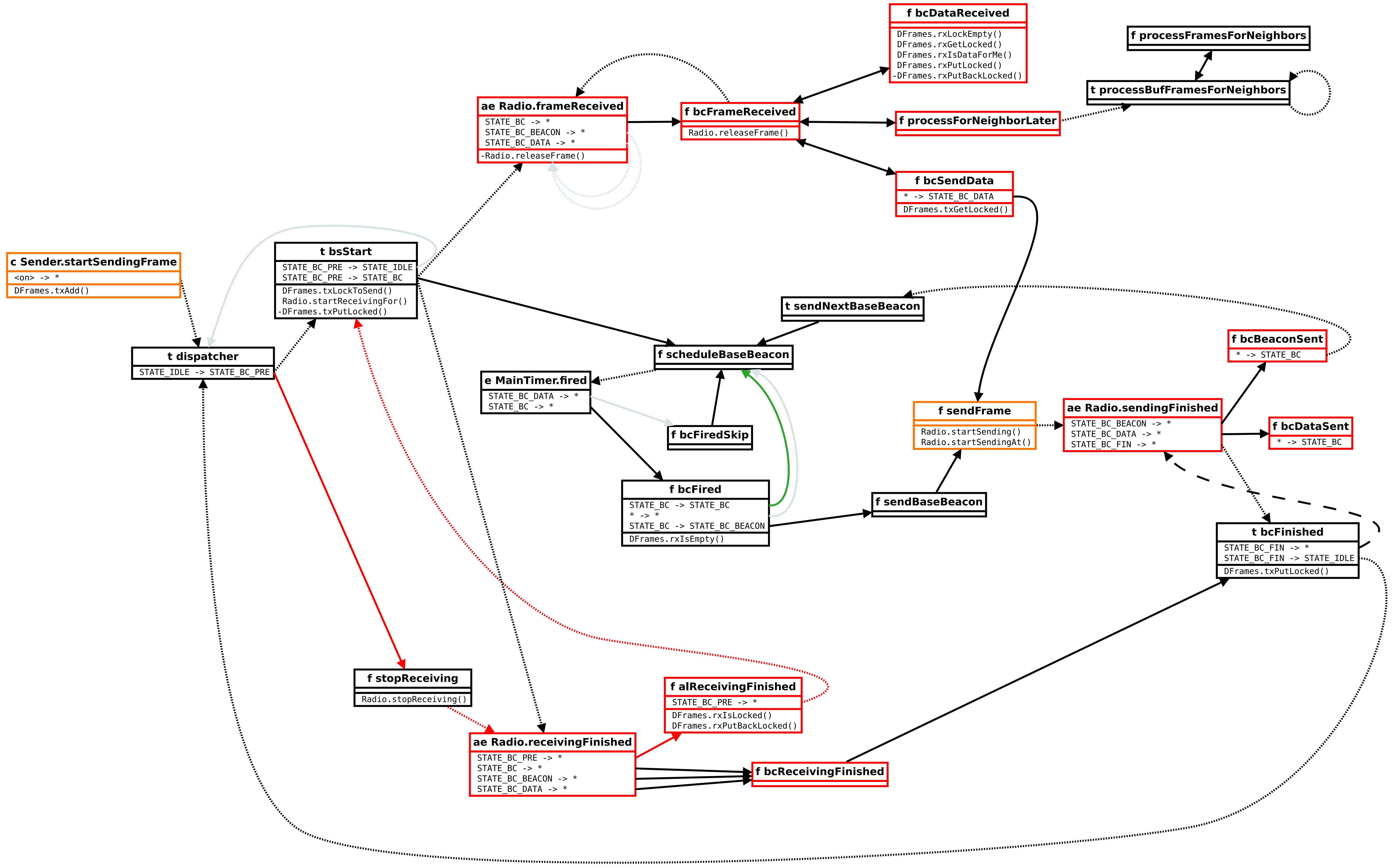


Figure 3: CherryRiMAC implementation: Control flow during the Broadcast operation

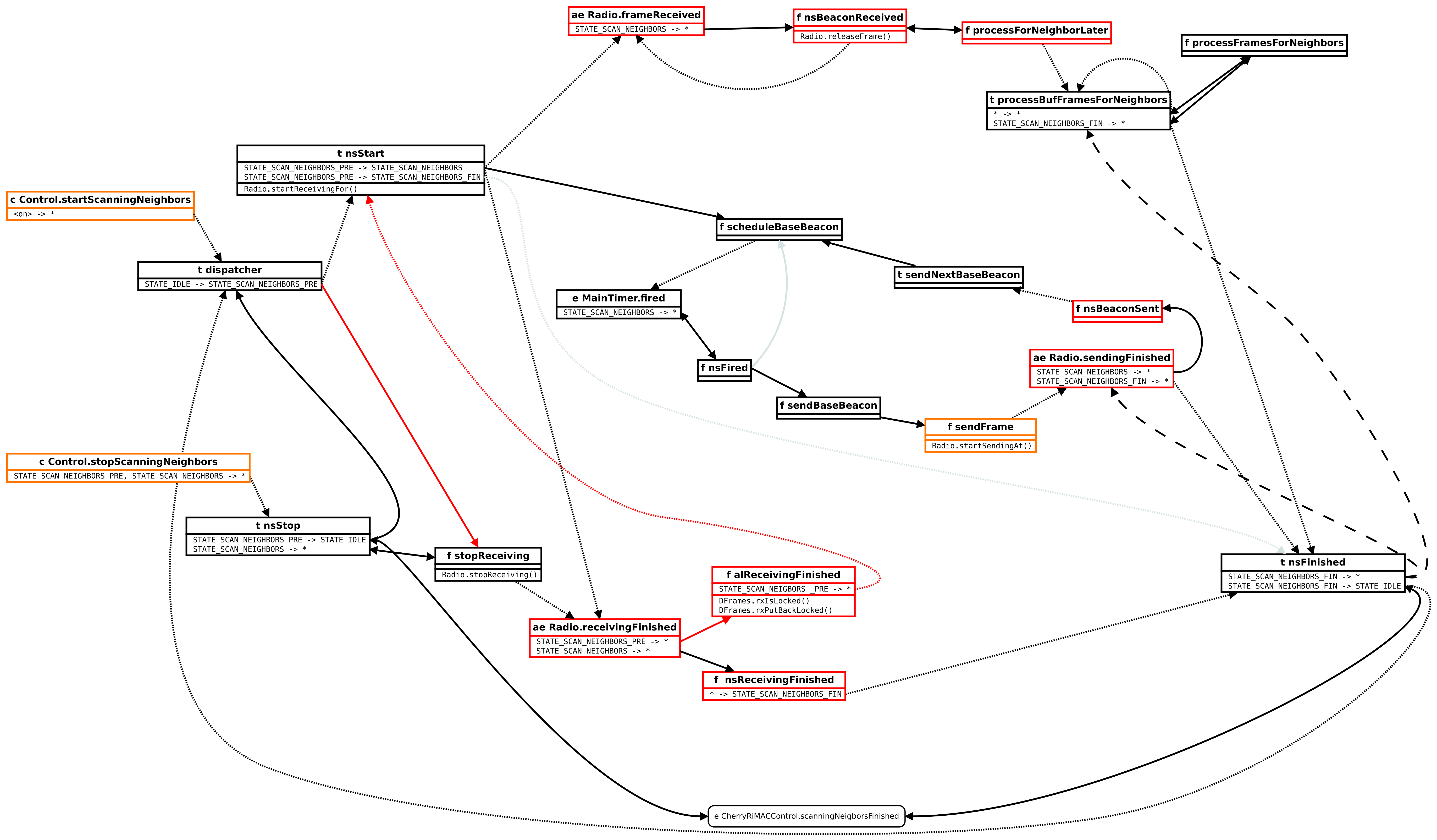

ae Radio.frameReceived
STATE_SCAN_NEIGHBORS -> *
f nsBeaconReceived
Radio.releaseFrame()
f processForNeighborLater
f processFramesForNeighbors
t processBufFramesForNeighbors
* -> *
STATE_SCAN_NEIGHBORS_FIN -> *
t nsStart
STATE_SCAN_NEIGHBORS_PRE -> STATE_SCAN_NEIGHBORS
STATE_SCAN_NEIGHBORS_PRE -> STATE_SCAN_NEIGHBORS_FIN
Radio.startReceivingFor()
c Control.startScanningNeighbors
<on> -> *
f scheduleBaseBeacon
t dispatcher
STATE_IDLE -> STATE_SCAN_NEIGHBORS_PRE
t sendNextBaseBeacon
e MainTimer.fired
STATE_SCAN_NEIGHBORS -> *
f nsBeaconSent
f nsFired
ae Radio.sendingFinished
STATE_SCAN_NEIGHBORS -> *
STATE_SCAN_NEIGHBORS_FIN -> *
f sendBaseBeacon
f sendFrame
Radio.startSendingAt()
c Control.stopScanningNeighbors
STATE_SCAN_NEIGHBORS_PRE, STATE_SCAN_NEIGHBORS -> *
t nsStop
STATE_SCAN_NEIGHBORS_PRE -> STATE_IDLE
STATE_SCAN_NEIGHBORS -> *
f stopReceiving
Radio.stopReceiving()
t nsFinished
STATE_SCAN_NEIGHBORS_FIN -> *
STATE_SCAN_NEIGHBORS_FIN -> STATE_IDLE
f alReceivingFinished
STATE_SCAN_NEIGBORS _PRE -> *
DFrames.rxIsLocked()
DFrames.rxPutBackLocked()
ae Radio.receivingFinished
STATE_SCAN_NEIGHBORS_PRE -> *
STATE_SCAN_NEIGHBORS -> *
f nsReceivingFinished
* -> STATE_SCAN_NEIGHBORS_FIN
e CherryRiMACControl.scanningNeigborsFinished

Figure 4: CherryRiMAC implementation: Control flow during the Neighbors Scan operation