Implementing and evaluating dual-radios with TSCH MAC for Industrial Wireless Sensor Networks.

Vegar Krogsethagen

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

August 2018

Implementing and evaluating dual-radios with TSCH MAC for Industrial Wireless Sensor Networks.

Vegar Krogsethagen

2018

Implementing and evaluating dual-radios with TSCH MAC for Industrial Wireless Sensor Networks

Vegar Krogsethagen

http://www.duo.uio.no

Printed: Reprosentralen, Universitetet i Oslo

Summary

Now that Internet of Things (IoT) is emerging, we see new industrial applications emerge in the industrial paradigm. Terms like Industrial Internet of Things (IIoT), Internet of Everything (Cisco), appear everywhere. An integral part in industrial networks is (Operational Technology) along with IP based networks (informational technology).
As WSN has been important in the development of IoT, IWSN will be an important part of the development of Industrial Internet of Things (IIoT). WSN technologies have shown great potential for industrial use. Particularly in process monitoring and control where processes such as pressure, temperature, flow, vibration can be measured and transferred wirelessly to a control system. Wireless communication has several advantages like: - Reduction in Cost, Flexibility and Performance. Traditional cable Industrial Automation and Control Systems (IACS) can come up to several thousand dollars in both cables, installation and maintenance. Moving machinery also poses major challenges, this is solved with different techniques like sliding contacts and slip rings, but this is also prone to wear.
We also see that by reducing costs we can increase the number of actuators and sensors which in turn gives an increased performance.
A promising MAC solution is to combine time-division multiple access (TDMA) and frequency hopping to meet the strict requirements of reliability and real-time characteristics, *called Time Slotted Channel Hopping (TSCH). It uses time synchronization and channel hopping to mitigate effects of external interference and multi-path fading.*
This is used in well-known standards such as WirelessHART and ISA100.11.a.
TSCH was standardized by IEEE 802.15.4-2015 and selected by IETF for standardization in "IPv6 over the TSCH mode of IEEE 802.15.4.e" (6TiSCH).
By using two homogenous radios on each sensor or actuator each link has different channels. This utilize the same effect that makes channel hopping so efficiently (frequency-dependent fading effects) and could make it possible to achieve extremely high reliability, and with extremely low jitter and delay.

Preface

I would like to thank Andreas Urke and my supervisor Professor Knut Øvsthus at Western Norway University of Applied Sciences (HVL) for excellent counseling. I would also like to thank the Western Norway University of Applied Sciences (HVL) for hospitality with my own laboratory space, as well as equipment.

Contents

List of Figures

List of Tables

1 Introduction

Industrial networks typically require high reliability and real-time characteristics (low latency and as little variation in latency as possible), traditionally solved with wired solutions. However, the benefits of wireless solutions have meant that more researchers have considered it for more demanding industrial applications [1]. There are two key phenomena that affect packet loss and delay, this is *external interference* where other wireless technologies operate on the same frequency band and can create collisions and packet loss, and *multi-path fading* where radio waves can propagate through different paths to the destination (hence receive multiple signals) and create a destructive effect that can result in packet loss [2]. These phenomena can be reduced using channel hopping, where subsequent packets are sent at different frequencies. Channel hopping has been shown to reduce ETX (metric indicating how many transmissions it takes for a packet to be successfully received) by 63% [3]. Channel hopping is used in several major standards such as ISA100.11.a, WirelessHART and IEEE 802.15.4.e Time Slotted Channel Hopping (TSCH) [4].

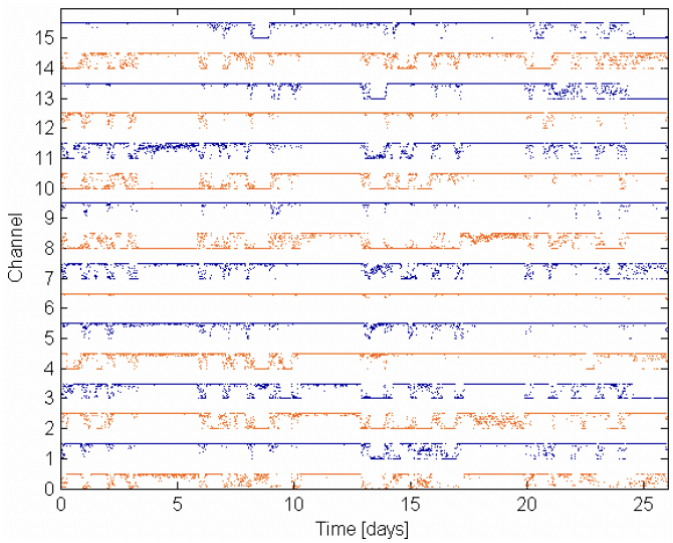

Figure 1: Stability of link between two nodes for different channels in factory deployment [5]

In [5] they placed 44 nodes in a 2.4GHz wireless sensor network in a factory. The nodes sent data packets every 28 seconds and retrieved information about the number of transmissions, received acknowledgements, average RSSI, and other metrics for a single link using one channel for 15 minutes at a time. They did this for 26 days. They concluded that *"(..) Based on this plot, no channel appears significantly better or worse than any other when averaged*

over time and paths, and the network appears to function well on all frequencies". The result can also be seen in Figure 1 (successful transmissions and receipt of acknowledgments).

However, it is not possible to predict which channel is interfered, so creating a hopping sequence that avoids channels exposed to interreference is difficult, and frame loss is inevitable. In the industrial network the use of retransmissions increases the reliability, but retransmissions also increase latency and jitter which has a negative impact on the requirements for real-time characteristics (low delay, low jitter).

2 Background

2.1 Internet of Things (IoT)

IoT is a network of physical devices such as industrial sensors, vehicles, home equipment, smartphones, which shares and exchanges data [6]. Devices sense and collect data from things around us and may use this information locally or even share this data across the world to be used for different purposes. A typical IoT devices (in industrial environment) consist of a processor, a sensor that collect data, and transmits or receives using a radio (shown in Figure 2). Sensors can collect different data such as temperature fluctuations, vibrations in industrial motors to see unnatural behaviors, or even gas leaks. IoT devices can be placed where access is challenging, and traditionally wired system is difficult to install.

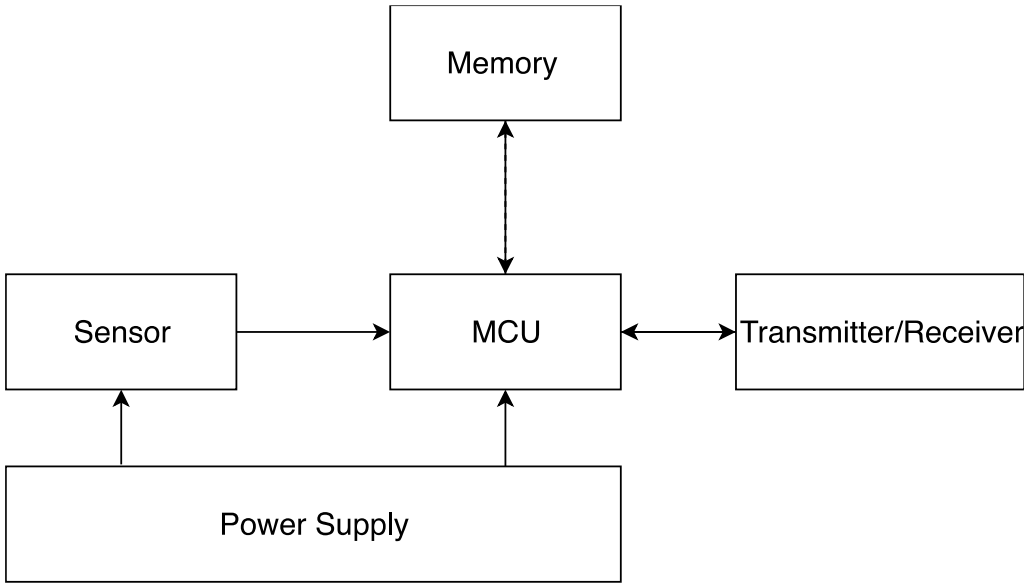

Figure 2: Typical IoT device

IoT is used in many different applications such as [6]:

- Smart buildings
    - Identify location of people as well as the state of the building. Control heating/air condition, and lighting system to reduce the power bill. Sensors can also monitor the building structure health.
- Smart cities
    - Sensors monitor humans or vehicle and can collect data from smart buildings. Use information boards to inform others about car incidents in the car line.
- Industrial systems
    - Use sensor to monitor industrial processes. Collect information about industrial motors health and use this to avoid motor failures. Sensors could monitor gas leaks, fire, unnatural variation, temperature etc.
      Data could be sent directly to actuators that can make changes dynamically [7], [28].
- Medical system
    - Patient mentoring sensor located at home or doctor office.
- Vehicles
    - Could be used to monitor the state of the vehicle. It can reduce fuel consumption, and even monitor air pressure in tires and inform about holes.

Internet of Things does not necessarily mean that wireless communication takes place, sensors could also be wired. But, due to the development of wireless network reliability that manages to establish end-to-end reliability to 99.999% [5], and years or even decade of battery lifetime [8], new opportunities arise through the use of wireless communications in industrial networks.

2.2 Wireless sensor network (WSN)

A WSN typically consists of several cheap, power-friendly, more functional sensors nodes located in an area of interest [9]. They are small but are equipped with radio, sensors and microprocessors, and therefore not only have the ability to sense but also process and send data [9]. Communication takes place over wireless medium and together they can monitor e.g. industrial area, war zone or environmental area [9], [10].

Compared to traditional networks, they have some unique characteristics and requirements that can be summed up this way [9]:

- Battery powered
    - Wireless sensor could use battery as power source and is expected to last several years or even decades before charging/replacing battery [8].
- Environment
    - Wireless nodes can be placed in hard or hostile areas where access is limited or even impossible [9].
- Limited power recourses, processing capability and storage capabilities
    - The nodes have limited power availability, processing capability and storage capabilities [9]. Storage and processing capability can be as low as 10kB RAM, 8Mhz CPU, and 48 kB flash memory [11]. Due to the limited processing capability large demanding protocols should be avoided [4].
- Self-Configurable
    - Because nodes can autonomously discover each other to build a topology, they could be placed in the area of interest and monitor without special planning [9]. They could also make dynamically changes to the topology on node failure e.g. out of battery hence also self-healing.
- Application Specific
    - The sensor often has a specific task, or build to a specific application [9]
- Unreliable sensor nodes
    - Due to the hostile / harsh environment the nodes are placed in, they are often exposed to damage or malfunction [9].

- Deferred to topology changes
    - Node can fail, get corrupted, run out of power or be exposed to interference [9]. Nodes can also be mobile which make them vulnerable to topology changes
- Multiple-to-one traffic pattern
    - Sensor data sent from a specific node can take more paths to destination [9].
- Redundant
    - Due to the fact that nodes are often sealed in an area of interest, as well as multiple-to-one traffic pattern, nodes may use other paths to the destination [9]. The fact that they are self-configurable allows them to have more redundant paths to their destinations.
- Tightly distributed
    - Node is placed close in an area of interest.

Wireless sensor nodes have great advantages over traditional wired networks, which reduce the cost and delay, and the fact that they can be placed in hostile/harsh environment where wired solutions are impossible [9]. The development in miniaturization will make it possible to get sensors down in a few cubic millimeters in the future [12] that will make it even easier to place those in demanding environments. Because of the size and wireless communication, sensor network can be installed in environments where wired solutions are difficult, this may be because the machines rotate or are mobile, or that the environment they are placing in is dangerous (e.g. war zones).

2.2.1 WSN Topology

There are three different topologies in WSN, Star, Star mesh and mesh [4], [24].
In a Star network (shown in Figure 3), all nodes have a specific path to the border router. This means that if there is an error between two nodes, the information will be lost.
Mesh (shown in Figure 4) has several redundant routes to its destination because nodes are also routing nodes, so the data can take other paths to the destination.

Star network

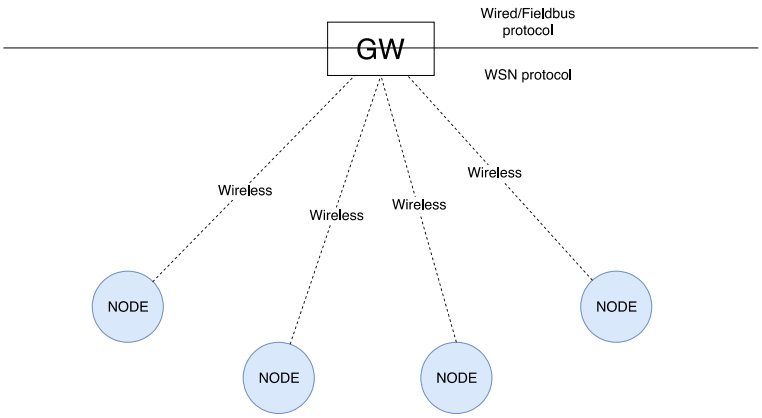

Figure 3: Star topology

- One-hop for any node to gateway
    - Low latency (one-hop)
    - Limitation of possible range
    - Or increase of transmit power

Mesh network

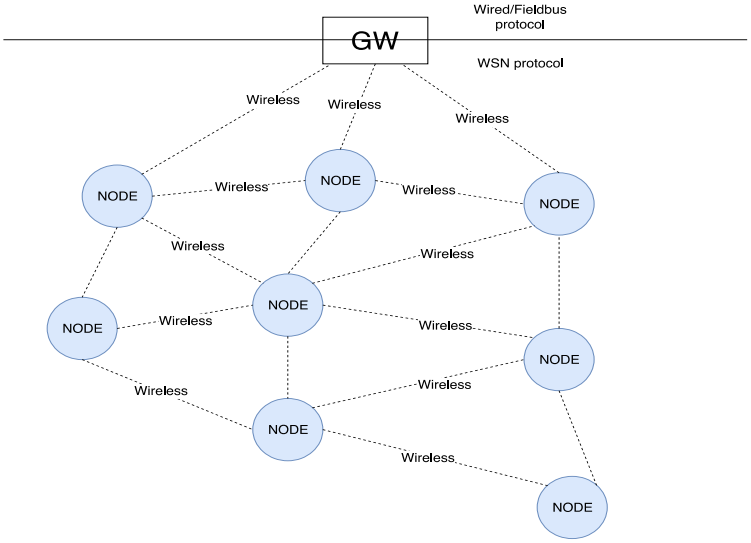

Figure 4: Mesh topology

- Many-hop to gateway
    - Latency can be high (many hops)
    - Routing to improve range
    - Transmission range can be kept low
    - Spending energy for transmitting data for other nodes.

A growing area of WSN is the Industrial Wireless Sensor Network (IWSN), which uses wireless systems to control and monitor different industrial tasks [4].

2.3 Industrial Wireless sensor network (IWSN)

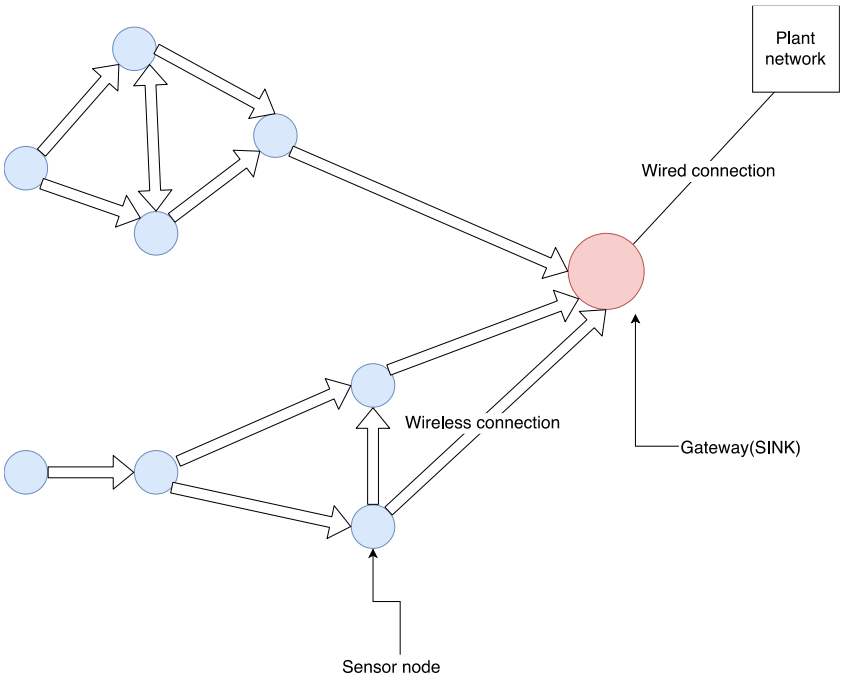

Figure 5: IWSN

WSN has been important in the development of IoT, and IWSN will be an important part of the development of Industrial Internet of Things (IIoT). WSN technologies have shown great potential for industrial use. Particularly in process monitoring and control where processes such as pressure, temperature, flow, vibration can be measured, assembled and transferred wirelessly to a control system (show in Figure 5) [10]. Although wireless systems have been used in control applications for several years, such as Supervisory Control and Data Acquisition (SCADA), WSN for process monitoring and control has not been widely available until recently due to the strict/specific requirements [10], [4].

The advantages of industrial wireless sensor networks versus wired industrial networks can be summed up this way:

- Flexibility
    - A wired solution can often be difficult, e.g. there can be dangerous chemicals/environment, or the cable distances can be very large. Machines can also be mobile or rotating and that makes it even more challenging.
- Cost
    - Traditionally wired Industrial Automation and Control systems (IACS) depend on the wired fieldbus system, and installation costs are often up to thousands of dollars [1], [12], which is up to 80% of the cost of the entire system [13]. Or as [7] states, "*(..) estimates that WSNs enable cost savings of up to 90% compared to the deployment cost of wired field devices in the industrial automation domain*".

- Performance
    - When reducing the cost with wireless communication, it becomes economically feasible to increase the number of sensors, actuators and connection points [8]. They also have a higher data communication speed, such as WirelessHART with data rate up to 250Kbps (IEEE 802.15.4 standard) [4].

      Where the wired sensor network shares the same fieldbus, each sensor in a wireless network can communicate simultaneous, hence has a higher network capacity. Also, with the use of self-healing wireless communication technology and cheap redundant sensor nodes, we can also increase availability [4].

What separates IWSN from other networks is the requirements. Requirements in Industrial Wireless sensor networks (IWSN) are different than other wireless networks, where IWSN is a stricter limit when it comes to packet loss, delay, jitter, reliability and security [4]. Or as [4] describe it "(..) *In comparison with other wireless networks, IWSNs require high reliability and real-time performance, which is challenging due to noisy surroundings*".

2.3.1 Requirements

The requirements in IWSN can be summarized in this way:

- Real-time characteristics
    - Deterministic behavior
        - Industrial communication requires very low latency, and as low (less then 20μs [27]) variation in latency as possible (hence, jitter) as possible. When industrial network is a part of application control loop, latency and jitter can result in degrading of the control system performance, and result in economic loss, or even human safety (e.g. gas pipe explosion).
    - Short latency
        - Industrial application requires short latency, this is especially true when used in control loops when requirements for latency can be as low as microseconds (less then 10μs [27]) [8]. Considering sensor data that may typically only be valid for a short period (i.e. real-time characteristics), latency can cause data to not be accurate anymore or of limited use. This defends the application's prioritization of new data instead of guaranteeing receipt of all packets [8].
- Reliability
    - Reliability is often defined as the network's ability to transfer data between two devices with as low delay as possible and is often measured in metrics such as packet loss or packet delivery ratio (PDR), or as [4] describes it "*Reliability is concerned with how much data is received successfully at the receiving end with minimum delay(..)"*.
      Latency and jitter can affect control loops performance, therefore is guaranteed delivery, or as [1] defines it "*ultra-high reliability"* said to be one of the most important requirements in IWSN. Given that IWSN can be used in security systems, critical alarm data must be received at the other end and with as little delay as possible [14], [28]. From a MAC layer point of view, retransmissions are often used to increase reliability, but this is increases latency and overall throughput because bandwidth is used to retransmit packages.

- Robustness
    - Nodes in IWSN are often exposed to physical harsh industrial environment where dust, dirt, liquid and vibration is inevitable [4]. Industrial wireless environment unlike traditional IT environment contains metal surfaces and machinery (AC convert e.g.). Therefor are redundancy mechanisms important when nodes failure occurs, or a gateway failure [13], [28]. On the MAC layer, robustness typically refers to the network's ability to maintain high reliability when exposed to difficult conditions, such as external interference and multi-path fading [29], [30]. Using unlicensed bands like 2.4GHz, different technologies will create interference, such as Wi-Fi, microwave, power converter, etc.
- Availability
    - Refers to users ability to access services provided by the network and is often measured in “downtime” [22]. Consequence of bad “Availability” is similar as reliability, and in case of a malfunction, process usually has to stop in controlled manners and it can take hours before its fully functional again [8].
- Scalability and adaptability
    - Considering that a sensor network can consist of 100 or even more than 1,000 nodes, protocols must also be scalable to the different topology sizes.
      Nodes in an IWSN are also subject to physical harsh industrial environments where dust, dirt, liquid is inevitable [4]. This along with the fact that nodes could be mobile, should IWSN protocols be adaptive to topology changes [9].
- Resource utilization
    - Energy
        - Wireless sensor usually use battery as power source and is expected to last several years or even decades before charging/replacing battery [8]. Network size, traffic load, number of retransmissions, mobility, all have an impact on power consumption. Due to limitation in node size, nodes should not deal with big demanding protocols for communication [4]. Because of the limited energy consumption, WSN should use an energy-efficient MAC layer that have a low duty cycle and CPU usage. However, IWSN often use a trade-off in battery

consumption to meet the more critical requirements in an industrial environment, such as real-time characteristics and reliability [23].

    - Bandwidth
        - As mentioned in Robustness, when using unlicensed band such as 2,4GHz, different co-exciting technology (Wi-Fi, microwave etc.) all "fight" for available bandwidth. Therefore, all the IWSN layers must be as bandwidth effective as possible for other requirements to be within available bandwidth [10].

- Topology organization
    - To meet the demands of reliability, robustness and power efficiency, topology management needs to be as flexible and efficient as possible. Nodes in the IWSN can be located in hazardous areas, or areas where access is limited, it is therefore a requirement that they must be self-organized and self-configurable [28]. Because of limited access and the harsh environments they are placed in, they must therefore also be self-healing [8].
- Security
    - Industrial network burglary can have a much higher consequence than in traditional networks. There are several known attack forms in the IWSN, ranging from man-in-middle attack, black-hole, selective forwarding and sink-hole. All these forms of attack can have fatal consequences and, at worst, can lead to loss of human life. Techniques for securing data against these attacks include everything from authentication and encryption, link-layer hop-by-hop encryption, and unique ID authentication for each device (challenge to realize sensor nodes is limited), but some types of attacks are difficult to protect against such as selective forwarding and sink-hole.
- Interoperability
    - Today's industrial systems use a combination of cable and wireless sensors to measure data and perform actions. Therefore, wireless systems must also support the old systems. There may also be other wireless systems that the new system must support [13], [28].

2.3.2 Standards

In today's IWSN there are two standards that dominate the industrial wireless networks and it is WirelessHART and ISA 100.11.a [4]. Both are based on IEE 802.15.4 (see appendix).

WirelessHART was the first industrial standard and was introduced in 2007 for control and measurements. It was self-organizing, self-healing mesh network (figure 4) and uses the IEEE 802.15.4 on channels 11-25 at 2.4GHz frequency spectrum [4], [24]. It uses *Time-division multiple access* (*TDMA*) that allows nodes to sleep when the timeslot is neither transmit nor received to reduce energy consumption. It uses synchronized timeslots, with a fixed length of 10ms for real-time communication. It uses Direct-Sequence Spread Spectrum (DSSS) for spreading the message signal by modulating following a bit sequence (radio pulse) and Frequency Hopping Spread Spectrum (FHSS) to spreading code modulation by hopping on a series of frequencies. For routing protocol, it uses graph routing, which contains several redundant routes to nodes [4]. Data transfers over a bad link can be avoided by using blacklisting mechanisms [4].

In September 2009, the International Society of Automation came up with a proposal called *ISA100.11a* for monitoring and controlling applications in the industrial environment [4]. Unlike WirelessHART they use variable timeslots and have features in frequency, spatial diversity and time. They use a channel-hopping schedule, where nodes communicate on different channels each timeslot in a slotframe [4].

Industrial applications can be divided into three parts: Security, Control and Monitoring - and everyone has different requirements. They are again divided into six classes from zero to six, where zero is the most critical and six is the least strict [26], [33] (shown in Table 1).

Category	Class	Type	Application	Examples
Safety	0	Always a critical function	Emergency Action	Safety interlock Emergency shutdown Automatic fire control
Control	1	Often a critical function	Closed-loop regulatory control	Control of primary actuators High frequency cascades
	2	Usually a non-critical function	Closed-loop supervisory control	Low frequency cascade loops Multivariable controls Optimizers
	3	Open-loop control	Open-loop control	Manual flare Remote opening of security gate Manual pump/valve adjustment
Monitoring	4	Short-term operational effect	Alerting	Event-based maintenance
	5	No immediate operational consequence	Logging and downloading / uploading	History collection, sequence-of-events, preventive maintenance

Table 1: Classification industrial application obtained from article [26] and RFC [33]

A commonly centered IWSN network usually consists of node, sink/network manager, management console and a process manager (shown in Figure 6) [28]. The nodes collect data, send them to Sink/Network manager, which in turn communicates with the controller [28]. The use of WSN in control systems has given us new opportunities. Due to the cost reduction wireless sensor network provides, Wireless Network Control System (WNCS) has become an important infrastructure technology for critical control systems (class 1-3) (Table 1) in industrial systems. [7]. In WNCS, sensor nodes attached to the physical plant sample and transmit data over a wireless channel to the controller [7]. The data collected may range from heat expansion, gas leaks or vibration changes. E.g. by looking at the profile of the vibration pattern on different machines you can discover un-naturalities before it breaks down.
The data collected by nodes is then forwarded to actuators that can make changes dynamically [7], [28]. When we introduce wireless communication, we also introduce no-zero delay and message loss. As we have discussed earlier, message loss in industrial systems

can have major consequences (financial loss, downtime or worst-case human safety). Because of the requirements of ultra-low delay (seconds, or even milliseconds depending on the control loops [13], [39]) and high reliability, wireless systems need mechanisms to handle the strict requirements [13].

Critical in such networks is the Media Access Control (MAC), which has a strong impact on both latency, packet delivery ratio (PDR) and power consumption. A promising MAC solution is to combine time-division multiple access (TDMA) and frequency hopping, called Time Slotted Channel Hopping (TSCH). This is used in well-known standards such as WirelessHART and ISA100.11.a. TSCH was standardized by IEEE 802.15.4-2015 [45], and selected by IETF for standardization in "IPv6 over the TSCH mode of IEEE 802.15.4.e" (6TiSCH) (shown in Figure 7) [4]. In TSCH, the nodes must be tightly synchronized (tens of milliseconds [34]), which allows a reservation-based contention-free media access.
[5] shows that no channels are much better or worse than the other, but all channels are not bad at the same time, or as the article concluded "(..) no channel appears significantly better or worse than any other when averaged over time and paths, and the network appears to function well on all frequencies ". And combined with the conclusion form article [2] "The results above suggest that changing frequency when re-transmitting increases the chances of a successful transmission," makes this MAC solution promising.
By using TDMA, we will also get a lower duty cycle which will in turn lead to lower energy consumption. This is important given that nodes should have a lifespan of years or even decades [8].

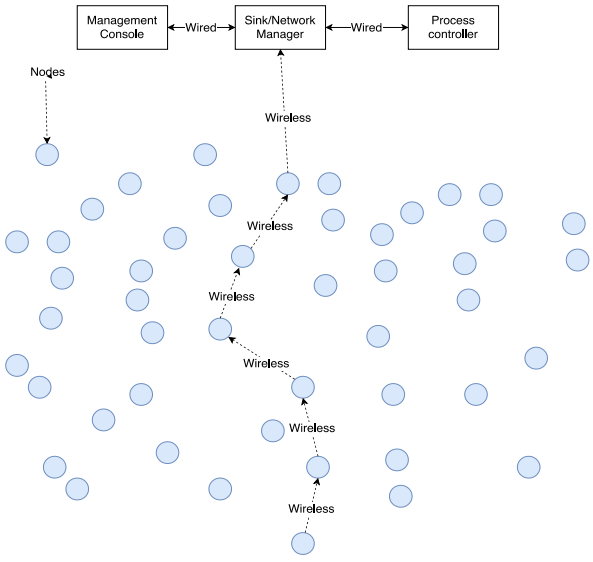

Figure 6: General IWSN

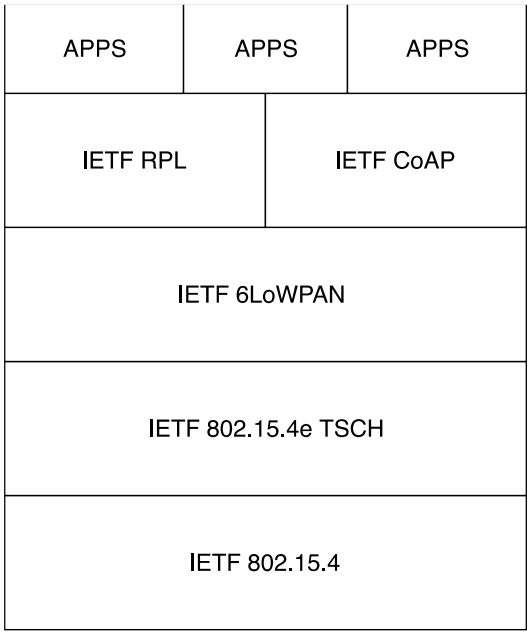

Figure 7: 6TiSCH

2.4 Time Slotted Channel hopping (TSCH)

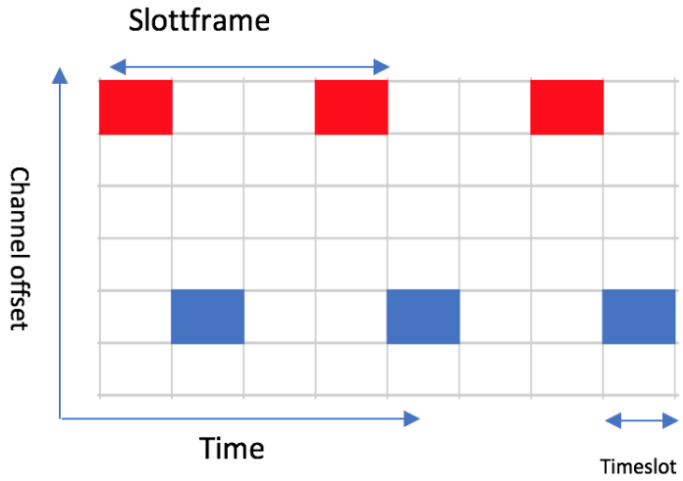

Figure 8: TSCH schedule one radio

Institute of Electrical and Electronics Engineers (IEEE) released a new IEEE 802.15.4e [45] standard who extended the features of the original IEEE 802.15.4 MAC. There are several key features to meet the requirements in factory automation in 802.15.4e:

- Deterministic & Synchronous Multi-channel Extension mode
    - Support of industrial applications which need deterministic latency and higher link reliability
- Time Slotted Channel Hopping mode
    - Process Automation applications with a particular focus on equipment and process monitoring
- Low Latency Deterministic Network mode
    - Support of factory automation and process automation
- Additional Frame Formats
- Low Energy mode
- Enhanced Beacon behavior
- Information Elements
- Miscellaneous – fast association, network metrics, enhanced Acknowledgement frames, channel diversity

One of the key features was Time Slotted Channel Hopping mode. Time-Slotted Channel Hopping (TSCH) mitigates two of the main causes of link failure [3] external interference and multi-path fading [2].

External interference
Occurs when wireless technologies operate on the same frequency band. Simultaneous transmissions will collide and introduce packet loss [2].

Multi-path fading
When a wireless device sends packets, the radio waves will take many different paths to the destination, which will then receive multiple signals. Depending on several factors, this will create a destructive effect that results in packet loss [2].

All sensors in a TSCH network are tightly synchronized in time (tens of milliseconds [34]) [15]. Although the nodes are equipped with clocks, the time of different types of nodes can be different over time. They must therefore be periodically synchronized again. Therefore, both data and acknowledgments packets contain information used to synchronize the clock. This means that neighbor nodes synchronize when exchanging data.

Time is then cut into timeslots, shown in Figure 8 represented by the red box. A timeslot is large enough for a maximum sized MAC frame to be sent from one node to another node, as well as responding with an acknowledgment(ack) frame indicating successful reception [15]. The length of timeslot is not defined in the standard where e.g. WirelessHART has a defined length of 10ms and ISA 100.11.a has variable length. IEEE 802.15.4 radios operate on the 2.4GHz band and have a maximum length of 127 bytes that takes about 4ms to transmit [15], [1]. So, in a 10ms timeslot you use 4ms to transmit, about 5ms for radio turnaround and 1ms to receive ack back [1], [15]. Due to channel hopping in TSCH, a packet loss will lead to a new transmission on another channel. In a timeslot, sensors can either transmit, sleep or listen for incoming data. In timeslots that is not a transmit or receive slot nodes can sleep to reduce power consumption, and if a node has nothing to send in his transmit slot, it can sleep even more to reduce power consumption.

Timeslots are then grouped into a slotframe. Slotframe is continuously repeated over time [15]. Same as timeslots, slotframe length are not defined in the standard and the length can vary from 10`s to 1000`s timeslots long [15]. The length is determining how often the slotframe is repeated, so when it is small, the slotframe will be repeated more frequently. Small slotframe length will give more available bandwidth, but at the same time it will also increase energy consumption since transmit or receive timeslots are repeated more frequently [15].

All this is governed by a TSCH schedule. Schedule indicates the timeslot as dedicated to transmitting or receiving, frequency to be communicated on and the neighbor's address [15]. When a node has received his schedule, he performs it. In its transmit slot it will see if it has any packets in outbound buffer, if it does, it will transmit it and wait for acknowledgment. As mentioned earlier, if a node has nothing in its outbound buffer, it will sleep to reduce power consumption. In his receiving slot a node must wake up and listen for incoming data. A node cannot know if anyone wants to communicate so it must be active, but some implementations have varying length of listening period. So instead of listening in 10ms (timeslot length), it's listening in, for example. 5ms before it assumes no data will be received and sleeps to reduce power consumption. Figure 8 shows an example of a TSCH schedule. Time is cut into timeslots, and four timeslots represent a slotframe that’s repeats over time.

If other nodes want to join, they have to listen for EB (Enhanced Beacon) from already synchronized nodes. EB contains information about timeslot size, current Absolute slot number (ASN), Slotframe information and 1-byte joining priority [15]. The priority is used to make better choices of which node to connect to. ASN is used to synchronized nodes in a wireless sensor network.

ASN is a timeslot counter and increases by 1 each timeslot.
$ASN = (k * S + t)$ [15]

k is number of repeated slotframe since the network started, S represent the length of the slotframe and t is the slottoffset [15]. ASN is a 5-byte number that makes it possible to increase for hundreds of years. Just how long depends on the length of timeslot, where a small timeslot leads to a faster increase in the ASN number [15].

When joining nodes has received ASN, they use this to calculate witch channel to communicate on. All scheduling cells have a specific slotoffset and channeloffset. This means that if node 1 has a transmitting slot to node 2 on channeloffset 1, node 2 will have a receiving slot for node 1 on the same channeloffset, i.e. channeloffset 1. The channel offset is then transformed to a frequency using the function below [15].

$frequency = ((ASN + channelOffset)\ mod\ nFreq)$ [15]

Then we look into a table that contains all available frequencies. nFreq is the size of the table, which means the number of available channels. How many values there is that the channeloffset depends on which band is being used (the 2400-2483.5 MHz band has 16 channels, 11-26) [15].

The fact that ASN increases with one per timeslot causes the nodes to communicate on different frequency each timeslot. This will cause a possible retransmission to occur at a different frequency.

Several literatures have measured the link quality in an industrial environment [2], [5]. The results indicate that no channels are much better or worse than others, but link quality is not bad at all frequencies at the same time. That means sending a retransmission at another frequency will result in less packet loss, leading to less retransmissions, which in turn leads to lower latency and variation of latency (jitter).

The node that has received EB is now part of the network and becomes a synchronizer and advertises EB to other joining nodes [17]. TCSH is used in ISA100.11a, WirelessHART and is used in tens of thousands of networks [1].

Successful channels in channel hopping are separated by at least 15MHz (three channel offsets in IEEE802.15.4) [2]. This is also proved in [2], where they tested if a transmission that failed on frequency 2.435GHz (channel 17) should be retransmitted one channel away, or more? Results showed that if a transmission failed at frequency 2.435GHz (channel 17), the retransmission should be made at least 2.5 frequencies away, that is channel 14 (2.420GHz)

or 20 (2.450GHz). Channel hopping can be done in different ways. The easiest way is blind channel hopping, where every node is hopping over all available channels (16 available channels in the 2400 - 2483.5 MHz band, 11-26). Whitelisting is a more advanced method, where two neighbor nodes agree upon a subset of channels [3]. This is dependent on the nodes themselves to collecting link quality statistics. Article [3] shows that the simplest method of blind channel hopping increases network connectivity by 26%, is 56% is more efficient and increases stability by 38%. Whitelisting can increase network performance even more by avoiding the least optimal channels, and a scheme of 6 channels gives best results.

TCSH was standardized by IEEE in 802.15.4-2015 and selected by IETF for standardization in the “IPv6 over the TSCH mode of IEEE 802.15.4e” (6TiSCH) working group [17].

2.5 Ipv6 over TSCH mode of IEEE 802.15.4e(6TiSCH)

6TiSCH combines IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) [32] and Routing Protocol for Low-Power and Lossy Networks (RPL) [31] with IEEE802.15.4e TSCH MAC [19] layer shown in figure 6. IEEE 802.15.4e TSCH standard do not define how to build and maintain that schedule, so a 6TiSCH operation sublayer (6top) (shown in Figure 9) is defining and standardizing how to manage and build the TSCH schedule [20].

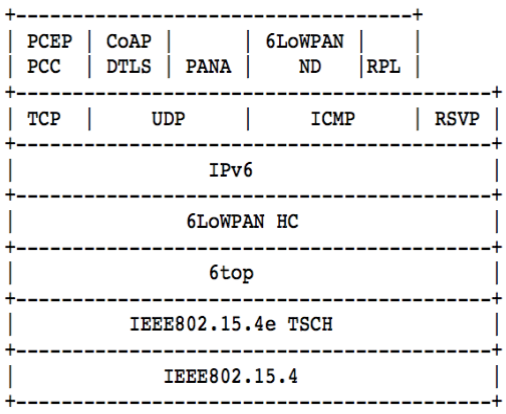

Figure 9: 6top obtained from [20]

6top enables the neighbors to negotiate resource allocation within the schedule and monitoring network statistics. 6top`s roles are to implement and terminate 6top Protocol, which allows neighbor nodes to communicate to add/delete cells to one another [20]. And also run one or more 6top schedules functions, which defines the rules that decide when to add/delete cells [20]. TSCH schedule orchestrates all communication and involves assigning timeslots to communicate. Nodes can have multiple timeslot to increase throughput and lower latency, but that means they have to listen more thereby increasing the energy consumption.

3 Proposal

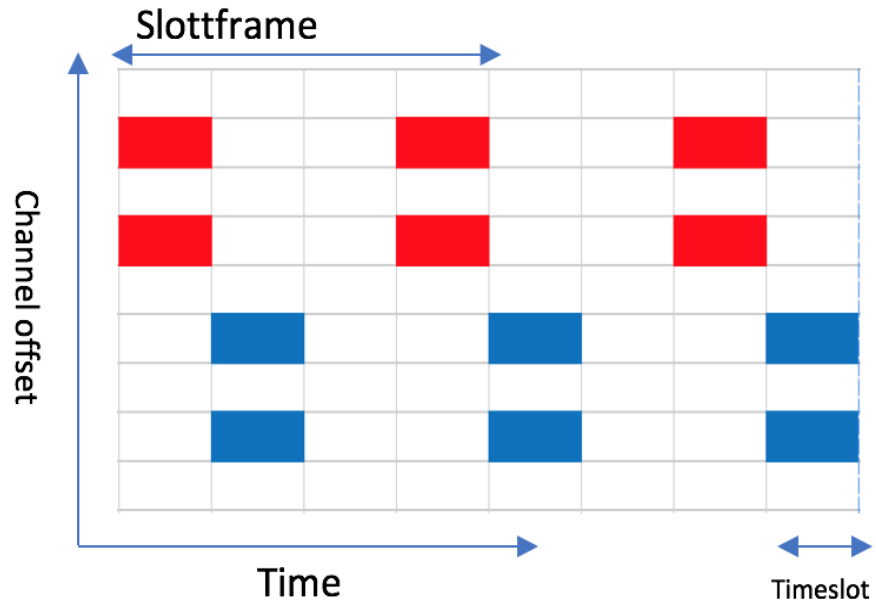

Figure 10: TSCH schedule two radio

The proposal is to use two homogenous radios on each sensor or actuator. By using two homogenous radios on each sensor or actuator each link has different channels. The conclusion in [5] was "No single channel is significantly better than any other and decreases in stability happen on different channels at different times", it might be possible to improve reliability and reduce jitter without the need for retransmissions by transmitting or receiving over to channels simultaneously. By utilizing the same effect that makes channel hopping so efficient (frequency-dependent fading effects), it can be possible to achieve extremely high reliability, and with extremely low jitter and delay. This could be done at the MAC-layer, by creating a TSCH schedule were every link uses two cells in the same timeslot but with different channel offset. This is shown in Figure 10, where the two red squares represent two different channels in the same transmission timeslot (shown downward), and the two blue squares represent two different channels in the same reception timeslot (shown downwards). This means that nodes will send duplicate data at two different frequencies and receive data at one or two different frequencies simultaneously depending on the frequency quality, and reduce packet loss and retransmission, hence reduce latency and jitter (Real-time characteristic).

Using the ASN number obtained through Enhanced beacon (EB) (both discussed in chapter 2.4) when joining, we can use the : $frequency = ((ASN + channelOffset)\ mod\ nFreq)$ [15], function to set channel offset on radio 2 at least 2.5 frequencies away (using result from [2]) two mitigate channel fading.

Using two radios could also give us the opportunity to synchronize faster.
A joining node must listen for EB at a given frequency but do not know what frequency to listen to. The node who searches for EB must also be active during the search period, which will lead to increased energy consumption.
By using two radios, synchronized nodes send out EB at two different frequencies while a joining node will listen on two different frequencies. This will hopefully reduce convergence time considerably. Listening on to two different frequencies using two radios will increase power consumption, but at the same time it could reduce the search period that will reduce power consumption.

The natural disadvantage of this proposal could be battery consumption, bandwidth used and the cost of two radios. But given that WSN gives us a cost reduction up to 90% compared with wired field devices [4], this proposal might be beneficial after all. Some applications have a lower power consumption requirement, where ultra-low delay, jitter and high reliability often come at the expense of power consumption, especially true in closed loops system where requirements are millisecond of delay [23], [27].

3.1 Method

There are several methods for assessing the proposed method. We could use a mathematical analysis. A second alternative is testing in a real-world environment with two radios. Finally, there is the alternative of simulation.

Real-world test has the advantage that it takes into account physical phenomena such as external fading and multi-path fading, but it might also be too time consuming. Software

must be created that's capable of processing both radio at the same time and it must be implemented in layers above. We must also find an environment that is representative of an industrial area.

Mathematical approach has the advantage that it is not too time consuming, but both physical phenomena and duty cycle can become complicated and difficult to calculate.

Simulation will be "easier" as we have control over the environment, but at the same time it is difficult to simulate physical phenomena such as external interference and path fading. One way to simulate External interference is to introduce a node that interfere at different frequencies in the simulation environment.

There are several operating system used on resource-constrained devices in IoT, Contiki-NG, TinyOS, LiteOS, FreeRTOS, MantisOS, Nano-RK and many more. My supervisor suggested Contiki-NG because he heard positive things about the operating system before and some people at the Western Norway University of Applied Sciences (HVL) have some experience in using it. Contiki-NG also has a simulation tool written in Java called Cooja. Here we can add firmware written with C to nodes and measure everything from radio duty cycle and PDR, as well as information about everything that happens in the layers above. COOJA simulation tool is based on the assumption that one node has one radio, so changes must be made to the actual source code of the simulation tool.

However, this might be less time-consuming then creating a Software for a real-world device that can process two radios that transmit and receive at the same time and implemented in the layers above.

I have done literature survey, searching for usage of multi-radios using the Contiki-NG implementation. I did not find any using Oria [44]. I found one publication on GitHub that used "two" radios with different transmission rang, but this was for the old Contiki-OS and it does not look like it supports simultaneous transmitting or receiving [43]

4 Design

I have chosen to use the new Contiki-NG (Next Generation) operating system. It was launched in December 2017 and have better documentation as a predecessor (Contiki-OS [36]). Here we have the opportunity to experiment with TSCH on the MAC layer, several different schedule mechanisms like 6top (6TiSCH, chapter 2.5), Orchestra (visit [11]) and RPL as routing protocol. Contiki-NG has a simulation tool (Cooja) written in JAVA, for simulating different sensor networks and retrieve data about radio duty cycle, CPU usage, PDR and number of retransmissions and much more. This allows developers to test applications or implementations on a fully emulated hardware device, which means they can test code before they try on real hardware. But, because the Cooja simulation tool is based on the assumption that one node has one radio, I need to make several changes to the simulation source code. Because the node in Cooja is a compiled and executable Contiki-NG system, an additional interface must also be added and integrated into the source code of the simulation tool. In order for the new Interface to work, I need to make changes to the radio driver, so it can send or receive data on two different radios simultaneously. I have not come across any good documentation on Cooja, just some introduction articles that explain the very basic.

In the following next, I will first explain how Contiki-NG is designed, what possibilities are offered by the MAC and NET layer as well as routing. Then I will explain my radio design and implementations.

The entire code can be retrieved from:
https://github.com/VegarKrogsethagen/Contiki-NG-Dual-radio.git

4.1 Contiki-NG

Contiki-NG [21] is an operating system used on resource-constrained devices in IoT. Contiki-NG contains an RFC-compliant, low-power IPv6 communication stack, enabling Internet connectivity. It can be run on many different systems based on energy-efficient architectures such as ARM Cortex-M3 and Texas Instruments MSP430.

Contiki-NG started as a fork of the Contiki operating system with five goals obtained from [21]:

1. *Focus on dependable (reliable and secure), standard-based IPv6 communication;*
2. *Focus on modern IoT platforms, e.g. ARM Cortex M3 and other 32-bit MCUs;*
3. *Modernize the structure, configuration, logging and platforms, to reflect the goals above;*
4. *Improve the documentation, both code API, module description, and tutorials;*
5. *Implement a more agile development process, with easier inclusion of new features, and with periodic releases.*

There are some small and significant changes to the operating system from the predecessor Contiki-OS to the new Contiki-NG. The former *core* directory is renamed to *OS*. *Apps* directory is now moved to *OS*, and top-level directory *dev*, *CPU* and *platform* is now under one directory called arch [35].

However, the biggest change is how the network stack is set up.

The networking stack has two main layers: MAC (Medium access Control) layer and NET(Network) layer.

When creating a new project, the choice of MAC and NET layer is clarified in the Makefile file located in your project folder.

The Contiki-NG is open-source and has many different contributors that often cause the documentation to be a little short. And my experience is that, many services that appear to be there are often removed without removing all of the code and information in wiki. I have not been able to find any specific documentation of the code build-up of Cooja (simulation tool) only some user guides that describes the very basic seen in [40], [41], [42]. And questions about documentation of specific services are sometimes answered with "the code is the documentation". I tried to ask the community about implementing Dual-radio in Cooja, without any answers. This resulted in that I spent many weeks/months just understanding the construction of the simulation tool and its connection to the operating system itself.

Next, I will present what options you have in the NET and MAC layer, which routing protocols you have to choose from, and how a Contiki mote in Cooja works.

4.1.1 MAC Layer

Contiki-NG offers four different predefined mac layer settings obtained from [38], [37].

- NULLMAC
    - Layer that does nothing. No transmission or receiving of packages.
- CSMA
    - IEEE 802.15.4 non-beacon-enabled mode. CSMA always-on radio.
- TSCH
    - IEEE 802.15.4 TSCH mode. Globally-sync, scheduled, frequency-hopping MAC. This includes 10ms default timeslot, standard TSCH queues and CSMA-CA [37]. As mentioned earlier, the IEEE 802.15.4e does not define the standard how to build and maintain the schedule, so here you have the ability to implement 6top (from 6TiSH) or Orchestra [11] which is an autonomous schedule.
- OTHER
    - Used when creating/customizing own mac layer

By default, Contiki-NG uses 6TiSCH minimal schedule that use a shared slot for sending and receiving, with a slotframe length of seven and default timeslot of 10ms [37].

The MAC layer is defined by a Makefile which lies in every project folder. Selection of MAC layer is set by MAKE_MAC = (your choice) in Makefile.
No matter which one you choose, you can customize/modify your own MAC with a NETSTAK_CONF_MAC flag in a .h file located in your project folder. If you are choosing MAC_OTHER, the NETSTACK_CONF_MAC flag is mandatory.

4.1.2 NET Layer

Contiki-NG offers three different predefined net layer settings obtained from [38]:

- NULLNET
    - Does nothing. No modification on packets up and down the stack.
- IPV6
    - uIP low-power IPv6 stack, with 6LoWPAN and IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL). RPL is enabled default, and more specified the RPL-lite version. RPL-classic is also possible.
- OTHER
    - Used when creating/customizing own mac layer.

The NET layer is defined by a Makefile which lies in every project folder. Selection of NET layer is set by MAKE_NET = (your choice) in Makefile.
Same as MAC layer, you can specify your own implementation of NET layer with a .h file in your project folder using NETSTACK_CONF_NET flag. The flag is mandatory when using NET_OTHER.

4.1.3 Routing protocol

Contiki-NG offers three different routing protocols obtained from [38].

- NONE
    - No routing protocols.
- LITE
    - RPL-lite implementation of RPL (default)
- CLASSIC
    - Classic implementation of RPL

Select one out of these three routing protocols using MAKE_ROUTING in the Makefile.

Routing protocol is defined by a Makefile which lies in every project folder. Selection of routing protocol is set by MAKE_ROUTING = (your choice) in Makefile.

4.1.4 Contiki-NG mote

Figure 11 shows a simplified overview of how motes in Contiki-NG is built up. Node/Mote could have many different interfaces, this can be e.g. LED, button, sensor and of course a radio. The radio is controlled by a radio driver that is platform specific. This can be anything from Texas instruments CC1200 chip, Bosh Sensortec, to the simulation tool (Cooja) own radio driver. However, all radio drivers must follow the rules defined by Radio.h located in Contiki-NG. Next, there will be a brief explanation of what features the radio driver must contain and what features the MAC layer must contain.

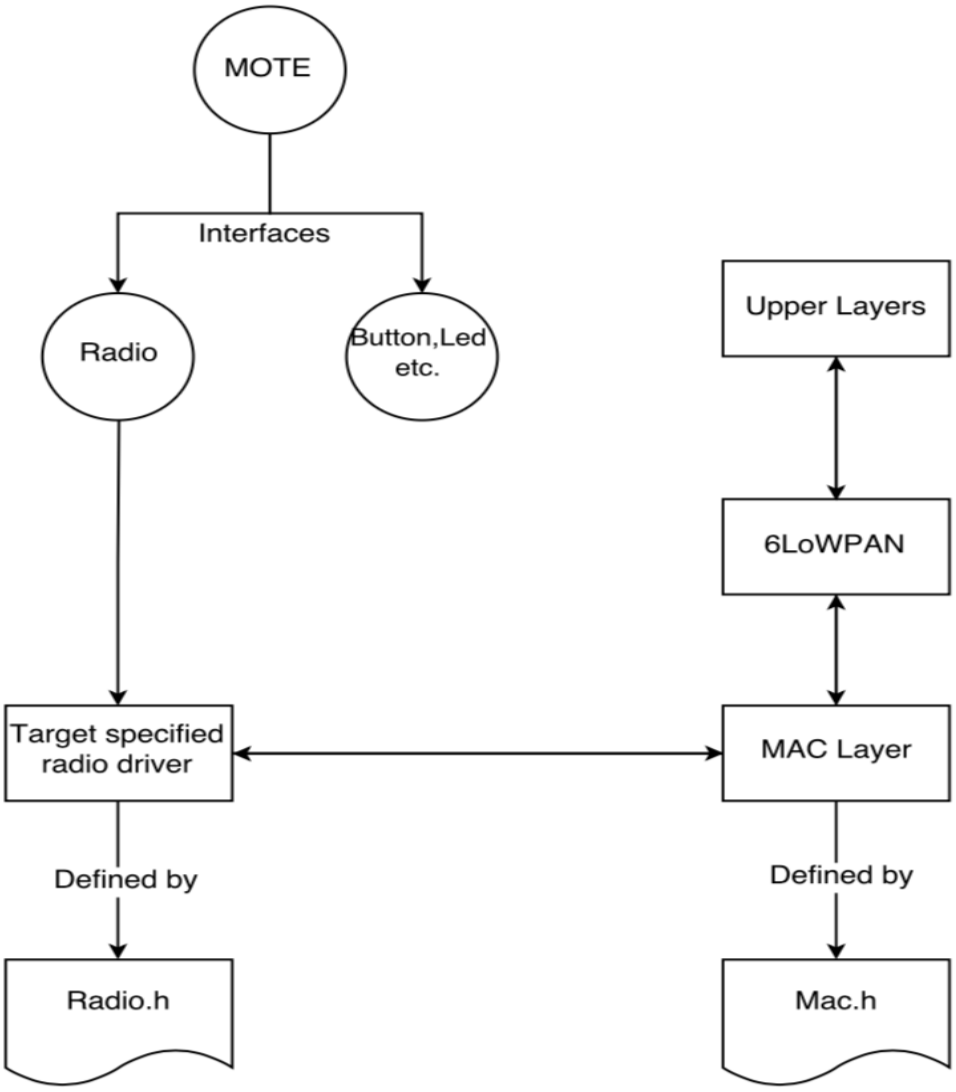

Figure 11 : Contiki-NG design

Radio driver

Contiki-NG uses Radio.h header that defines which rules a radio-driver must follow. This header contains twelve functions obtained from radio.h in Contiki-NG source code:

Init button

- Initialization of the radio driver (threads/processes etc.)

Prepare

- Prepare the radio with a packet to send

Transmit

- Sends the packet that has previously been prepared

Send

- Prepares and transmits a packet.

Read

- Read a received packet into a buffer.

Channel clear

- Preforms a Clear-channel Assessment (CCA) to find out if there is a packet in the air. Used in etc. CSMA. (Carrier sensing)

Receiving packet

- Checks if the radio driver is currently receiving a packet

Pending packet

- Checks if the radio driver has just received a packet.

Get values

- Get radio parameter value

Set values

- Set radio parameter value

On

- Turns the radio on

Off

- Turns the radio off

Mac Layer

Contiki-NG uses a Mac.h that defines which functions the mac-layer must contain.

Obtained from Mac.h in Contiki-NG source code:

Init

- Initializes the MAC driver

Send

- Sends a packet for the packetbuffer

Input

- Notification of incoming packets

On

- Turns MAC layer on

Off

- Turns MAC layer off

4.2 My Design

Since the COOJA simulation tool is built with the assumption that a node has one radio, several changes need to be made. Changes in Cooja`s source code, implement new interface (new radio) on the node itself, as well as changing the structure of the radio driver and changes to the MAC layer. Figure 12 illustrates a relay node that is forwarding data packets. Starting from the bottom where the node receives its data on two different radios. Next the packets are forwarded to the radio driver. When a node receives data, it checks whether the package has been received on both radios, or only one. This is checked through the sequence number in the received package to avoid duplicates. If a packet is received on the radios, the MAC layer is informed. When a packet is sent down from the MAC layer, it uses the send function in the radio driver, which then creates a duplicate of the package before it is sent on both radios. Next, there will be a more detailed explanation of the changes made to the node, followed by an explanation of all the parts in the radio driver and finally, changes to the MAC layer.

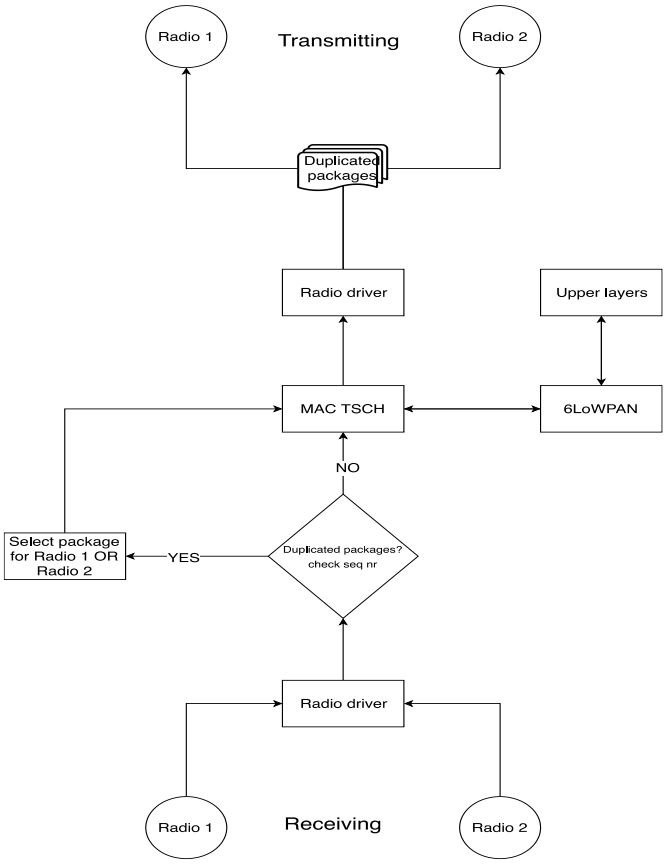

Figure 12 : My design

4.2.1 ContikiMote

I'm using a simulated mote called Contiki Mote. On a Contiki mote we compile Firmware (written in c) and control and analyzes through Cooja (simulation tool). Motes can be analyzed in Cooja with a lot of details that make emulation "slow", or it can be simulated by ruling out unnecessary information that makes it faster (more information visit [40]).
Since a node in Cooja only have one radio at default, I had to attach an additional interface. This radio is called DummyInterface. Here's a simplified explanation of the changes I had to make on the mote, in more detail see appendix B.

I created a new interface on my mote called *DummyInterface*. This interface is based on the default ContikiRadio and an undocumented new_interface example already existing in Cooja. Function in ContikiRadio and DummyInterface are the same, but the variable names are different. E.g. when Cooja-radio-driver calls isReceiving(), function in ContikiRadio will replay with simReceiving and a value form 0 to 1. When the driver calls the same function for my DummyInterface, DummyInterface will replay with simReceivingDummy and a value from 0 to 1. The values represent the state of the radios. 0 means the radio is not receiving something, and 1 means the radio is receiving something.

Both radio-interfaces have its own *doActionAfterTick* functions that's run after every CPU tick. These functions check if the radio is on, changes in the output power, changes in radio channel, ongoing transmission, new transmission, etc.

Both radio-interfaces are then implement/generated on the ContikiRadio from a configuration file.

4.2.2 Cooja-radio-driver

The Radio header contains twelve functions as shown in the chapter 4.1.1. All these functions are implemented in cooja-radio-driver. Unlike the radio-interfaces that checks after mote tick, Cooja-radio-driver checks for changes before mote tick. This function checks for example if the radios are On or Off by calling isRadioOn(), check of the radios are receiving by calling isReceiving(), and if one or both of them have received data. If one or both of them have received something, it starts a process that calls the read function and pass it to the layers above. Now there will be an explanation of all the parts in the radio driver.

Functions that is modified in the Cooja-radio driver:

Read function: Figure 13 shows a flowchart of how the radio driver reads data received on the radios. First, it checks whether one or both of the radios have received data. If nobody has received data, it does nothing, but has one or both of the radios received data it checks if the incoming data exceeds radio buffer set by Cooja. If the data exceeds this limit, the driver will clear the radio buffers. If both radios have received the same data, we select only one of them to avoid duplicates, this is checked by looking at the sequence number. If only one of the radios has received data, the data is handled. Then the MAC layer is notified, and the package is processed in the layers above.

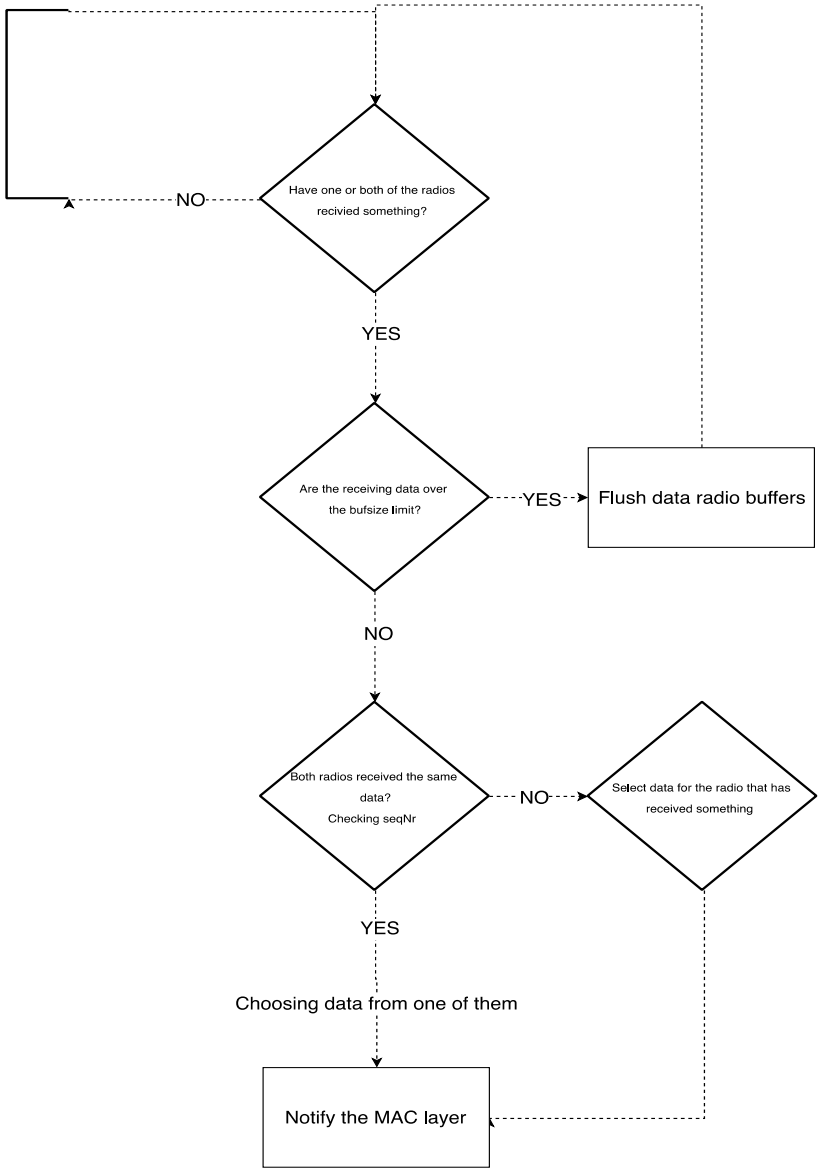

Figure 13: Read function

Send function: Figure 14 shows a flowchart of how the radio driver sends data passed down from the layers above. First, we check if the radios are on, if not we turn them on. Then we look at the package prepared by the layers above and check if the package exceeds the bufsize limit of 127 bytes defined by the TSCH standard (chapter 2.4) or if the package is empty. If it is over the limit or is empty, we will return an error message that is captured by the MAC layer. Then we check the radios already have data in the outgoing buffer, if so we will send an error message captured by the MAC layer. Finally, the data is copied to the radio's outgoing buffer before its transmitted on both radios.

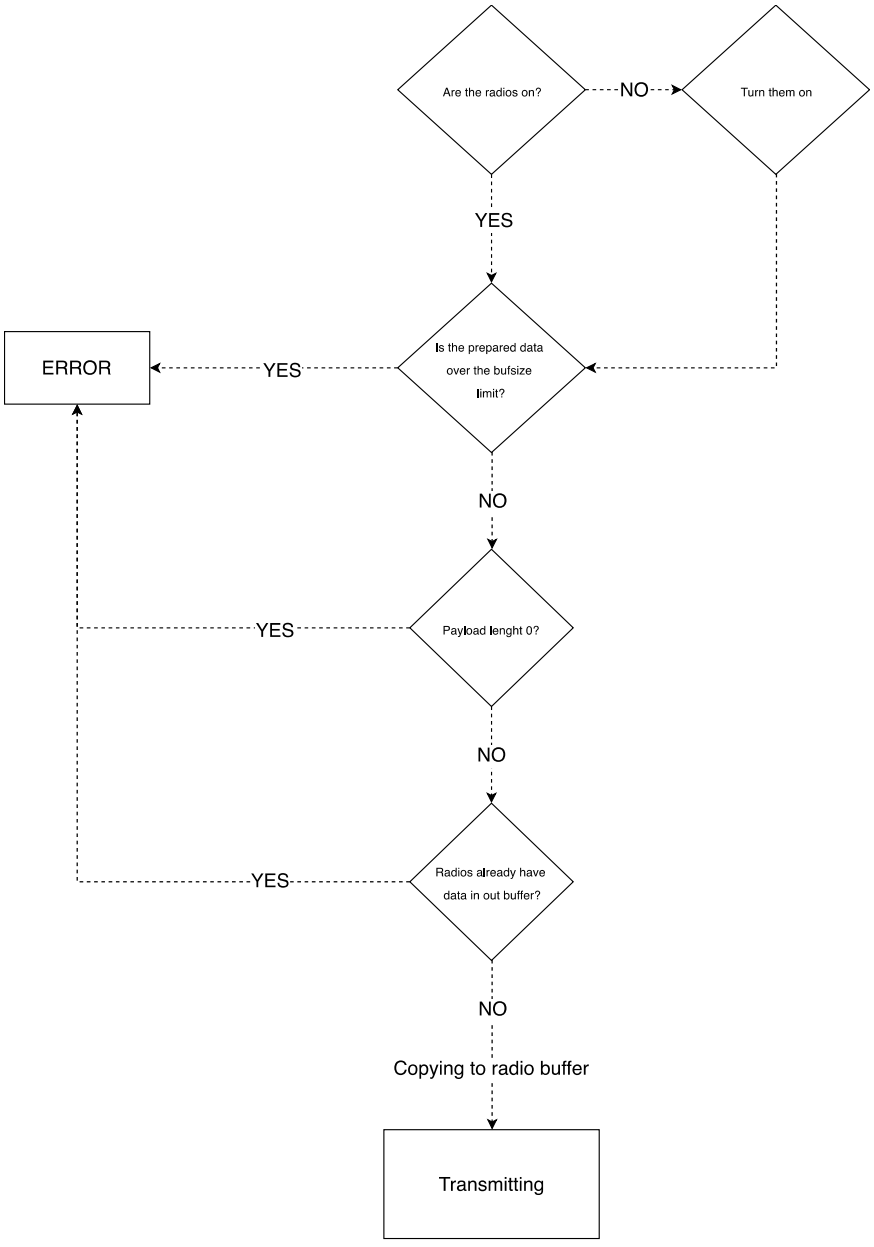

Figure 14: Send function

Pending packet:

The mac layer calls this function to check if the radio is not receiving anything right now but have received something. Instead of only checking ContikiRadio, it checks both radios.

doInterfaceActionsBeforeTick:

Figure 15 shows a flowchart of how the radio-driver checks whether the radio has received data or not. This feature is run after each mote CPU tick. But instead of just checking if one radio has received data, it checks both radios. It first checks if the radios are on or off. If the radios are off, then there is no reason to check if they have received something. It then checks if the radios are currently receiving something, if yes, don't do anything other den updating radios signal strength. If the received data is grater then zero, start a process that run the read function (fig 13).

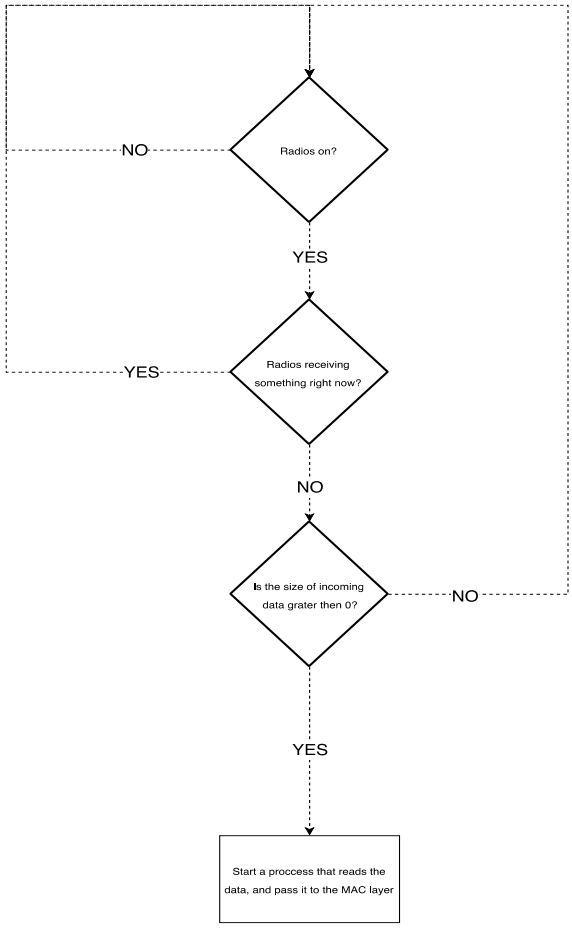

Figure 15: doInterfaceActionsBeforeTicks

Receiving packet function:

The MAC layer checks if the radios are receiving something before it calls the transmit functions which starts the send function. Instead of only checking ContikiRadio and return 0 or 1, it also checks DummyRadio.

4.2.3 MAC layer

Channel selection function:

The TSCH mac layer (TSCH schedule) is responsible for which channel to transmit/receive on. As explained in chapter 2.5, all scheduling cells have a specific slotoffset and channeloffset. This means that if node 1 has a transmitting slot to node 2 on channeloffset 1, node 2 will have a receiving slot for node 1 on the same channeloffset, i.e. channeloffset 1 [15]. The channel offset is then transformed to a frequency using the function below, obtained from Contiki-NG code:

```
uint8_t
tsch_calculate_channel(struct tsch_asn_t *asn, uint8_t channel_offset)
{
  uint16_t index_of_0 = TSCH_ASN_MOD(*asn, tsch_hopping_sequence_length);
  uint16_t index_of_offset = (index_of_0 + channel_offset) %
tsch_hopping_sequence_length.val;
  return tsch_hopping_sequence[index_of_offset];

}
```

Contiki-ng have five predefined schemes from the IEEE standard. One of the schedules is down below:

[16, 17, 23, 18, 26, 15, 25, 22, 19, 11, 12, 13, 24, 14, 20, 21]

The table above contains all available frequencies in 2400-2483.5 MHz band, channel 11-26. By using the Absolute slot number (asn) that increases with one per timeslot, the length of the table above (16 due to 16 available frequencies) and the channel offset that each slot represents, the function returns a channel in the table.

Because we have implemented two radios, the function has to calculate two different frequencies and pass the calculated values down to the radio-driver.

MAC layer calculates channel offset by using the function in chapter 2.5. The MAC layer then calls the set_channel function in the radio driver with the calculated channel offset as argument. Article [2] has done an experiment where they tested if a transmission failed on frequency 2.435GHz (channel 17) should retransmission be done one channel away, or more? The result was that if a transmission failed at frequency 2.435GHz (channel 17), the retransmission should be made at least 2.5 frequencies away, that is channel 14 (2.420GHz) or 20 (2.450GHz). Therefore, radio 2 chooses a channel 5 away from radio 1, we use either uses five channels over, or five offsets below depending on which channel radio 1 uses.

Scanning for EB function:

When a new node wants to connect to the network, it must listen for Enhanced beacon (EB). This is done by selecting a random frequency and listening to it for a given period before selecting a new frequency and doing the same operation. Now it selects two different frequencies and listens for a given period before selecting two new frequencies.

4.2.4 Radio Medium

When creating a new project in Cooja, we must choose which radio medium we want. Contiki-NG offers different radio medium, all of them have different attribute when it comes to e.g. package loss. Where one medium has random packet loss by creating a random number and depending on the generated number drops the packet. They all extends from AbstractRadioMedium. Because Cooja assumes that one mote has only one radio, I had to do some modifications.

Cooja have a MoteInterfaceHandler which AbstractRadioMedium uses to registers motes interfaces to the Radio-medium of your choice. I created a new object for DummyRadio, and a function to get the object for AbstractRadioMedium (getRadio and getDummyRadio). When we add a new mote to the radio-medium, AbstractRadioMedium calls these new functions and puts all radio interfaces in a list. This list is used to go through all potentials destinations.

I also had to modify the AbstractRadioMedium when potential source and destination is evaluated. Default it only checks if the receiver and transmitter is the same radio, but that’s not enough. I implemented a getPosition() function that return the position of the mote/radio. This is because we don't want packets sent from one radio source to be transmitted to the other radio on the same mote. This is not necessary in a real-world device, but because this is a simulation it looks true all potential destinations before transmitting.

5 Result and analysis

5.1 Background information

5.1.1 Simulation information

In this chapter we will review three different simulations using COOJA simulation tools. Here we will measure Packet Deliver Ratio (PDR), number of retransmissions, connection time (joining time), duty cycle and latency.

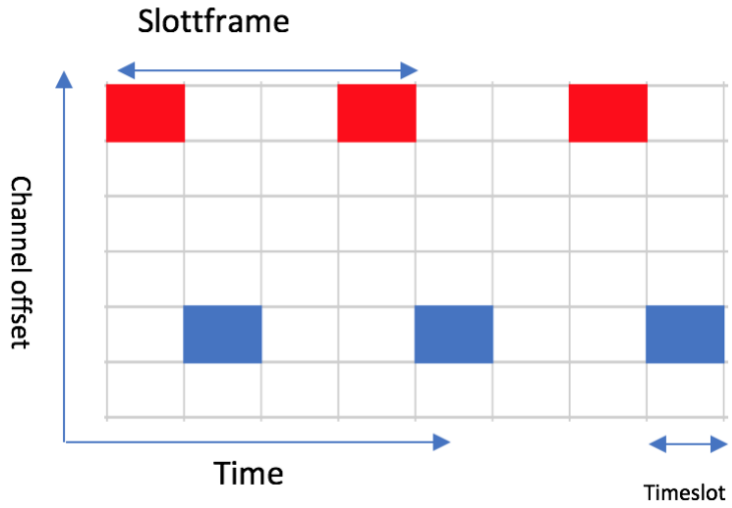

Figure 16: Calculating ASN

Latency is calculated using Absolute slot number (ASN). One slotframe represents a set of timeslots (seven in my experiments), and the slotframes are repeated over time. Looking at Figure 16, the red box is the first timeslot and blue is next timeslot, and one timeslot has a length of 10ms. By calculating how many timeslots it takes before a package is received at the recipient we can calculate latency.
ASN is synchronized in a TSCH network so all nodes will have the same ASN number.
$ASN = (k * S + t)$ [15]

I use the 6TiSCH minimal schedule that uses one shared timeslot for transmitting and receiving. Maximum number of transmissions per package is set to 7 by Contiki-NG, which

is also defined in the standard. This is because applications often prioritize new data instead of guaranteeing receipt of all packets [8].
The point of using ASN to look at latency is to look at the context, where increased retransmission increases latency calculated in number of timeslots before the package is received.

In all simulations, radios have these configurations:

Transmission range	50 m
interference range	100 m
Radio bitrate	250.00 kbps

5.1.2 Disturber node

Do to the fact that typical fading events are frequency-dependent and do not span all available channels at the same time, I created a disturber node. Disturber node is placed within interferences range of the transmitting or receiving nodes. It sends out packets continuously with a random duration between 1-10ms (10ms is the Contiki-NG default timeslot length). The node also generates a random number between 11-26 (available channels in the 2400-2483.5 band), this number is used to set which channel the disturber node is to interfere on.

5.2 Reliability and latency

One simulation equals 10 000 packets being sent from “client” to “server”. TX 1 means one transmission, TX 2 means two transmissions (or one retransmission), TX 3 means three transmissions (or 2 retransmission) etc.

5.2.1 UDP Client node

MAC	TSCH
TSCH Schedule	6TiSCH Minimal
Slotframe length	7
Payload size	64 bytes
Timeslot length	10ms
EB sending period	16sec
Number of packages	10 000
UDP package sending period	1sec
Routing protocol	None

Table 2: Client node Packet delivery ratio (PDR)

Client node listen for EB for synchronization. After that the client starts sending out UDP package to the server every second. The UDP package contains current ASN numbers derived from client node as payload. I created a java script to save information printed from the nodes, shown in Appendix C.

5.2.2 UDP Server node

MAC	TSCH
TSCH Schedule	6TiSCH Minimal
Routing protocol	None

Table 3: Server node Packet delivery ratio (PDR)

Server node is set as coordinator and starts sending out EB for synchronization. Then listens for incoming packages. Server counts the number of received UDP packets and checks sequence numbers to exclude duplicate packages.

When the client transmitted a UDP packet, he placed his current ASN's number in payload, and the server can then use this number and compare it with current ASN numbers. By calculating the difference, we find latency calculated in ASN.

Server node use PowerTracker to collect information about radio duty cycle. I created a java scripts to save information from the server, shown in Appendix C.

5.2.3 Results and analysis

Environment 1: No Interference

The first test is to verify the changes made in the simulation tool and test the hypothesis that in a perfect environment without interference, two radios will not improve Packet Deliver Ratio (PDR), reduce number of retransmissions hence reduce latency, but only have a higher energy consumption.

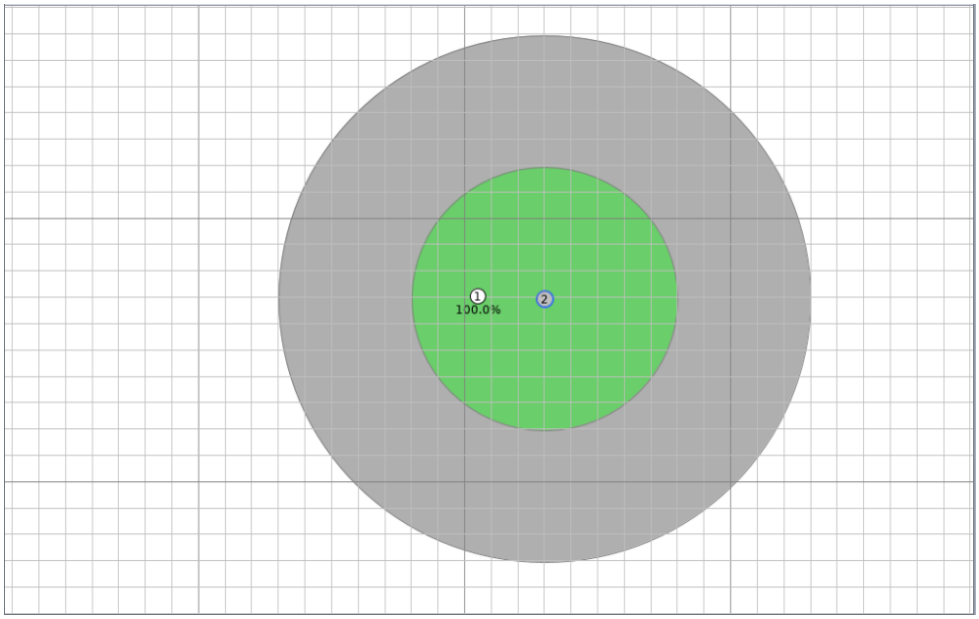

Figure 17 : Environment 1

Figure 17 shows the simulation environment. Node 1 represents client and is located 20m away from node 2 representing the server.

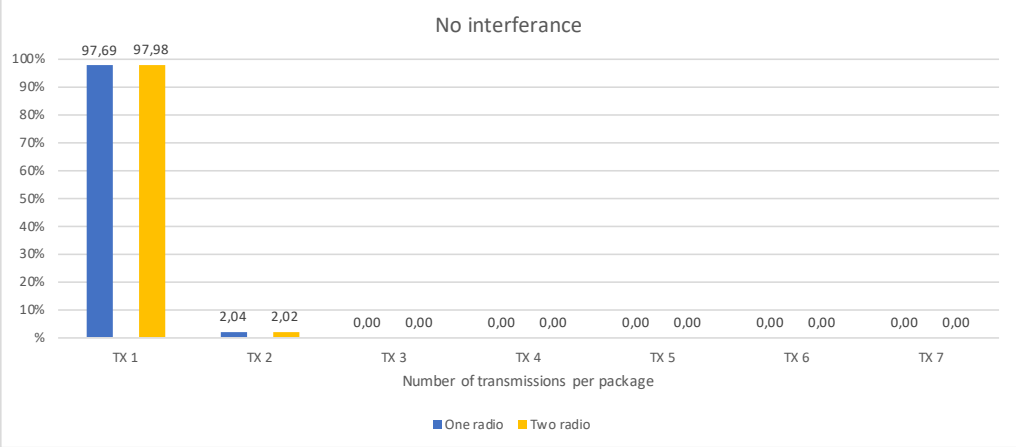

Figure 18 : Retransmissions with no interference. The figure shows percentage in Y-direction and number of retransmissions in X-direction

Figure 18 shows, as expected, that in a perfect environment without any disturbances, nodes with one or two radios get more or less the same result. We see that some of the packets need one retransmission, this is due to the 6TiSCH minimal schedule that uses a shared slot so EB collisions can occur. We verified this by lower the rate at how often nodes emit EB from every 16s to 32s. This is showed in Figure 19 where we get a higher percentage of packets without retransmissions. The rest of the simulations will be with the standard EB cycle of 16s.

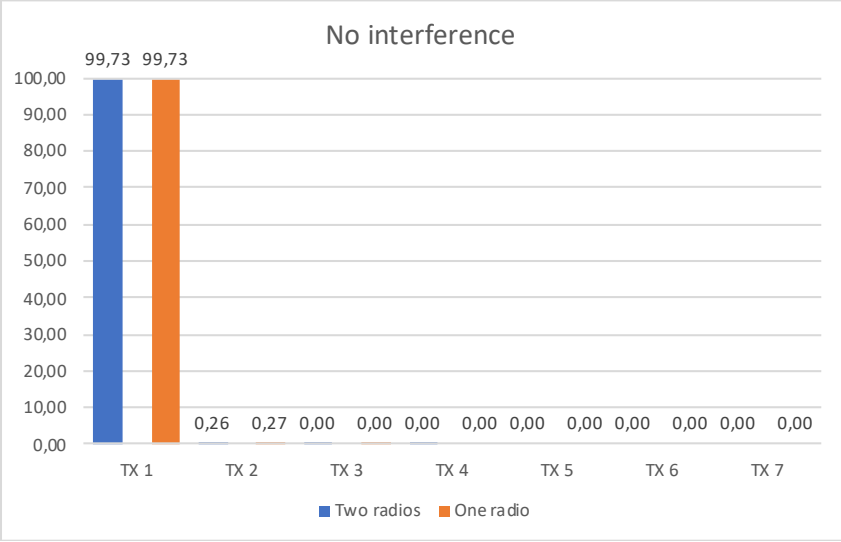

Figure 19: EB every 32s. The figure shows percentage in Y-direction and number of retransmissions in X-direction

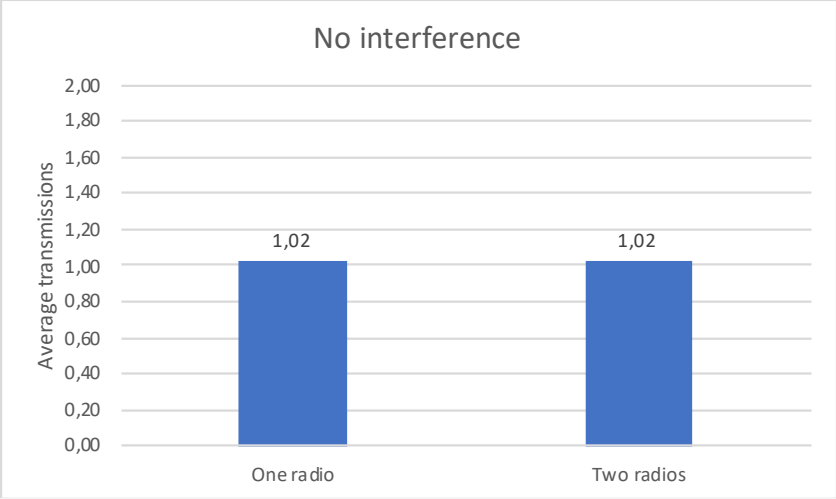

Figure 20: Average transmissions no interference. Figure shows average transmissions per package in Y-direction

Figure 20 shows that nodes with one radio and two radios need exactly as many transmissions per package. We can also read this from the figure 18 which shows that the percentage of packets sent without retransmission is equal using one and two radios.

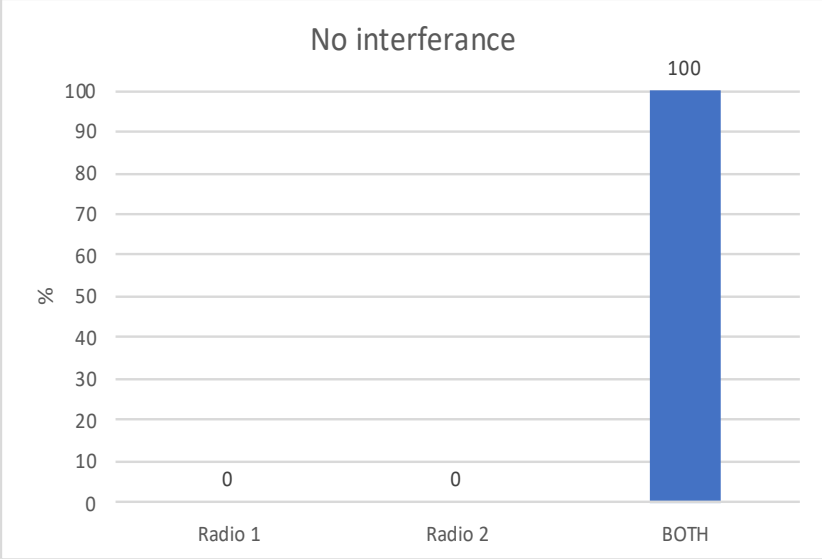

Figure 21 : No interference radio distribution. The figure shows percentage in Y-direction.

Figure 21 shows that all packets sent are received simultaneously on both radios. This makes sense because both channels that are communicated on are always without disturbing elements. This also confirms our implementation.

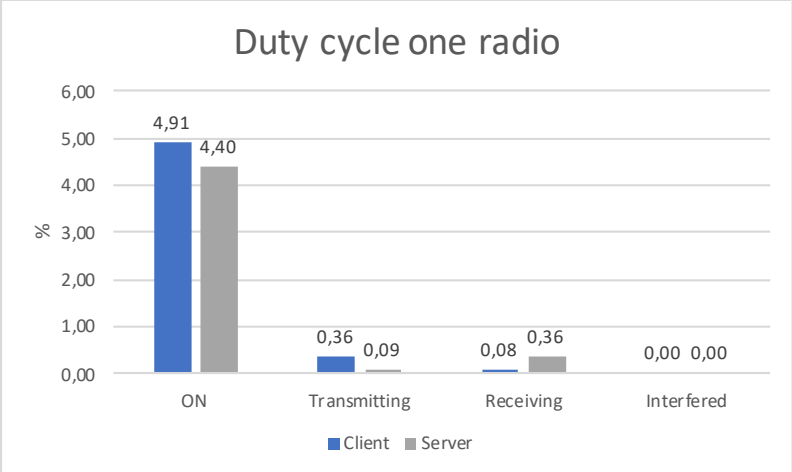

Figure 22: Duty cycle one radio no interference

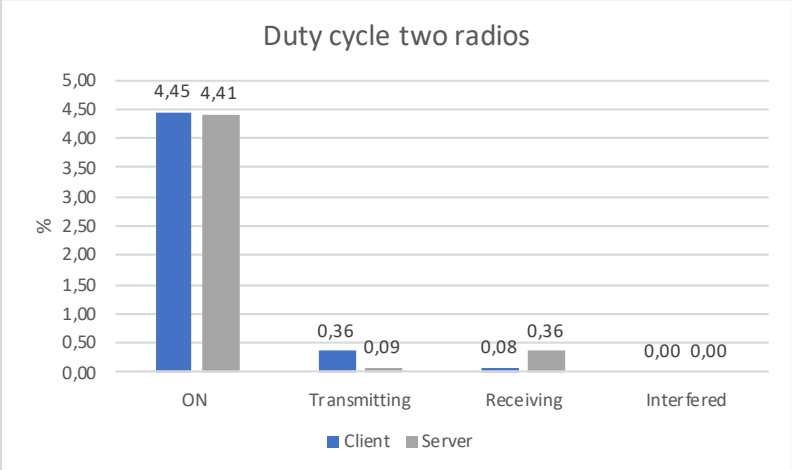

Figure 23: Duty cycle two radios no interference

If we compare the duty cycle for nodes with one radio and two radios (figure 22 and 23), we see that nodes with one radio has somewhat higher duty cycle. This is due to some retransmissions nodes with one radio must do. However, due to the fact that dual-radio nodes use twice the amount of power on transmission and receiving, power consumption will still be higher using two radios.

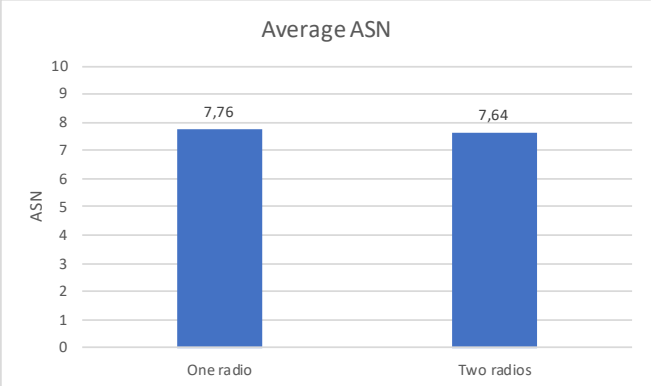

Figure 24: Average ASN. Figure show average ASN in Y-direction, and radios in X-direction.

Figure 24 shows that without any kind of interference, latency using one radio vs. two radios is more or less the same. This can also be seen in figure 18 where the number of transmissions per packet is equal. When they need as many retransmissions per package, latency will be similar.

Environment 2: One disturber

In test number two we will test the hypothesis that in an environment with a disturber node that interferes on one random channel, two radios will improve packet deliver ratio (PDR), reduce the number of retransmissions hence reduce delay. This is because we communicate at two different frequencies at the same time, but there is interference on one channel at a time.

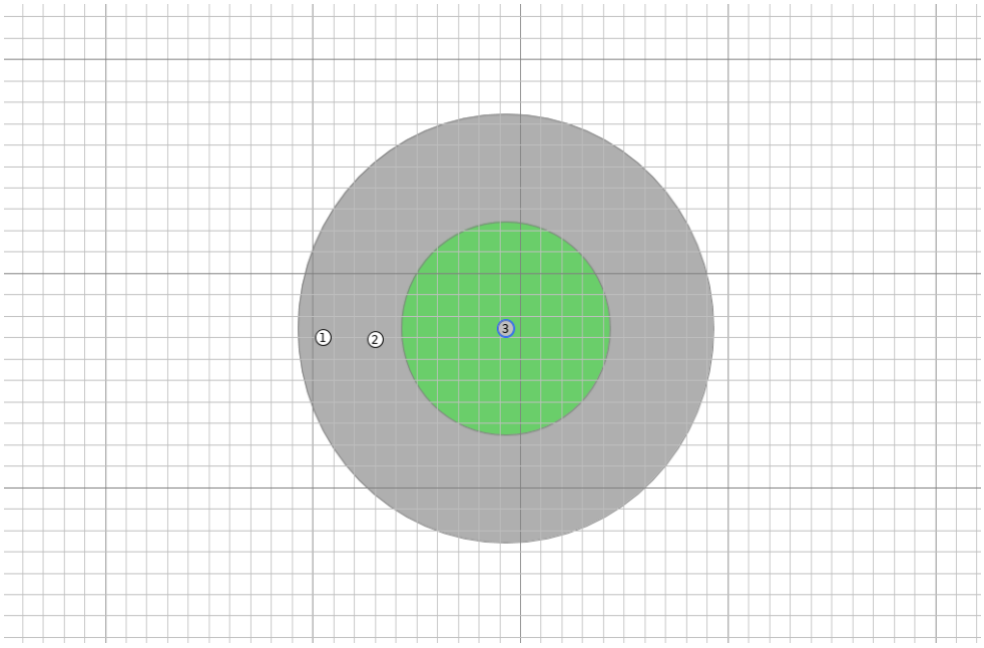

Figure 25 : Environment 2

Figure 25 shows the simulation environment. Node 1 represents client and is located 20m away from node 2 representing the server. Node 3 is a disturber node in interference range of both nodes.

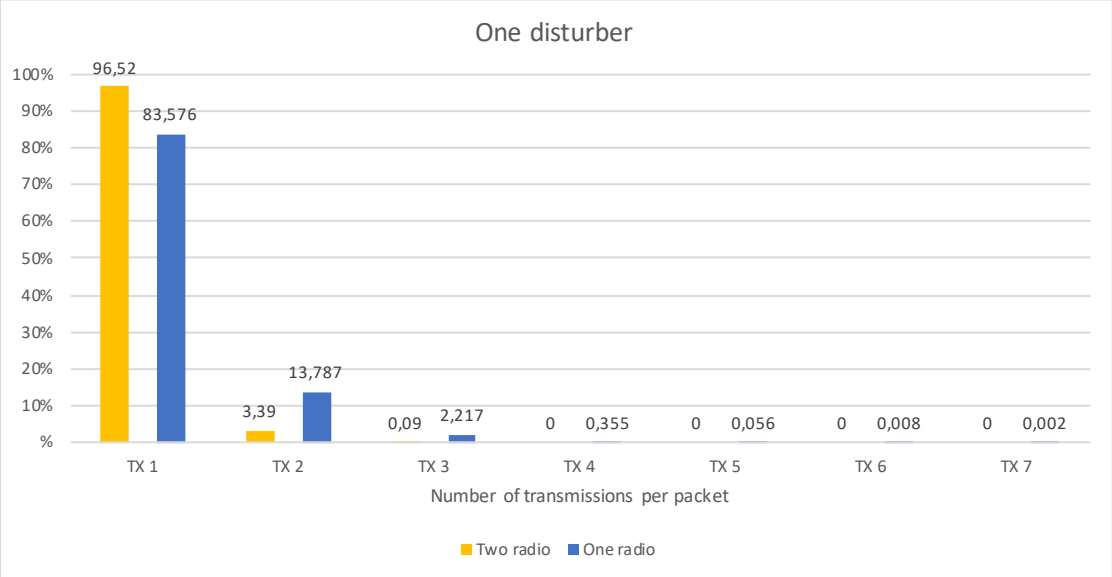

Figure 26: Data one disturber. Figure shows percentage in the Y-direction and the number of retransmissions in the X-direction.

Figure 26 shows that when we introduce a disturber node that randomly interfere on different channels in random durations, we get a significant increase in number of retransmissions using only one radio. If we look at the proportion of packets received with one retransmission, nodes with one radio have increased by 575.98% compared to environment without interference, while using two radios, it has "only" increased by 67.98%. Result shows that when using one radio we have packets that exceed the limit of seven transmissions (the limit set by the TSCH standard), hence packet loss, but using two radios we have a worst case of three transmissions (or two retransmissions). The reason for an increase in retransmissions using two radios is that even though the disturber node only interferes on one channel at a time, it has a random duration that allows it to interfere over a timeslot at two different frequencies. If it has a duration of e.g. 5ms, it could interfere 5ms on one channel and 5ms on another channel therefor interfere over one timeslot (10ms).

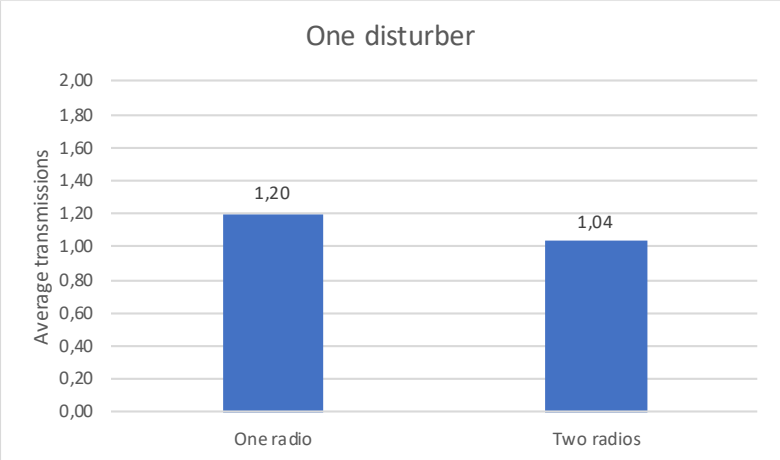

Figure 27: Average transmissions one disturber. Figure shows average transmissions per package in Y-direction

Figure 27 shows that nodes with one radio needs 1.2 transmission per packet on average, while two-radio nodes need 1.04 transmission per packet on average. Compared with environment without interference, transmission per packet has increased by 17.65% with one radio and 1.96% using two radios.

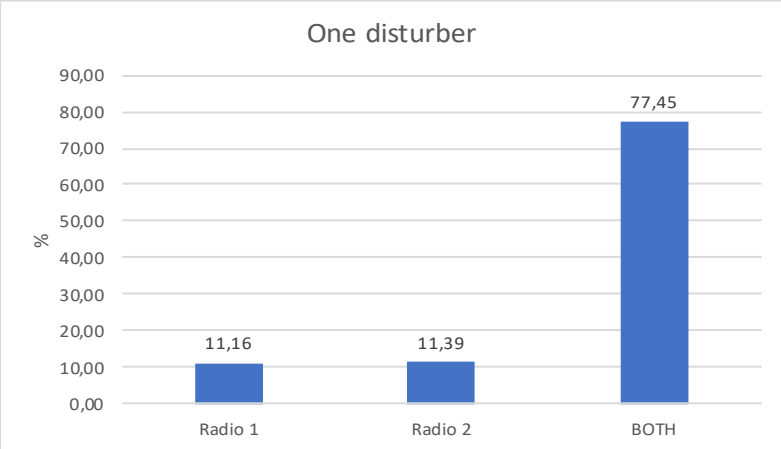

Figure 28: One disturber radio distribution. Figure shows the percentage in Y-direction.

Figure 28 shows that not all packets are received on both radios. 22.55% of the packets were only received on one of the radios. Looking at figure 26 again, we see that the proportion of packets that did not require retransmissions has decreased, but by using two radios, it has not decreased more than 1.49%.

This shows that two radios have an impact when we introduce disturbances.

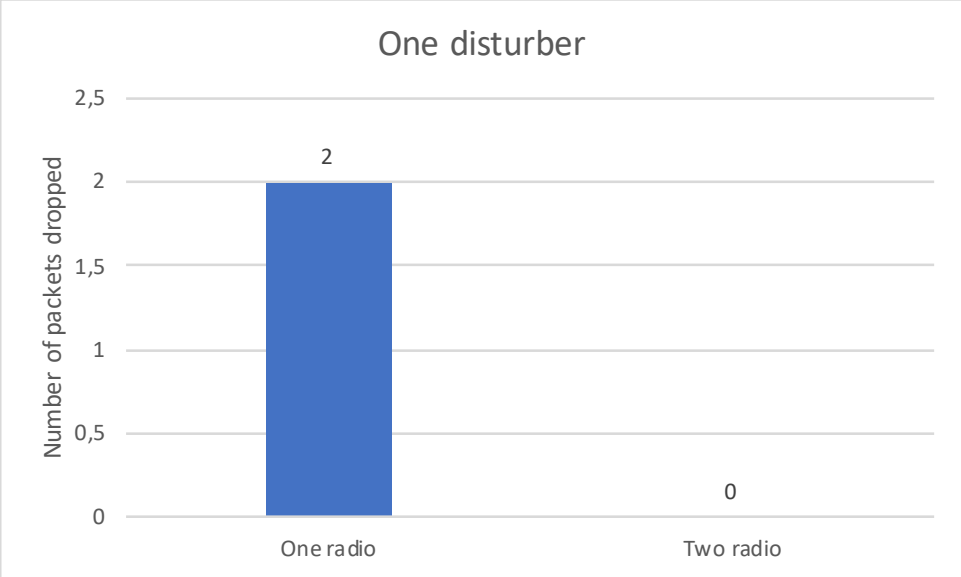

Figure 29 : One disturber, packet dropped

Figure 29 shows that with one radio, two packets are being dropped because they exceed the limit of seven retransmission, but with two radios all packets are received.

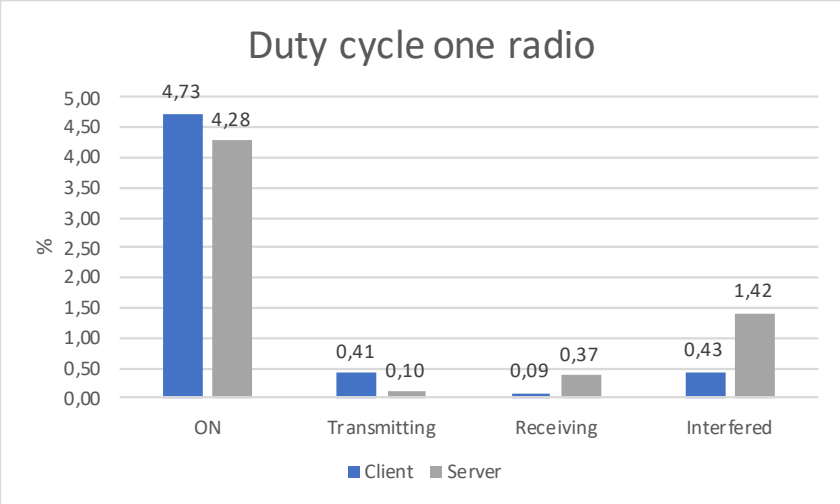

Figure 30: Duty cycle one radio one disturber. Percentage in y-direction, and radio modes in x-direction.

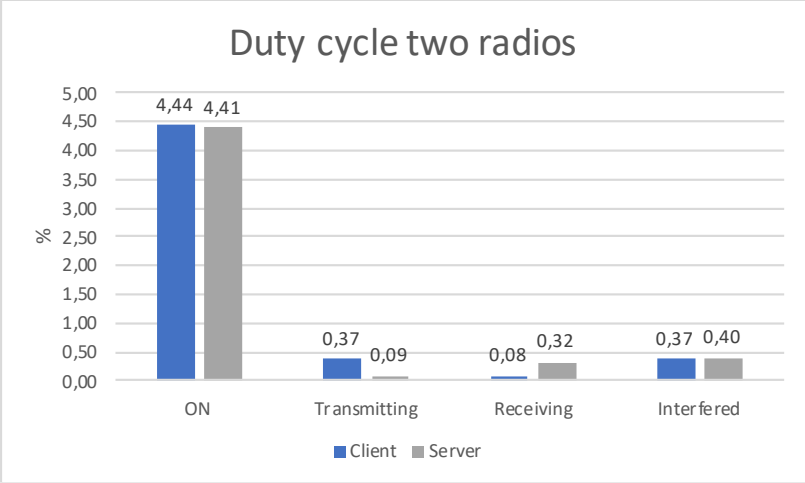

Figure 31 : Duty cycle two radios one disturber. Percentage in y-direction, and radio modes in x-direction.

Fig 30 and fig 31 show that the duty cycle for nodes with one radio differs from nodes with two radios. The reason for that is all retransmissions nodes with one radio must do before package is received. We see that the node with one radio has been 0.41% of the time in transmission mode, but with two radios it has "only" been in transmission mode 0.37% of the time. The same increase can be seen in receive mode.

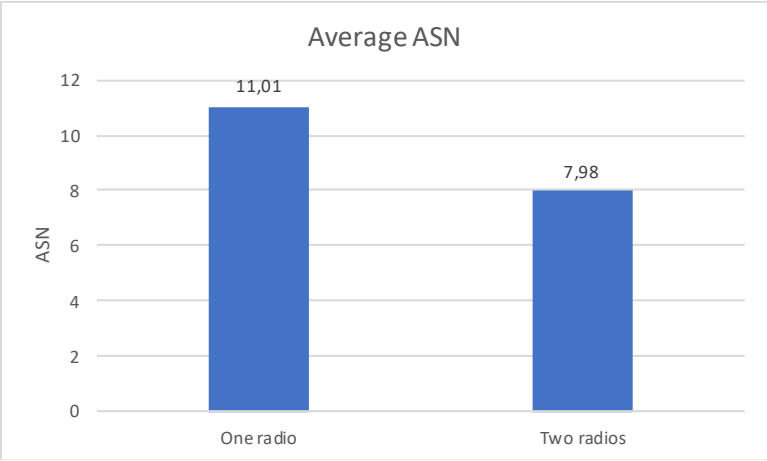

Figure 32: Average ASN. The figure shows ASN on average in Y direction, and radios in X direction

Figure 32 shows that due to retransmissions, we get an increase in average ASN. Average ASN when using one radio have increased by 41.88% and with two radios 4.45% compared to environment without interference. As we have discussed before, latency increases when we need to use retransmissions, so when we get a reduction in the proportion of packets sent without retransmissions, latency will naturally also increase.

Environment 3: Two disturbers

In test number three we will test the hypothesis that in an environment with two disturber nodes that interferes on two different channels simultaneous, will we not only see an increase in retransmissions using one radio but also using two radios. But we will still see that two radios improve packet deliver ratio (PDR), reduce the number of retransmissions hence reduce delay.

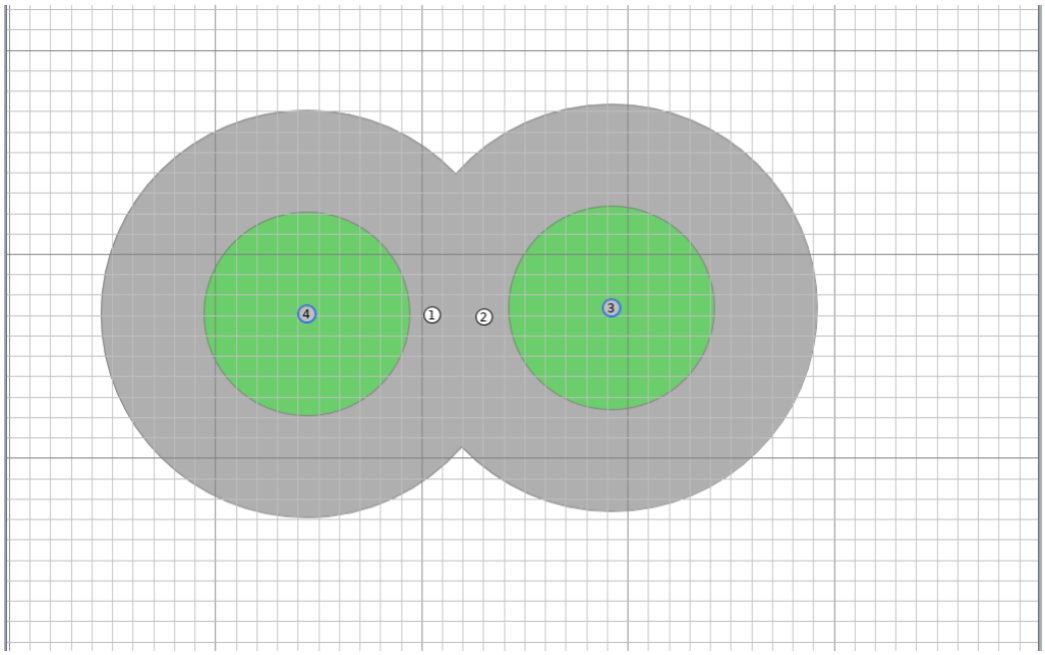

Figure 33 : Environment 3

Figure 33 shows the simulation environment. Two nodes are placed 20m apart from each other (node 1 and 2). Disturber nodes (3 and 4) are placed within interference range of the two nodes.

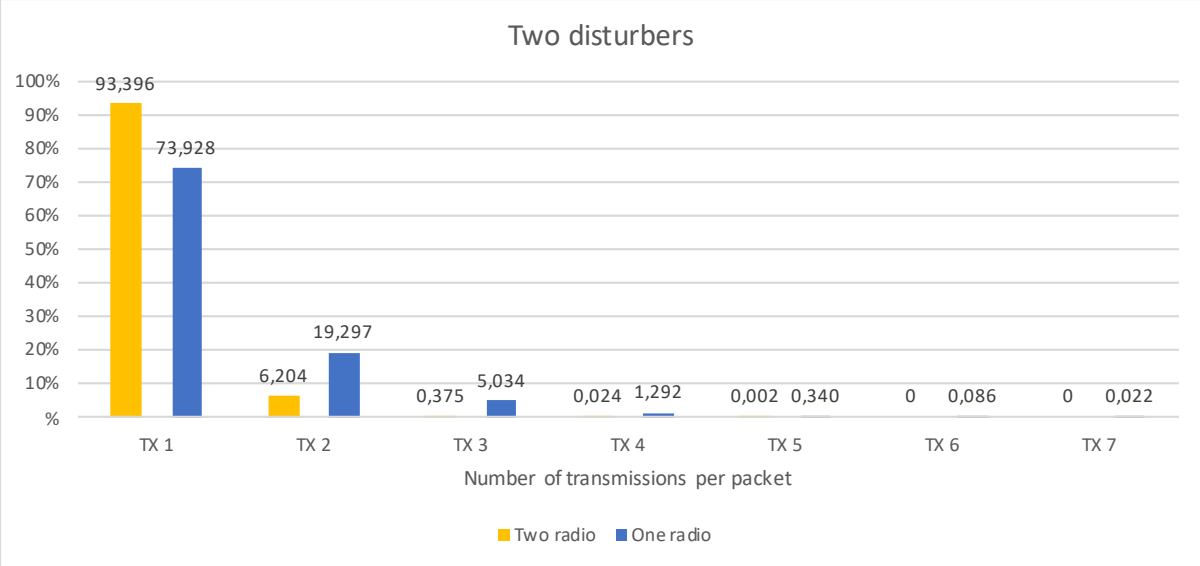

Figure 34 : Data Two disturbers. Figure shows the percent in the Y-direction and the number of retransmissions in the X-direction

Figure 34 shows that when introducing two disturber nodes, the number of retransmissions is increasing even further. Proportion of packets sent without retransmissions have fallen by 23.75% using one radio and 4.67% using two radios compared to environment without interference.

We also see that we have a worst case of five transmissions (or four retransmissions) using two radio and packet loss using one radio.

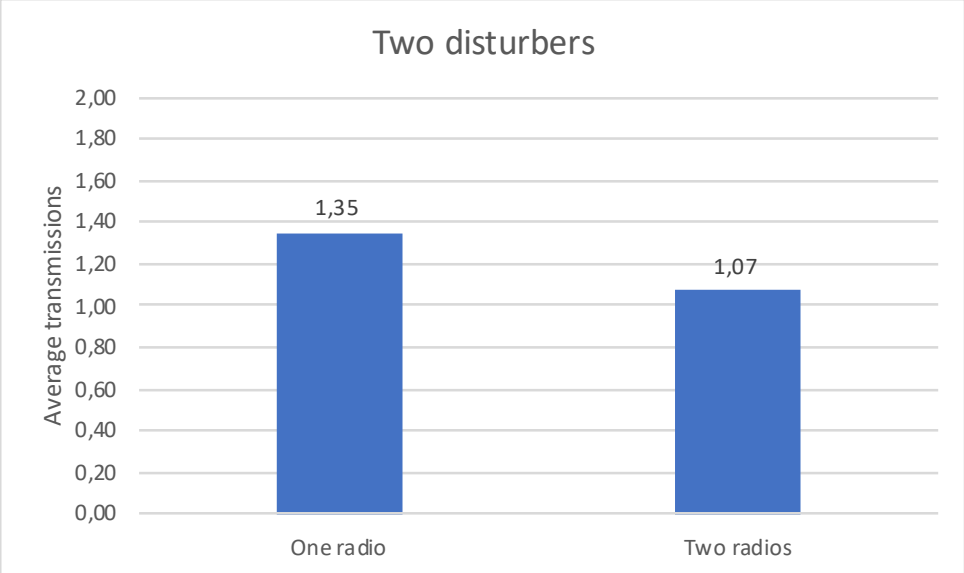

Figure 35: Average transmissions two disturber nodes. Figure shows average transmissions per package in Y-direction

Figure 35 shows that nodes with one radio needs 1.35 transmission per packet on average, while two-radio nodes need 1,07 transmissions per packet on average. Compared with

environment without interference, transmission per packet has increased by 32.35% with one radio and 4.90% using two radios.

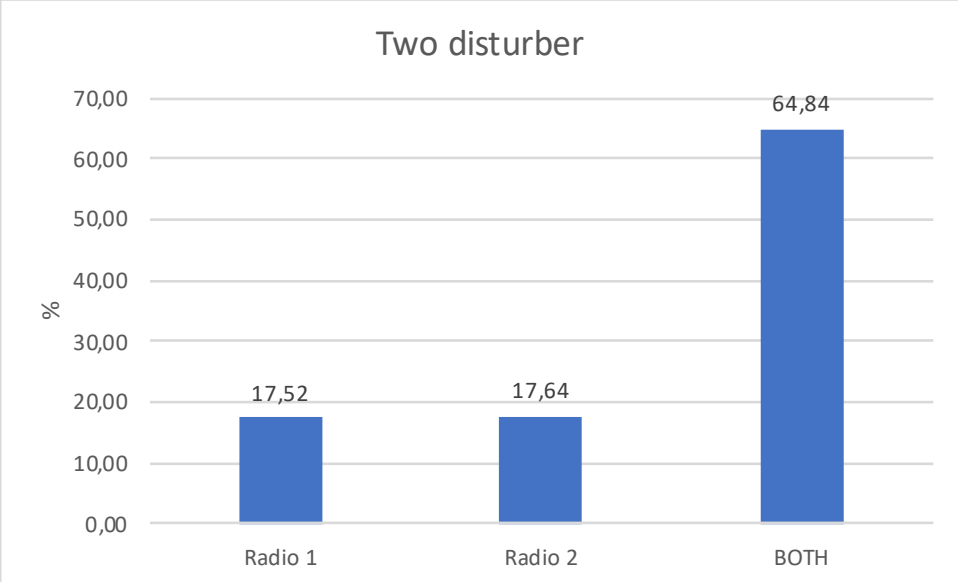

Figure 36 : Two disturber radio distribution. Figure shows the percentage in y-direction.

Figure 36 shows that not all packets are received on both radios. 35.4% of the packets were only received on one of the radios. This shows that two radios have an impact when we introduce disturbances.

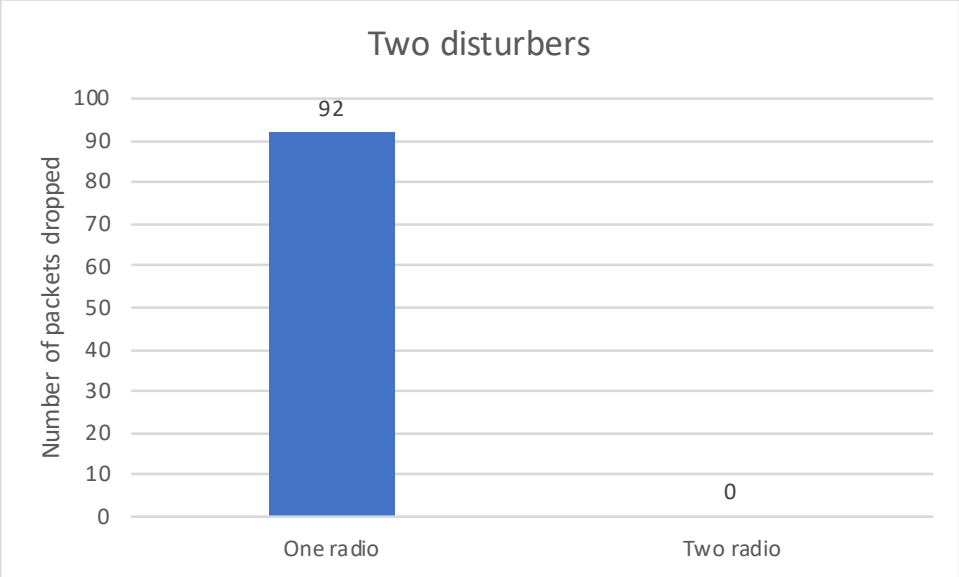

Figure 37 : Two disturbers packets dropped

Figure 37 shows that we get 92 (or ~1%) packet loss using one radio, and 0 when using two radios.

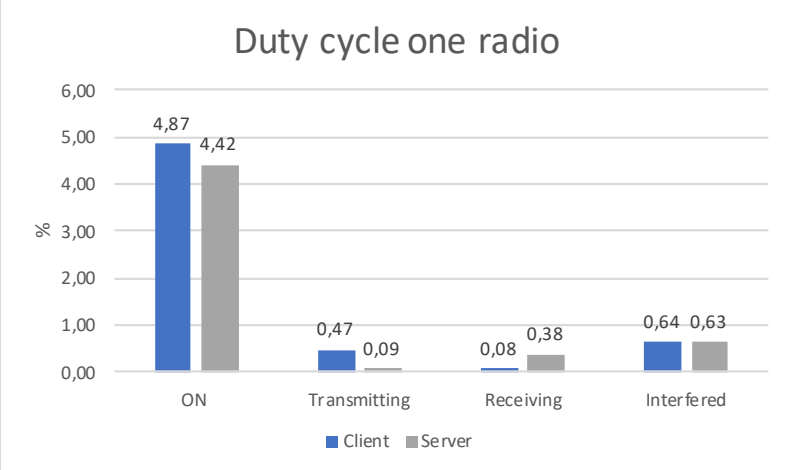

Figure 38: Duty cycle one radio two disturbers. Figure shows percentage in Y-direction and radio modes in X-direction

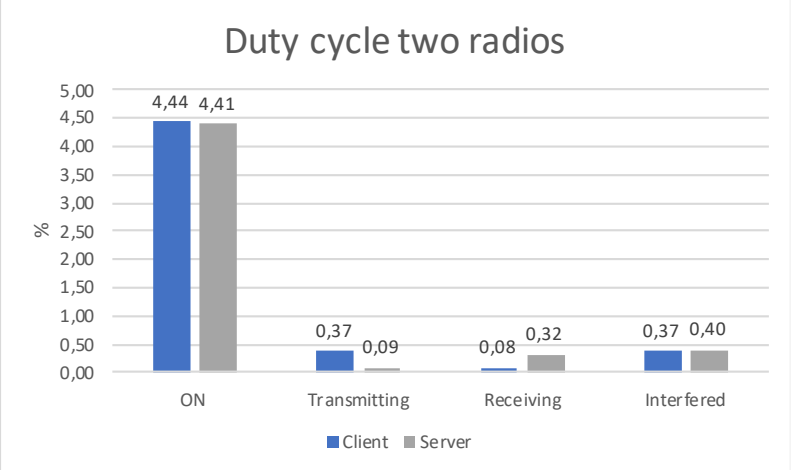

Figure 39: Duty cycle two radios two disturbers. Figure shows percentage in Y-direction and radio modes in X-direction

The node with one radio is more often "ON", respectively 4.87% versus 4.44% to radios(figure 38 and 39). This is because of all the retransmissions node with one radio must do as we can see in the columns Transmitting (0.47% vs 0.37%) and Receiving (0.38% vs 0.32%). The client must send the same package multiple times, while the server might receive more. This may be because the packet never arrived at the server side, or that ack was not received at the client side.

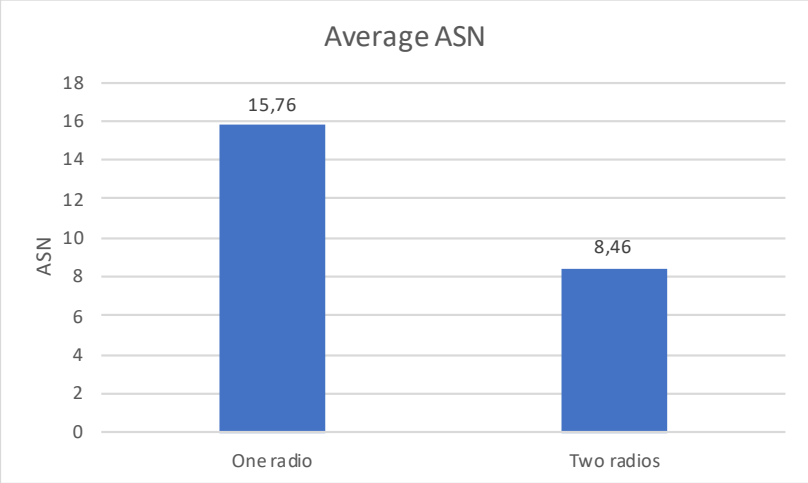

Figure 40: Average ASN. Figure shows average ASN in Y-direction and radios in X-direction

Figure 40 shows that due to retransmissions, we get an increase in average ASN. Average ASN when using one radio have increased by 103.09% and with two radios 10.73% compared to no interference simulation.

Environment 4: Three disturbers

Test number four will be in an environment filled with interference. We introduce three disturber nodes that interfere on three or less different frequencies. Here we will see that both radio and dual radios will get more retransmissions that lead to more latency, but nodes with two radios will still get better results.

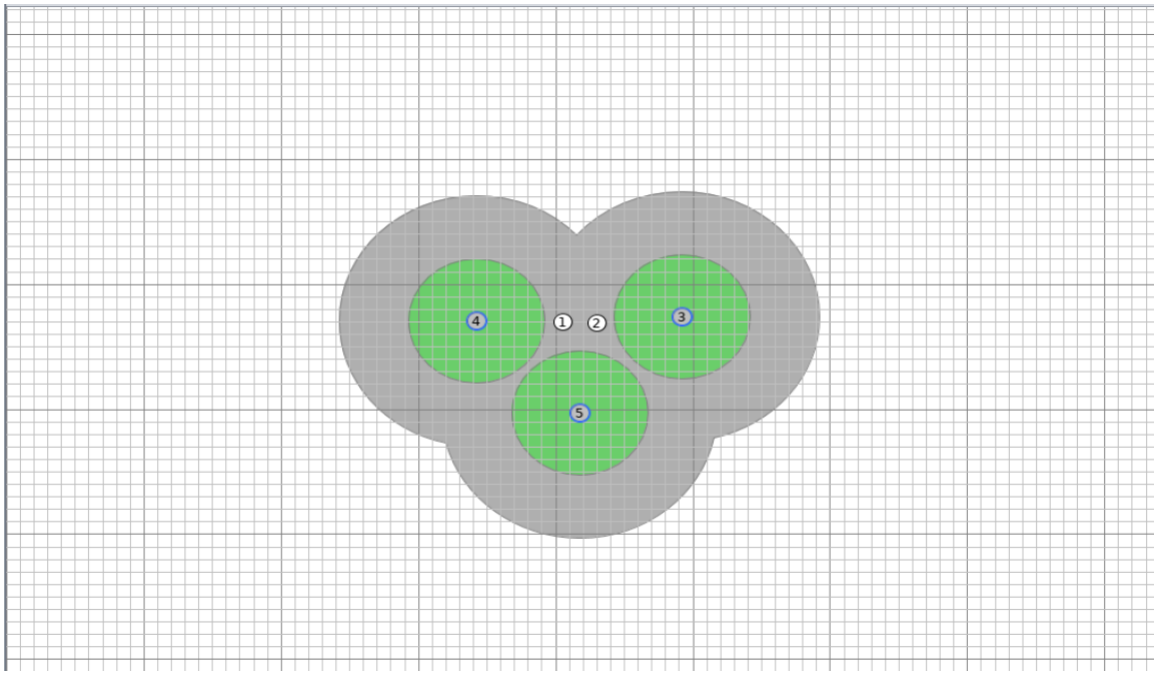

Figure 41: Environment 4

Figure 41 show the simulation environment. Two nodes (1 and 2) placed 20m apart from each other. Disturber nodes (3,4 and 5) is placed within interference range of the two nodes.

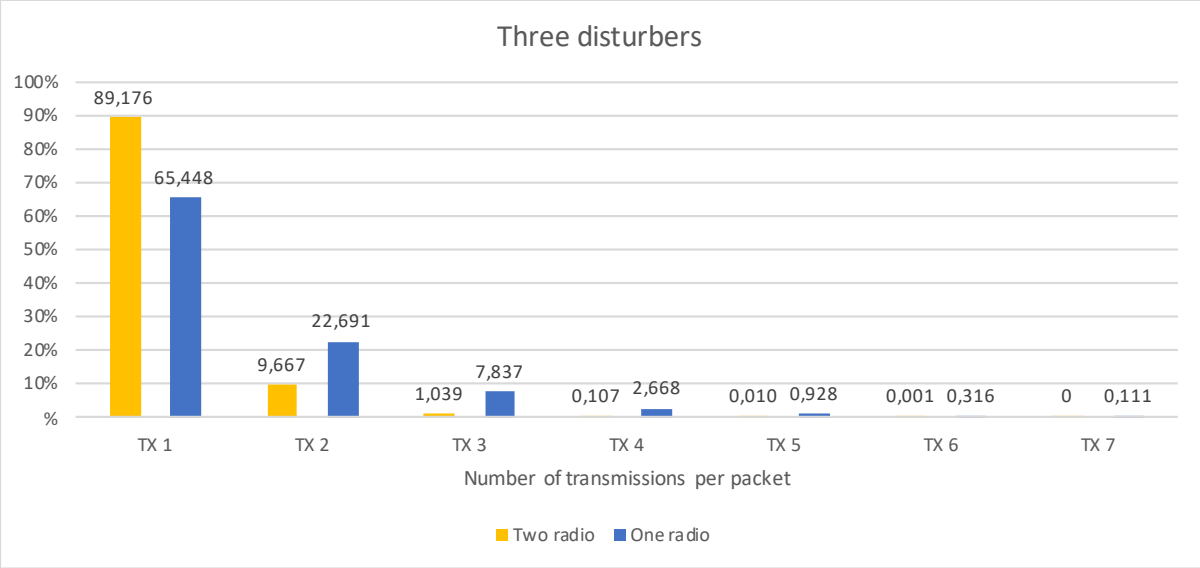

Figure 42: Data three disturbers. Figure shows percentage in Y-direction and the number of retransmissions in the X-direction.

In the latest simulation, we introduced three random disturbers, and the number of retransmissions is increasing even more (shown in Figure 42). When comparing the node with one radio and two-radio, we see that the number of packets sent without retransmissions decreases by 32.53% against two radios 8.51% compared to the perfect environment. Worst case with one radio is above seven transmissions (hence packet drop), and five using two radios.

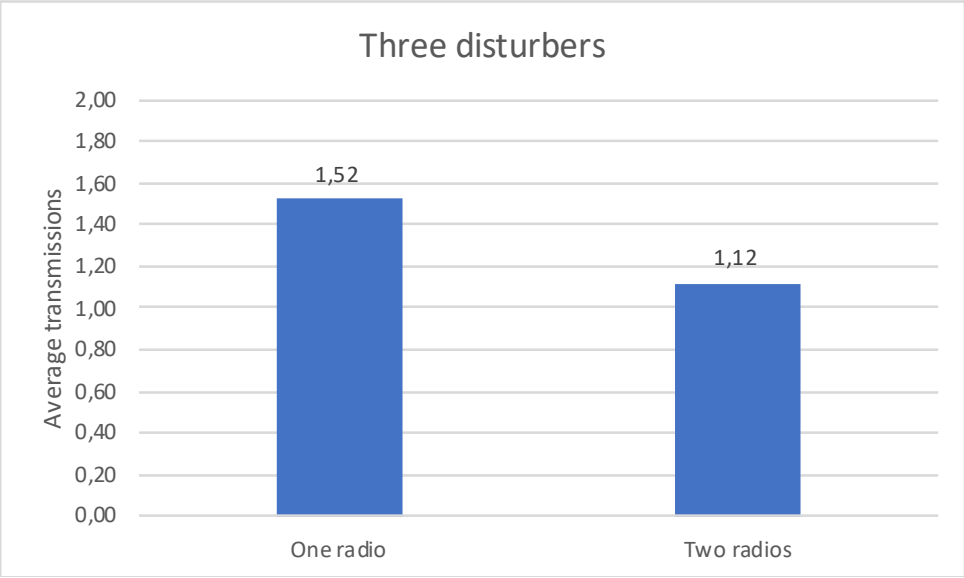

Figure 43: Average transmissions three disturber nodes. Figure shows average transmissions per package in Y-direction.

Figure 43 shows that nodes with one radio needs 1.52 transmission per packet on average, while two-radio nodes need 1,12 transmissions per packet on average. Compared with

environment without interference, transmission per packet has increased by 49.02% with one radio and 9.80% using two radios.

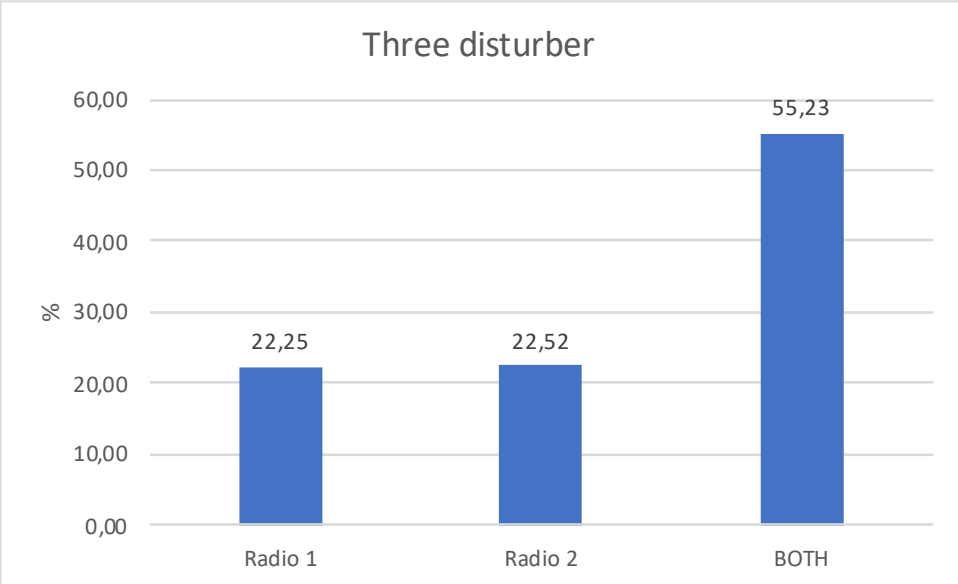

Figure 44 : Three disturbers radio distribution. Figure shows percentage in Y-direction.

Figure 44 shows that not all packets are received on both radios. 44.77% of the packets were only received on one of the radios.

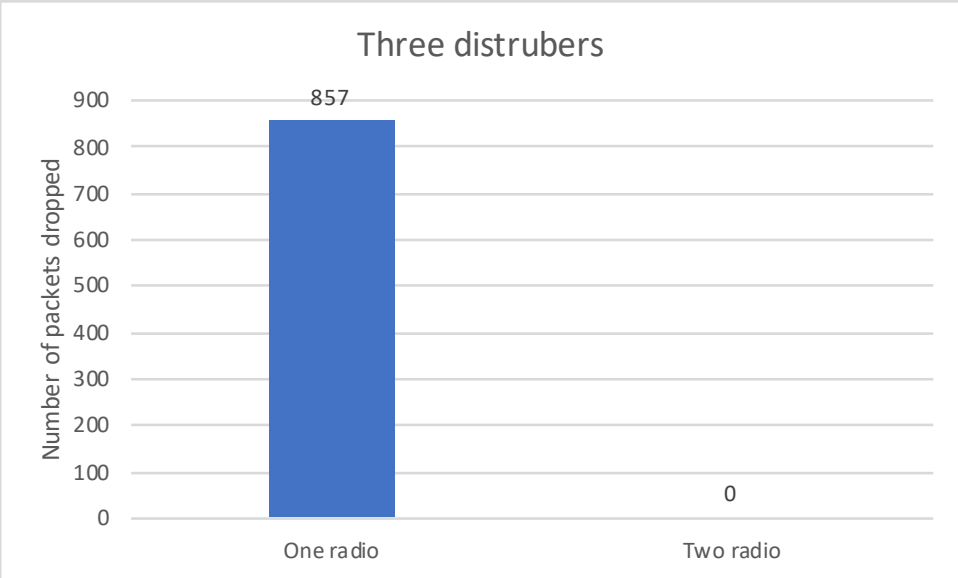

Figure 45 : Package dropped one radio

Figure 45 shows that 857 (or ~8.6%) packets is being dropped with one radio, and 0 when using two radios. Considering the strict requirements of reliability, packaging losses in industrial closed loops can result in degradation of the control system performance, and result in economic loss, or even human safety (e.g., gas pipe explosion).

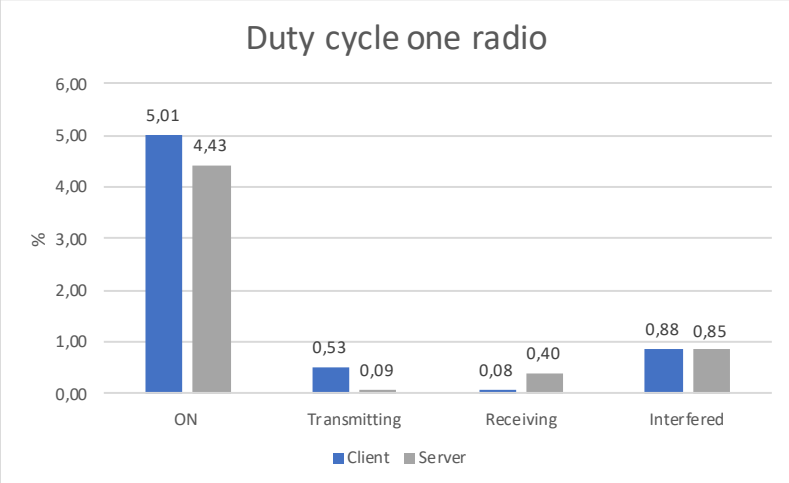

Figure 46: Duty cycle one radio three disturbers. Figure shows percentage in Y-direction and radio modes in X-direction

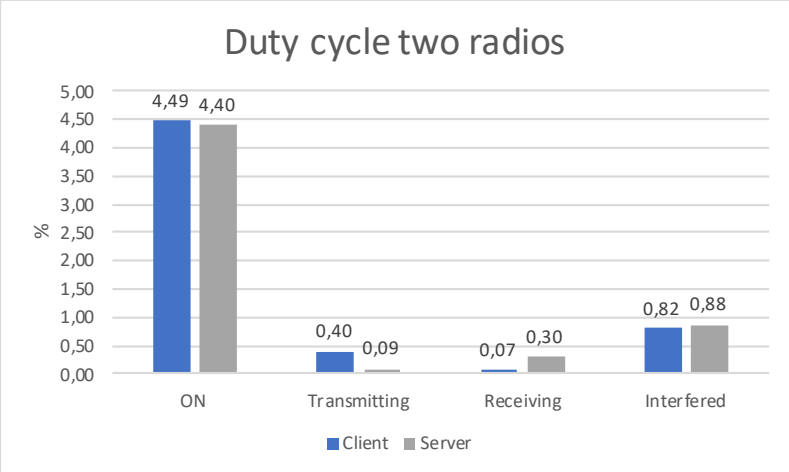

Figure 47: Duty cycle two radios three disturbers. Figure shows percentage in Y-direction and radio modes in X-direction

Duty cycle shows that the node with one radio is ON 5.01%, and with two radios 4.49% (figure 46 and 47). This is because of all retransmissions node with one radio must do as we can see the Transmitting column (0.53% vs. 0.40%) and Receiving (0.40% vs. 0.30%).

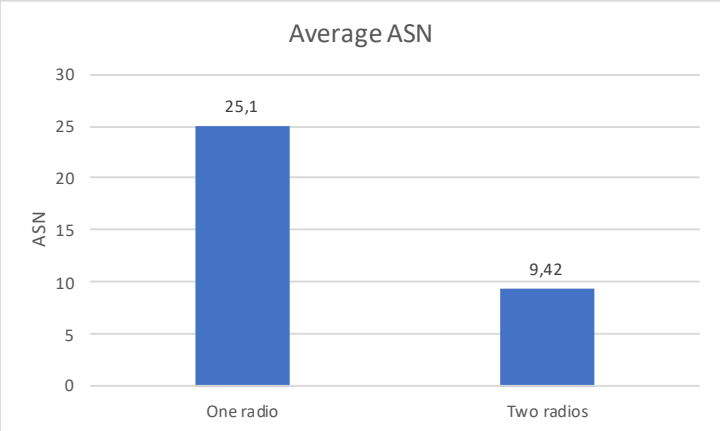

Figure 48: One radio average ASN

Due to retransmissions, we get an increase in average ASN (shown in figure 48). Average ASN when using one radio have increased by 223.45% and with two radios 23.30% compared to no interference simulation.

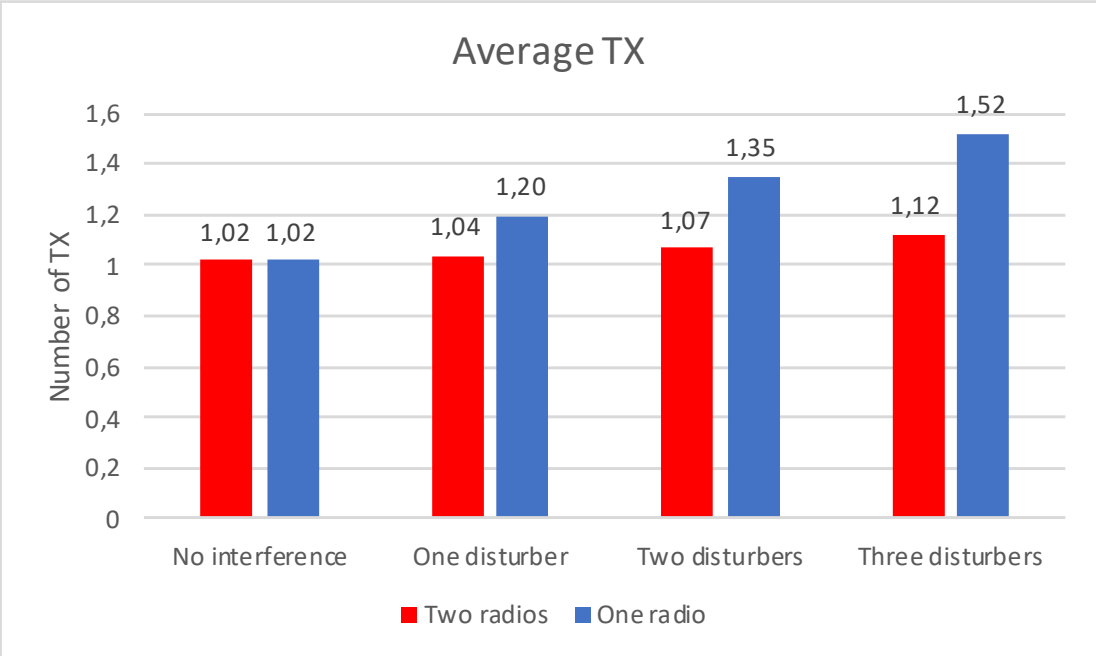

Figure 49: Average TX

Figure 49 shows that with no interference we need the same number of transmissions on average. But, when adding interference to the environment, we see that with three disturber nodes, nodes with one radio need 1.5 transmission on average per packet, compared with two radios 1.12 transmission on average.

5.3 Joining time

In this simulation one node is set as a coordinator. The other node (Joining node) is trying to join by listening for Enhanced beacon (EB). When the node is "synchronized", timestamp is collected. Server node is set as coordinator and starts sending out EB for synchronization. It sends out EB every 1 second. This simulation is repeated 1000 times. I created a java script that saves timestamp printed from joining node, shown in Appendix C.

In this test we will look at the effect of two radios at convergence time. The hypothesis is that when joining node listen on two different frequencies while coordinator node sends out Enhanced beacons (EB) on two different frequencies we will see a significantly reduce in convergence time.

5.3.1 Joining node

MAC	TSCH
TSCH Schedule	6TiSCH Minimal

Table 4: Joining node

5.3.2 Coordinator node

MAC	TSCH
EB sending period	1 Seconds
Slotframe length	101
TSCH Schedule	6TiSCH Minimal

Table 5: Coordinator node

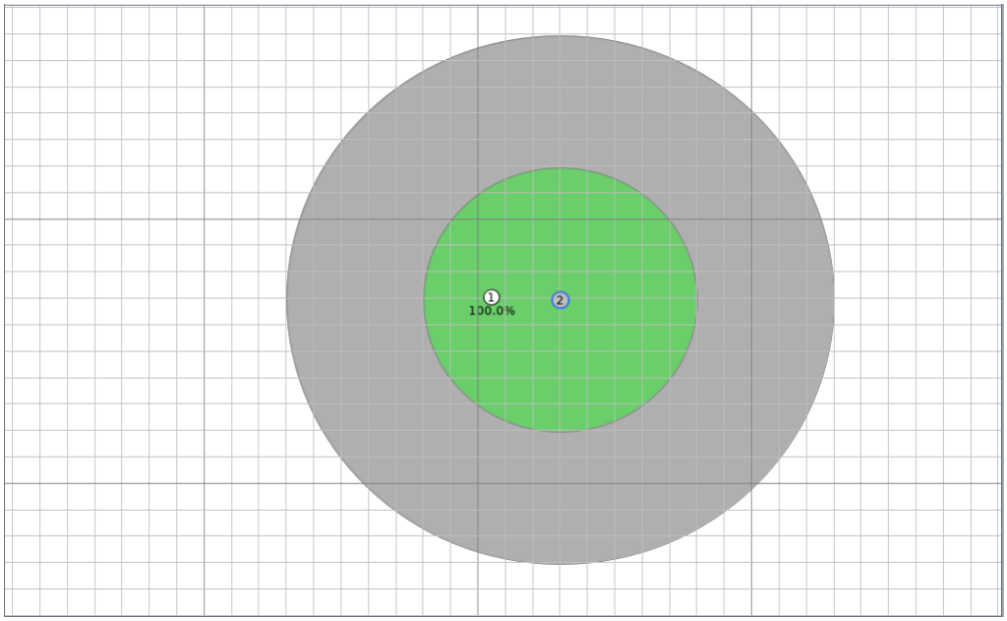

Figure 50 : Joining environment

Figure 50 shows the joining environment. Node 1 represents joining node and is located 20m away from node 2 representing the coordinator.

5.3.3 **Results and analysis**

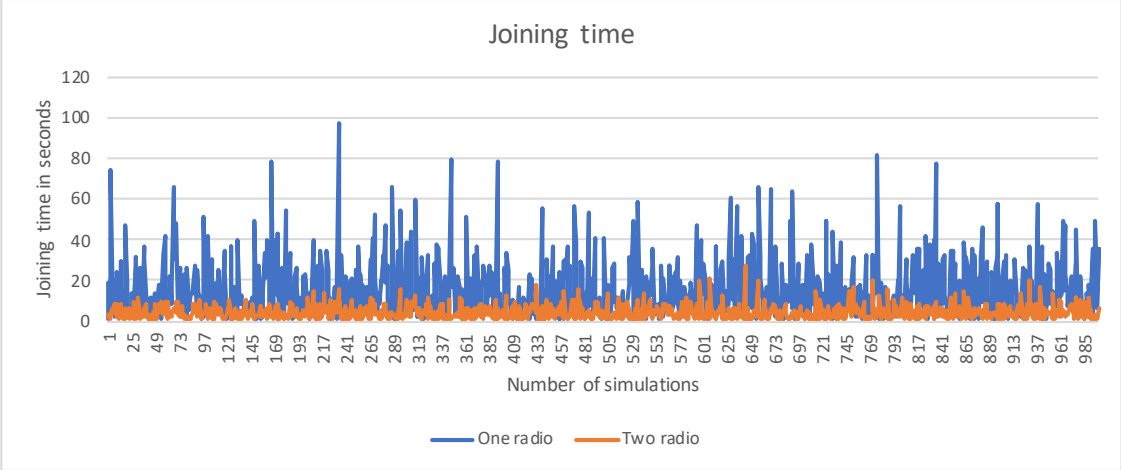

Figure 51 : Joining data. Figure shows number of simulations in X-direction and joining time in second in Y-direction.

	One radio	Two radios
Average time	14,917s (~15s)	4,121s(~4s)
Worst case	97s	27s

Table 6: Result joining simulation

Figure 51 shows, as expected, that two radios significantly reduce convergence time.
With two radios, we get an average connection time of 4.121s (~ 4s), and 14.917s (~ 15s) using one radio. This is a reduction of average connection time of 72.37%.
We also see that two-radio nodes have a worst-case of 27s, and worst case using one radio of 97s, this is a worst-case reduction of 73.37%.

To verify the average connection time using one radio, I have looked for other documents that have done similar tests. In [16] they have used Python as simulation tools, the result showed a connection time of ~15s, which is identical to my result.

Reducing joining time reduces power consumption as nodes must be active throughout the search period. However, listening on two radios and sending EB on two radios also increases the energy consumption.

5.4 Two-hop simulation

In the last test we will look at the impact two radios have on networks that require more hop to destination and the impact this has on latency. In the same way as Chapter 5.2, we use ASN to calculate latency and disturber nodes to simulate interference.

5.4.1 UDP Client node

MAC	TSCH
TSCH Schedule	6TiSCH Minimal
Slotframe length	7 (timeslots)
Timeslot length	10ms
Routing protocol	RPL-lite
Number of packets	10 000
UDP package sending period	1sec
Payload size	64 bytes

Table 7: Two-hop Client node

Client node waits for EB for synchronization. After that the client sends out UDP packet to the server every second. The UDP package contains current ASN numbers derived from client node as payload.

5.4.2 UDP Server node

MAC	TSCH
TSCH Schedule	6TiSCH Minimal
Routing protocol	RPL-lite
Slotframe length	7
Timeslot length	10ms

Table 8: Two-hop Server node

Server node is set as coordinator and starts sending out EB for synchronization. Then listens for incoming packages. The server counts the number of packets received and takes out the ASN number from the payload and uses the current ASN number to calculate the difference. I created a java script to save information printed from the server node, shown in Appendix C.

5.4.3 Intermediate node

MAC	TSCH
TSCH Schedule	6TiSCH Minimal
Routing protocol	RPL-lite
Slotframe length	7
Timeslot length	10ms

Table 9: Two-hop Intermediate node

The intermediate node is waiting for EB to be synchronized. After this, he is only part of the routing topology and forwarding packets that are being sent. After the node has become part of the network, it also sends out periodic EB.

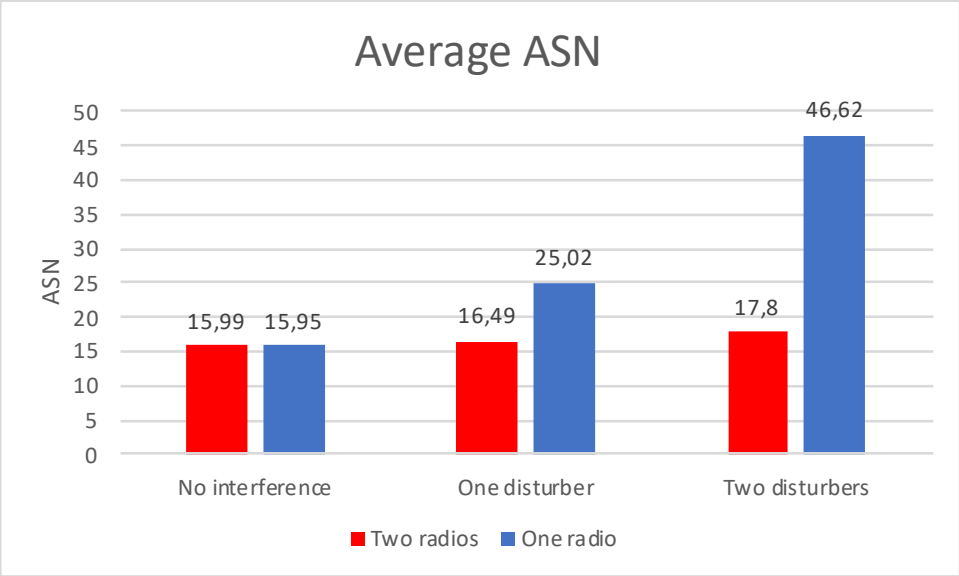

Figure 52: Average delay 2hop. Figure shows Average ASN in Y-direction.

5.4.4 Results and analysis

Figure 52 shows, as expected, that without any interference, we get more or less the same result. However, when adding interference to the environment, nodes with one radio will get a much higher average latency. In particular, we see it when we add two disturber nodes, where nodes with one radio gets an average ASN of 46.62 and two radios get an average of 17.8. We also see that when adding interference, nodes with two radios do not change that much, without interference it has an average ASN of 15.99 and with two disturbers it has an

average of 17.8, this is only an increase of 11.32%. Looking at nodes with one radio, average is increasing 192.29%.

5.5 Discussion

Based on all simulations we see that using two radios may have some advantages. In a perfect environment without interference two radios will not help much, it only has a higher energy consumption, but this is not very representative of an industrial environment. There will be interference that could make link quality poor.

When we add random interference, we see that not only do we get a high number of retransmissions, but also packet loss using one radio.

When we compare Figure 18 and Figure 42, we see that the proportion of packets received without retransmission has gone from 97.69% to 65.45% using one radio, a reduction of 33%, and an average delay of 25.10 ASN (figure 48). Looking at the use of two radios, the proportion of packets received without retransmission has gone from 97.98% to 89.176%, which is a reduction of 9%, and an average delay of 9.42 ASN. The high number of retransmissions do not only increase latency, but you also get a large variation of latency (jitter). When we consider the requirements for Real-time characteristics in industrial systems, where the requirements for latency were less than 10μs and less then 20μs jitter [27], could the high proportion of retransmissions make us not reach these requirements.

Looking at Figures 29, 37 and 45 we see that we have packet loss using one radio, and no packet loss when using two radios. Considering the strict requirements of reliability, packet losses in industrial closed loops can result in degradation of the control system performance, and result in economic loss, or even human safety (e.g., gas pipe explosion). The reason for packets drop is that they exceeded the limit of 7 transmissions. This is also defended by the fact that some industrial network prioritizes new data instead of guaranteeing reception of all packets [8].

Figure 46 and 47 shows a lower duty cycle in percentage when using two radios.
This is because of all retransmissions nodes with one radio must do before the packet is received. A lower duty cycle results in less energy consumption.

Convergence time can also be greatly reduced by using two radios. A joining node must listen for Enhanced Beacon at a given frequency but do not know what frequency to listen to. Instead of listening on one radio after Enhanced Beacon at one random frequency, we listen to beacons at two different frequencies simultaneous. This reduces the average joining time from ~ 15s to ~ 4s, which means a reduction of 72.37%. We also see that we reduce worst cases from 97s to 27s, meaning a reduction of 72.16%. This also leads to a reduction of power consumption as joining nodes must be active throughout the search period.

On the negative side, implementing two radios will increase energy consumption.

Implementing two radios comes with a tradeoff. The node with two radios will have a higher energy consumption as it sends packets over two radios and listen with two radios.
In an environment without interference, implementing two radios will help us meet the requirements of real-time characteristics (low latency and jitter), but only have a higher power consumption that reduces network lifetime.

Looking at the radio duty cycle (figure 46-47) with a random interference sources, we see that with one radio we get overall higher values which means higher energy consumption. But even though the duty cycle is lower in percent with two radios, power consumption will also be higher in these percentages. So, when the node radio is "ON" 5% of the simulation time, both radios will be “ON” 5% of the simulation time, but in these 5 percent’s, it will use twice as much energy.

This solution is not necessarily equally interesting for all industrial applications. However, for applications where ultra-high reliability, low jitter and delay are more important than power consumption, this solution can work. This is particularly important in closed-loops (Table 1), where sensors directly interact with actuators that make a dynamic change like in the WNCS.

6 Conclusion

Implementing two radios on one node or actuator has its advantages, but also disadvantages. Two radios have advantages in terms of reliability and Real-time characteristics, but disadvantages in terms of power consumption, bandwidth used and cost of two radios.

In a perfect environment without any interference, implementing two radios will not have any effect, but in an environment with random noise sources, two radios have a positive effect. By adding interference, we see that the number of retransmissions per packet increases drastically using one radio, this has a negative effect on real-time data that relies on as little latency and variation of latency as possible (jitter). This is particularly important in industrial closed loops where actuators make changes dynamically and the latency and jitter requirements are in milliseconds. We also see that using one radio in a very noisy environment can cause packet loss, but by using two radios, the simulations have shown that we avoid this. Packet loss is also very critical in industrial systems where data can contain alarms and in the worst case this can lead to loss of human life. Convergence time can also be greatly reduced by using two radios. This reduces the average joining time from ~15s to ~4s, which means a reduction of 72.37%. We also see that we reduce worst cases from 97s to 27s, this is a reduction of 72.16%. Lowering joining time will also reduce power consumption as nodes must be active throughout the search period, but at the same time we increase power consumption by listening on two radios

The natural disadvantage of this proposal is the cost of two radios, bandwidth and power consumption.

Using two radios also increases power consumption, as nodes sends and receives on two radios at the same time. One thing the simulations shows is that the duty cycle in percent when using two radios is somewhat lower than using one radio, this is because of all retransmissions nodes with which one radio must do. But even though the duty cycle is lower in percent with two radios, power consumption will also be higher in these percentages.

6.1 Future work

One way to reduce power consumption with two radios is to classify traffic.
For example, if we only send Enhanced beacons (EB) with one radio, we still will get faster connection time when joining nodes search on two different frequencies, but we will reduce the power consumption on an already converted networks.
So, by sending specific critical real-time data on two radios (two different frequencies), we could reduce the power consumption.

In my implementation, there is also a problem if we want to use whitelisting.
A whitelisting selects preferred frequencies and excludes others, but we use frequency offset calculated for radio 1 of the MAC layer to select channel for radio 2.
So, when we select a channel 5 above or below radio 1, this can be one of the excluded channels.

Implementing this on a real node will of course be very interesting. The phenomenon of multi-path fading is something difficult to simulate. The changes I've made in the MAC layer will also work on a real node, but the radio driver I've changed works only for simulation using Cooja.

Bibliography

[1] D. Dujovne, T. Watteyne, X. Vilajosana and P. Thubert, "6TiSCH: deterministic IP-enabled industrial internet (of things)," in IEEE Communications Magazine, vol. 52, no. 12, pp. 36-41, December 2014.
doi: 10.1109/MCOM.2014.6979984

[2] T. Watteyne, C. Adjih and X. Vilajosana, "Lessons learned from large-scale dense IEEE802.15.4 connectivity traces," 2015 IEEE International Conference on Automation Science and Engineering (CASE), Gothenburg, 2015, pp. 145-150. doi: 10.1109/CoASE.2015.7294053

[3] T. Watteyne, A. Mehta, and K. Pister, "Reliability through frequency diversity: Why channel hopping makes sense," in Proceedings of the 6th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks, ser. PE-WASUN '09. New York, NY, USA: ACM, 2009, pp. 116–123. [Online]. Available: http://doi.acm.org/10.1145/1641876.1641898

[4] H. A. Salam and B. M. Khan, "IWSN - Standards, Challenges and Future," in IEEE Potentials, vol. 35, no. 2, pp. 9-16, March-April 2016.doi: 10.1109/MPOT.2015.2422931

[5] L. Doherty, W. Lindsay and J. Simon, "Channel-Specific Wireless Sensor Network Path Data," 2007 16th International Conference on Computer Communications and Networks, Honolulu, HI, 2007, pp. 89-94.doi: 10.1109/ICCCN.2007.4317802

[6] Dimitrios. Serpanos; Marilyn Wolf , "Internet-of-Things (IoT) Systems: Architectures, Algorithms, Methodologies", 2018 doi: 10.1007/978-3-319-69715-4

[7] P. Park, S. Coleri Ergen, C. Fischione, C. Lu and K. H. Johansson, "Wireless Network Design for Control Systems: A Survey," in IEEE Communications Surveys & Tutorials, vol. 20, no. 2, pp. 978-1013, Secondquarter 2018. doi: 10.1109/COMST.2017.2780114

[8] J. Åkerberg, M. Gidlund and M. Björkman, "Future research challenges in wireless sensor and actuator networks targeting industrial automation," 2011 9th IEEE International Conference on Industrial Informatics, Caparica, Lisbon, 2011, pp. 410-415. doi: 10.1109/INDIN.2011.6034912

[9] Jun Zheng; Abbas Jamalipour, "Introduction to Wireless Sensor Networks," in Wireless Sensor Networks:A Networking Perspective , 1, Wiley-IEEE Press, 2009, pp.500-
doi: 10.1002/9780470443521.ch1

[10] G. Zhao, "Wireless sensor networks for industrial process monitoring and control: A survey," Network Protocols and Algorithms, vol. 3, no. 1, pp. 46–63, 2011. [ONLINE] Available: https://pdfs.semanticscholar.org/9293/78d9cf6c289f65cfae760b5eb36a55c9373b.pdf

[11] Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH [Online] Available: http://www.simonduquennoy.net/papers/duquennoy15orchestra.pdf

[12] Madhav Patil Department of Electrical Engineering, Kishan Gutta Department of Electrical Engineering, “Wireless Sensors in Industrial Instrumentation A Survey”. [ONLINE]Available: http://www.asee.org/documents/zones/zone1/2014/Student/PDFs/228.pdf

[13] V. C. Gungor and G. P. Hancke, "Industrial Wireless Sensor Networks: Challenges, Design Principles, and Technical Approaches," in IEEE Transactions on Industrial Electronics, vol. 56, no. 10, pp. 4258-4265, Oct. 2009.doi: 10.1109/TIE.2009.2015754

[14] M. Luvisotto, Z. Pang and D. Dzung, "Ultra High Performance Wireless Control for Critical Applications: Challenges and Directions," in *IEEE Transactions on Industrial Informatics*, vol. 13, no. 3, pp. 1448-1459, June 2017. doi: 10.1109/TII.2016.2617459

[15] “Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement” RFC7554 [Online].
Available: https://tools.ietf.org/html/rfc7554

[16] Apostolos Karalis, Dimitrios Zorbas, and Christos Douligeris, “ATP A Fast Joining Technique for IEEE802154 TSCH Networks” [Online].
Available: https://www.researchgate.net/profile/Dimitrios_Zorbas/publication/325286100_ATP_A_Fast_Joining_Technique_for_IEEE802154-TSCH_Networks/links/5b03f7c9aca2720ba0996561/ATP-A-Fast-Joining-Technique-for-IEEE802154-TSCH-Networks.pdf

[17] Duy, Thang Phan ; Dinh, Thanh ; Kim, Younghan, “A rapid joining scheme based on fuzzy logic for highly dynamic IEEE 802.15.4e time-slotted channel hopping networks” International Journal of Distributed Sensor Networks, 2016, Vol.12(8)

[18] Cooja documentation [Online]. Available: https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja

[19] Tsung-Han Lee,Lin-Huang Chang, Yan-Wei Liu, Jiun-Jian Liaw, Hung-Chi Chu, “Priority-based scheduling using best channel in 6TiSCH networks” [ONLINE].
Available: https://doi.org/10.1007/s10586-017-1185-9

[20] 6TiSCH Operation Sublayer (6top) draft
[Online]. Available: https://tools.ietf.org/html/draft-ietf-6tisch-6top-interface-04

[21] More about Contiki-NG
[Online]. Available: https://github.com/contiki-ng/contiki-ng/wiki/More-about-Contiki-NG

[22] J. Farkas, B. Varga, X. Vilajosana, E. Grossman, C. Gunther, P. Thubert, P. Wetterwald, J. Raymond, J. Korhonen, Y. Kaneko, S. Das, Y. Zha, F.-J. Goetz, J. Schmitt, T. Mahmoodi, S. Spirou, and P. Vizarreta, “Deterministic Networking Use Cases,” Internet Engineering Task Force, Internet-Draft draft-ietf-detnet-use-cases- 11, Oct. 2016, work in Progress.
[Online]. Available: https://tools.ietf.org/html/draft-ietf-detnet-use-cases-11

[23] P. Zand, S. Chatterjea, K. Das, and P. Havinga, “Wireless industrial monitoring and control networks: The journey so far and the road ahead,” Journal of Sensor and Actuator Networks, vol. 1, no. 3, p. 123âA Ş¸152, Aug 2012. doi: 10.3390/jsan1020123

[24] “IEEE Standard for Low-Rate Wireless Networks," in IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011), vol., no., pp.1-709, April 22 2016 doi: 10.1109/IEEESTD.2016.7460875

[25] Office of Energy Efficiency and Renewable Energy, “Wireless success story - industrial technologies program (itp),” U.S. Department of Energy, Tech. Rep., 2010.

[26] J. Werb, “The technology behind isa100.11a user driven design,” ISA100 Wireless Compliance Institute, Tech. Rep., 2010. [Online]. Available: http://isa100wci.org/Documents/PDF/Yokohama- Meeting/ AM1_26- Aug-2010_Jay_Werb_WCI- Tech- Overview

[27] M. Ehrlich, L. Wisniewski, and J. Jasperneite, “State of the art and future applications of industrial wireless sensor networks,” Nov 2016.

[28] A. A. Kumar S., K. Ovsthus and L. M. Kristensen., "An Industrial Perspective on Wireless Sensor Networks — A Survey of Requirements, Protocols, and Challenges," in IEEE Communications Surveys & Tutorials, vol. 16, no. 3, pp. 1391-1412, Third Quarter 2014. doi: 10.1109/SURV.2014.012114.00058

[29] D. D. Guglielmo, S. Brienza, and G. Anastasi, “IEEE 802.15.4e: A survey”, Computer Communications, vol.88, pp. 1–24, 2016.
[Online]. Available: https://doi.org/10.1016/j.comcom.2016.05.004

[30] M. R. Palattella, N. Accettura, L. A. Grieco, G. Boggia, M. Dohler and T. Engel, "On Optimal Scheduling in Duty-Cycled Industrial IoT Applications Using IEEE802.15.4e TSCH," in IEEE Sensors Journal, vol. 13, no. 10, pp. 3655-3666, Oct. 2013. doi: 10.1109/JSEN.2013.2266417

[31] RFC 6550 “RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks” [Online]. Available: https://tools.ietf.org/html/rfc6550

[32] RFC 4944 “Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks” [Online]. Available: https://tools.ietf.org/html/rfc6282

[33] Industrial Routing Requirements in Low-Power and Lossy Networks [Online] Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc5673.txt.pdf

[34] R. Yu, “Mesh network protocols for the industrial internet of things”, MICROWAVE JOURNAL, vol. 57, no. 12, pp. 38–+, 2014.

[35] “Contiki-NG wiki” [Online] Available: https://github.com/contiki-ng/contiki-ng/wiki/More-about-Contiki-NG

[36] “Contiki-OS wiki” [Online] Available: http://www.contiki-os.org

[37] “Contiki-ng Documentation: TSCH and 6TiSCH” [Online] Available: https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-TSCH-and-6TiSCH

[38] “The Contiki-NG configuration system»
[*Online] Available:* https://github.com/contiki-ng/contiki-ng/wiki/The-Contiki-NG-configuration-system

[39] A. Willig, “Recent and emerging topics in wireless industrial communication,” IEEE Transactions on Industrial Informatics, vol. 4, no. 2, pp. 102–124, 2008.

[40] “An Introduction to Cooja» [Online]. Available: https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja

[41] “Cooja simulation” [Online]. Available: http://anrg.usc.edu/contiki/index.php/Cooja_Simulator

[42] “Cooja Simulator Manual» [*Online]*. *Available:* https://www.researchgate.net/publication/304572240_Cooja_Simulator_Manual

[43] “Dual-radio implantation Contiki-OS”
[*Online]*. *Available:* https://github.com/clovervnd/Dual-radio-simulation

[44] “Oria Find printed and electronic books, journals, articles, book chapters and more. at UiO and other Norwegian academic and research libraries”
[*Online]*. *Available:* https://www.ub.uio.no

[45] “IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011) - IEEE Standard for Low-Rate Wireless Networks”
[Online]. Available: https://standards.ieee.org/findstds/standard/802.15.4-2015.html

7 Appendix A

7.1 ISO stack – WirelessHART vs ISA100.11a

Layer	WirelessHART	ISA100.11a
Application	Application	Upper application layer
		Application Sublayer
Presentation	Not defined	Not defined
Session	Not defined	Not defined
Transport	Transport	Transport
Network	Network Layer Services	Network Layer
	Network Layer	
Data link	Logical Link Layer	Upper Data Link Layer
	MAC Sublayer	MAC Extension
		MAC Sublayer
Physical	Physical (2,4GHz bands, DSSS modulation IEE 802.15.4 - 2006 radio	Physical

Table 10: ISO stack – WirelessHART vs ISA100.11a

7.2 Duty cycle

Power consumption is very imported in a battery powered device. Devices can be places at challenging spots, and charging/changing battery is difficult. Duty cycle model is widely used to analyze IoT devices [9] and is calculated as a percentage of the time the devices is active. The more it can sleep ("turned off"), the lower energy consumption.

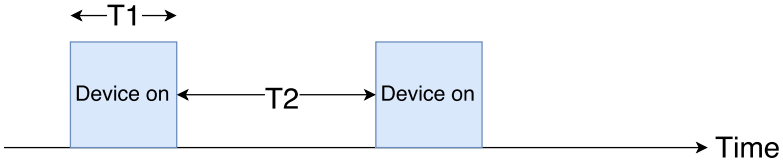

Figure 53 : Duty cycle

$$Duty = \frac{T1}{T2} * 100\%$$

7.3 Retransmission

Industrial networks require high reliability. This is done by using acknowledgement by the recipient, shown in figure 54. If the sender does not receive acknowledgement within a timeframe (ack-timeout), he assumes that the receiver did not receive the packet and transmits the packet again. This will increase delay.

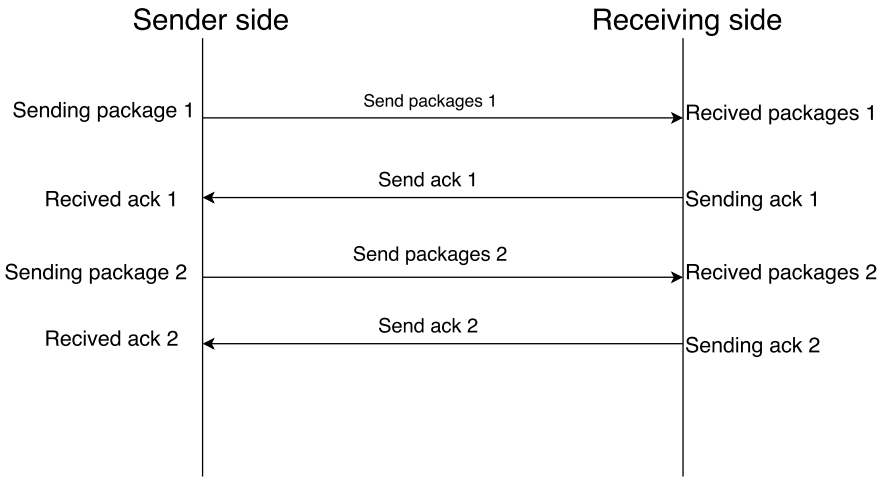

Figure 54 : Acknowledgement

7.4 IEEE 802.15.4

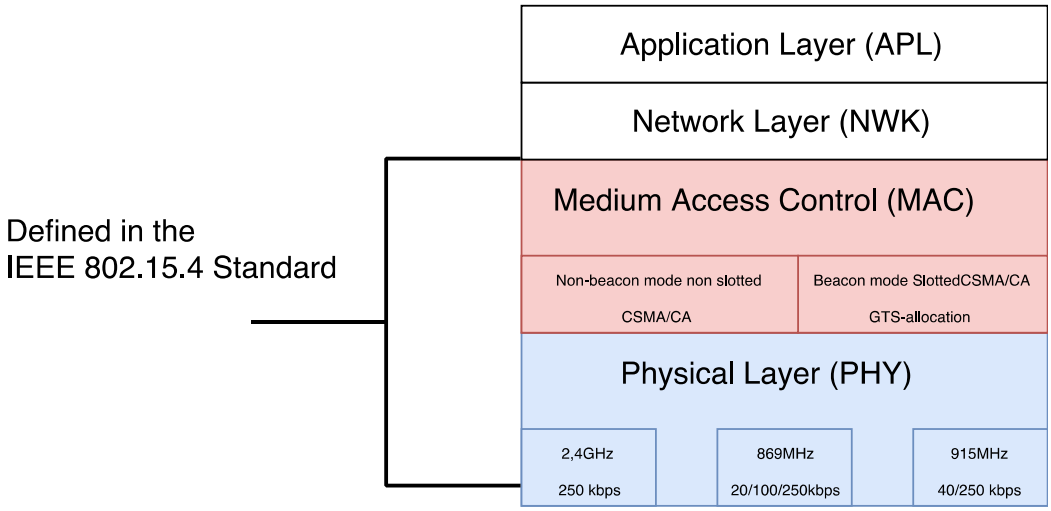

Figure 55: IEEE 802.15.4

Both WirelessHART and ISA have the IEEE 802.15.4 [24] standard as base. The standard was designed for low rate wireless personal area networks (LR-WPANs) and has several advantages such as:

- Good against noise
    - Direct Sequence Spread Spectrum (DSSS)
        - Spreading the message signal by modulating following a bit sequence (radio pulse)
- Good against interference
    - CSMA/CA to access the physical medium.
    - GTS (PAN coordinator is guaranteed timeslot
    - Channel energy scan - PLME-ED request prior to using the channel to know how much energy there is (activity/noise/interferences)
        - Energy, Carrier Sense (CCA), CCA + Energy
- Low consumption protocol (low duty cycles, sleep up to 99%)

The physical layer provides data transmission services such as direct sequence spread spectrum (DSSS) to minimize data loss due to noise, and process RF transceiver, perform channel selection, and signal MGMT functions.
The MAC layers task is to send MAC packets through the physical layer, treat access to the physical layer and network beaconing, check package validation and guarantee timeslot (GTS).

8 Appendix B

Here I will explain the change made in the Cooja simulation tool, and changes made on the MAC layer. I explain only the changes I have made and why they had to be done.

(..) means that there are codes that have been removed as they are either not relevant or I have not made changes there.

The entire code can be retrieved from:
https://github.com/VegarKrogsethagen/Contiki-NG-Dual-radio.git

8.1 Cooja-radio-driver

```
(.....)
```

```
1.
2.  /* COOJA */
3.  char simReceiving = 0;
4.  char simInDataBuffer[COOJA_RADIO_BUFSIZE];
5.  int simInSize = 0;
6.  rtimer_clock_t simLastPacketTimestamp = 0;
7.  char simOutDataBuffer[COOJA_RADIO_BUFSIZE];
8.  int simOutSize = 0;
9.  char simRadioHWOn = 1;
10. int simSignalStrength = -100;
11. int simLastSignalStrength = -100;
12. char simPower = 100;
13. int simRadioChannel = 26;
14. int simLQI = 105;
15. /* COOJA */
16.
17. /*Dummy Radio*/
18. char simReceivingDummy = 0;
19. char simInDataBufferDummy[COOJA_RADIO_BUFSIZE];
20. int simInSizeDummy = 0;
21. rtimer_clock_t simLastPacketTimestampDummy = 0;
22. char simOutDataBufferDummy[COOJA_RADIO_BUFSIZE];
23. int simOutSizeDummy = 0;
24. char simRadioHWOnDummy = 1;
25. int simSignalStrengthDummy = -100;
26. int simLastSignalStrengthDummy = -100;
27. char simPowerDummy = 100;
28. int simRadioChannelDummy = 1;
29. int simLQIDummy = 105;
30. /*Dummy Radio*/
31. static const void *pending_data;
32. //static int seqnr=0;
33. /* If we are in the polling mode, poll_mode is 1; otherwise 0 */
34. static int poll_mode = 0; /* default 0, disabled */
35. static int auto_ack = 0; /* AUTO_ACK is not supported; always 0 */
36. static int addr_filter = 0; /* ADDRESS_FILTER is not supported; always 0 */
37. static int send_on_cca = (COOJA_TRANSMIT_ON_CCA != 0);
38.
39. PROCESS(cooja_radio_process1, "cooja radio process");
40. /*---------------------------------------------------------------------------*/
```

```
                                        (.....)

/*---------------------------------------------------------------------------*/
void
radio_set_channel(int channel)  // Set radio frequency on radio 1
{
  simRadioChannel = channel;

}
void
radio_set_channelDummy(int channel) // Set radio frequency on radio 2
{
  simRadioChannelDummy = channel;

}

/*---------------------------------------------------------------------------*/
void
radio_set_txpower(unsigned char power)  // Sett transmission power on both
radios
{
  /* 1 - 100: Number indicating output power */
  simPower = power;
  simPowerDummy = power;
}
/*---------------------------------------------------------------------------*/
                                        (.....)
/*---------------------------------------------------------------------------*/
static int
radio_on(void)  // Turn both radios on.
{
  simRadioHWOn = 1;
  simRadioHWOnDummy = 1;
  return 1;
}
/*---------------------------------------------------------------------------*/
static int
radio_off(void)  // Turn both radios off.
{
  simRadioHWOn = 0;
  simRadioHWOnDummy = 0;
  return 1;
}
/*---------------------------------------------------------------------------*/
static void
doInterfaceActionsBeforeTick(void)
{
  if(!simRadioHWOn && !simRadioHWOnDummy) {
    simInSizeDummy = 0;
    simInSize = 0;

    return;
 // Check if both radios are on, if not there is no point in
checking for incoming packages
  }

  if(simReceiving && simReceivingDummy)
  {
     simLastSignalStrength = simSignalStrength;
     simLastSignalStrengthDummy = simSignalStrengthDummy;
     return;
```

```
98.  // Check if radios are reciving something right now, if they
     are we do anything other than update them.
99.   }
100.  else if(simReceiving && !simReceivingDummy)
101.   {
102.     simLastSignalStrength = simSignalStrength;
103.     // Check if radio 1 is receiving something and radio 2
     don't
104.   }
105.  else
106.   {
107.     simLastSignalStrengthDummy = simSignalStrengthDummy;
108.    // Checking radio 2.
109.   }
110.  if(simInSize > 0 || simInSizeDummy > 0)
111.  {
112.    // If they have a payload greater then 0 we pull a process
     that reads the data and notify MAC layer.
113.       process_poll(&cooja_radio_process1);
114.  }
115.}
116.                                        (.....)
117.static int
118.radio_read(void *buf, unsigned short bufsize)
119.{
120.  int tmp = simInSize;
121.  int tmp1 = simInSizeDummy;
122.
123.  if(simInSize <0 && simInSizeDummy <0) {
124.    // Checking if the received data is greater than 0, if not
     we return 0.
125.    return 0;
126.  }
127.
128.  if(bufsize < simInSize && bufsize < simInSizeDummy) {
129.    simInSize = 0; /* rx flush */
130.    simInSizeDummy = 0; /* rx flush */
131.    // if the incoming data is over the buffsize limit for both
     radios we return 0.
132.    return 0;
133.  }
134.  else if(bufsize < simInSize) {
135.    simInSize = 0; /* rx flush */
136.   // Checks only radio 1s buffer limit.
137.  }
138.  else if(bufsize < simInSizeDummy) {
139.    simInSizeDummy = 0; /* rx flush */
140.   // Checks only radio 2s buffer limit
141.  }
142.
143.
144.  if(simInDataBuffer[2]==simInDataBufferDummy[2])
145.  {
146.      memcpy(buf, simInDataBuffer, simInSize);
147.      simInSize = 0;
148.          simInSizeDummy = 0;
149.      if(!poll_mode) {
150.        packetbuf_set_attr(PACKETBUF_ATTR_RSSI, simSignalStrength);
151.        packetbuf_set_attr(PACKETBUF_ATTR_LINK_QUALITY, simLQI);
152.      }
```

```
153.   // Comparing Seq-Nr for incoming data. If there are a match
       we choose data from radio 1. Return the data(tmp) and notify
       MAC layer.
154.          return tmp;
155.}
156.  else if(simInSize>0)
157. {
158.     // Checking if radio 1 have received something. Return the
       data(tmp) and notify MAC layer.
159.          memcpy(buf, simInDataBuffer, simInSize);
160.          simInSize = 0;
161.          simInSizeDummy = 0;
162.      if(!poll_mode) {
163.        packetbuf_set_attr(PACKETBUF_ATTR_RSSI, simSignalStrength);
164.        packetbuf_set_attr(PACKETBUF_ATTR_LINK_QUALITY, simLQI);
165.      }
166.
167.          return tmp;
168.}
169. else
170. {
171.     // Checking if radio 2 have received something. Return the
       data(tmp1) and notify MAC layer.
172.      simLastPacketTimestamp = simLastPacketTimestampDummy;
173.      memcpy(buf, simInDataBufferDummy, simInSizeDummy);
174.      simInSize = 0;
175.          simInSizeDummy = 0;
176.      if(!poll_mode) {
177.        packetbuf_set_attr(PACKETBUF_ATTR_RSSI, simSignalStrengthDummy);
178.        packetbuf_set_attr(PACKETBUF_ATTR_LINK_QUALITY, simLQIDummy);
179.      }
180.
181.          return tmp1;
182.}
183.
184.}
185./*---------------------------------------------------------------------------*/
186.                                    (.....)
187./*---------------------------------------------------------------------------*/
188.static int
189.radio_send(const void *payload, unsigned short payload_len)
190.{
191.
192.
193.  int radiostate = simRadioHWOn;
194.  simRadioHWOnDummy = simRadioHWOn;
195.  /* Simulate turnaround time of 2ms for packets, 1ms for acks*/
196.#if COOJA_SIMULATE_TURNAROUND
197.  simProcessRunValue = 1;
198.  cooja_mt_yield();
199.  if(payload_len > 3) {
200.    simProcessRunValue = 1;
201.    cooja_mt_yield();
202.  }
203.#endif /* COOJA_SIMULATE_TURNAROUND */
204.
205.  if(!simRadioHWOn) {
206.    // Turn radio 1 on.
207.
208.    /* Turn on radio temporarily */
209.    simRadioHWOn = 1;
210.  }
211.  if(!simRadioHWOnDummy) {
```

```
212.    // Turn radio 2 on.
213.
214.    /* Turn on radio temporarily */
215.    simRadioHWOnDummy = 1;
216.  }
217.
218.  if(payload_len > COOJA_RADIO_BUFSIZE) {
219.    // The payload is grater then the limited bufsize, returns
    error.
220.    // The payload is greater than the limited bufsize.
221.    return RADIO_TX_ERR;
222.  }
223.  if(payload_len == 0) {
224. // No payload, returning error.
225.    // empty package.
226.    return RADIO_TX_ERR;
227.  }
228.  if(simOutSize > 0 && simOutSizeDummy > 0) {
229.    // Already something in buffer, returning error.
230.    return RADIO_TX_ERR;
231.  }
232.
233.  /* Transmit on CCA */
234.#if COOJA_TRANSMIT_ON_CCA
235.  if(send_on_cca && !channel_clear()) {
236.    return RADIO_TX_COLLISION;
237.  }
238.#endif /* COOJA_TRANSMIT_ON_CCA */
239.       // Copying data to buffer.
240.       memcpy(simOutDataBuffer, payload, payload_len);
241.       memcpy(simOutDataBufferDummy, payload, payload_len);
242.       simOutSize = payload_len;
243.       simOutSizeDummy = payload_len;
244.     while(simOutSize > 0 && simOutSizeDummy > 0)
245.     {
246.      // Sending the data
247.       // Sending the data
248.       cooja_mt_yield();
249.
250.     }
251.        // Turn radios off.
252.        simRadioHWOnDummy = radiostate;
253.        simRadioHWOn = radiostate;
254.        return RADIO_TX_OK;
255.
256.}
257./*---------------------------------------------------------------------------*/
258.static int
259.prepare_packet(const void *data, unsigned short len)
260.{
261.  // Called by MAC layer, saving pending package before
    transmitting it.
262.  pending_data = data;
263.  return 0;
264.}
265./*---------------------------------------------------------------------------*/
266.static int
267.transmit_packet(unsigned short len)
268.{
269.  // Called by MAC layer, sending the pending package.
270.  int ret = RADIO_TX_ERR;
271.  if(pending_data != NULL) {
```

```
272.    ret = radio_send(pending_data, len);
273.  }
274.  return ret;
275.}
276./*---------------------------------------------------------------------------*/
277.static int
278.receiving_packet(void)
279.{
280.  // Called by MAC layer, checking if radios are receiving
   something right now.
281.
282. if(simReceiving == 1 || simReceivingDummy == 1 ){ return 1; }
283.else return 0;
284.
285.}
286./*---------------------------------------------------------------------------*/
287.static int
288.pending_packet(void)
289.{  // Called by MAC layer, checking if radios have a pending
   package.
290.
291. if ((!simReceiving && simInSize > 0) || (!simReceivingDummy && simInSizeDummy > 0))
292.  { return 1;}
293.  else return 0;
294. // return !simReceiving && simInSize > 0;
295.}
```

```
296.
297./*---------------------------------------------------------------------------*/
298.PROCESS_THREAD(cooja_radio_process1, ev, data)
299.{
300.  int len;
301.
302.  PROCESS_BEGIN();
303.
304.  while(1)
305.    {
306.    // Waiting for notification of incoming data. This is
   pulled by on doInterfaceActionBeforeTicks.
307.    PROCESS_YIELD_UNTIL(ev == PROCESS_EVENT_POLL);
308.    if(poll_mode) {
309.      continue;
310.    }
311.
312.    packetbuf_clear();
313.    // Reads the data and notify MAC layer.
314.    len = radio_read(packetbuf_dataptr(), PACKETBUF_SIZE);
315.    if(len > 0) {
316.    packetbuf_set_datalen(len);
317.    NETSTACK_MAC.input();  }
318.  }
319.
320.  PROCESS_END();
321.}
322.
323.
324./*---------------------------------------------------------------------------*/
325.                                    (……)
326./*---------------------------------------------------------------------------*/
327.static radio_result_t
328.set_value(radio_param_t param, radio_value_t value)
329.{
330.  switch(param) {
331.  case RADIO_PARAM_RX_MODE:
332.    if(value & ~(RADIO_RX_MODE_ADDRESS_FILTER |
333.        RADIO_RX_MODE_AUTOACK | RADIO_RX_MODE_POLL_MODE)) {
334.      return RADIO_RESULT_INVALID_VALUE;
335.    }
336.
337.    /* Only disabling is acceptable for RADIO_RX_MODE_ADDRESS_FILTER */
338.    if ((value & RADIO_RX_MODE_ADDRESS_FILTER) != 0) {
339.      return RADIO_RESULT_NOT_SUPPORTED;
340.    }
341.    set_frame_filtering((value & RADIO_RX_MODE_ADDRESS_FILTER) != 0);
342.
343.    /* Only disabling is acceptable for RADIO_RX_MODE_AUTOACK */
344.    if ((value & RADIO_RX_MODE_ADDRESS_FILTER) != 0) {
345.      return RADIO_RESULT_NOT_SUPPORTED;
346.    }
347.    set_auto_ack((value & RADIO_RX_MODE_AUTOACK) != 0);
348.
349.    set_poll_mode((value & RADIO_RX_MODE_POLL_MODE) != 0);
350.    return RADIO_RESULT_OK;
351.  case RADIO_PARAM_TX_MODE:
352.    if(value & ~(RADIO_TX_MODE_SEND_ON_CCA)) {
353.      return RADIO_RESULT_INVALID_VALUE;
354.    }
355.    set_send_on_cca((value & RADIO_TX_MODE_SEND_ON_CCA) != 0);
356.    return RADIO_RESULT_OK;
357.  case RADIO_PARAM_CHANNEL:
358.    if(value < 11 || value > 26) {
```

```
359.      return RADIO_RESULT_INVALID_VALUE;
360.    }
361.    radio_set_channel(value);
362.    return RADIO_RESULT_OK;
363. case RADIO_PARAM_CHANNELDummy:  // Channel set by MAC layer.
364.    if(value < 11 || value > 26) {
365.      return RADIO_RESULT_INVALID_VALUE;
366.    }
367.    radio_set_channelDummy(value);
368.    return RADIO_RESULT_OK;
369.  default:
370.    return RADIO_RESULT_NOT_SUPPORTED;
371.  }
372.}
373./*---------------------------------------------------------------------------*/
374.static radio_result_t
375.get_object(radio_param_t param, void *dest, size_t size)
376.{
377.  if(param == RADIO_PARAM_LAST_PACKET_TIMESTAMPContikiRadio)
378.   {
379.    if(size != sizeof(rtimer_clock_t) || !dest) {
380.      return RADIO_RESULT_INVALID_VALUE;
381.    }
382.
383.    *(rtimer_clock_t *)dest = (rtimer_clock_t)simLastPacketTimestamp;
384.    return RADIO_RESULT_OK;
385.  }
386.  else if(param == RADIO_PARAM_LAST_PACKET_TIMESTAMPDummy) {
387.    if(size != sizeof(rtimer_clock_t) || !dest) {
388.      return RADIO_RESULT_INVALID_VALUE;
389.    }
390.
391.    *(rtimer_clock_t *)dest = (rtimer_clock_t)simLastPacketTimestampDummy;
392.    return RADIO_RESULT_OK;
393.  }
394.  return RADIO_RESULT_NOT_SUPPORTED;
395.}
396./*---------------------------------------------------------------------------*/
397.static radio_result_t
398.set_object(radio_param_t param, const void *src, size_t size)
399.{
400.  return RADIO_RESULT_NOT_SUPPORTED;
401.}
402./*---------------------------------------------------------------------------*/
403.const struct radio_driver cooja_radio_driver =
404.{
405.    init,
406.    prepare_packet,
407.    transmit_packet,
408.    radio_send,
409.    radio_read,
410.    channel_clear,
411.    receiving_packet,
412.    pending_packet,
413.    radio_on,
414.    radio_off,
415.    get_value,
416.    set_value,
417.    get_object,
418.    set_object
419.};
420./*---------------------------------------------------------------------------*/
421.SIM_INTERFACE(radio_interface,doInterfaceActionsBeforeTick,doInterfaceActionsAfterTick);
422.SIM_INTERFACE(dummy_interface,doInterfaceActionsBeforeTick,doInterfaceActionsAfterTick);
```

8.2 Cooja-config

```
/*-----------------------------------------------------------------------*/
                                    (.....)
/*-----------------------------------------------------------------------*/

org.contikios.cooja.contikimote.interfaces.ContikiRadio.RADIO_TRANSMISSION_RATE_kbps = 250
org.contikios.cooja.contikimote.interfaces.DummyInterface.RADIO_TRANSMISSION_RATE_kbps = 250
org.contikios.cooja.contikimote.ContikiMoteType.MOTE_INTERFACES = org.contikios.cooja.interfaces.Position org.contikios.cooja.interfaces.Battery org.contikios.cooja.contikimote.interfaces.ContikiVib org.contikios.cooja.contikimote.interfaces.ContikiMoteID org.contikios.cooja.contikimote.interfaces.ContikiRS232 org.contikios.cooja.contikimote.interfaces.ContikiBeeper org.contikios.cooja.interfaces.RimeAddress org.contikios.cooja.contikimote.interfaces.ContikiIPAddress org.contikios.cooja.contikimote.interfaces.ContikiRadio org.contikios.cooja.contikimote.interfaces.DummyInterface org.contikios.cooja.contikimote.interfaces.ContikiButton org.contikios.cooja.contikimote.interfaces.ContikiPIR org.contikios.cooja.contikimote.interfaces.ContikiClock org.contikios.cooja.contikimote.interfaces.ContikiLED org.contikios.cooja.contikimote.interfaces.ContikiCFS org.contikios.cooja.contikimote.interfaces.ContikiEEPROM org.contikios.cooja.interfaces.Mote2MoteRelations org.contikios.cooja.interfaces.MoteAttributes
/*-----------------------------------------------------------------------*/
                                    (.....)
/*-----------------------------------------------------------------------*/
```

8.3 DummyRadioInterface

```
1.  /*
2.   * Copyright (c) 2008, Swedish Institute of Computer Science.
3.   * All rights reserved.
4.   *
5.   * Redistribution and use in source and binary forms, with or without
6.   * modification, are permitted provided that the following conditions
7.   * are met:
8.   * 1. Redistributions of source code must retain the above copyright
9.   *    notice, this list of conditions and the following disclaimer.
10.  * 2. Redistributions in binary form must reproduce the above copyright
11.  *    notice, this list of conditions and the following disclaimer in the
12.  *    documentation and/or other materials provided with the distribution.
13.  * 3. Neither the name of the Institute nor the names of its contributors
14.  *    may be used to endorse or promote products derived from this software
15.  *    without specific prior written permission.
16.  *
17.  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
18.  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
19.  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
20.  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
21.  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
22.  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
23.  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
24.  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
25.  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
26.  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
27.  * SUCH DAMAGE.
28.  *
29.  */
30. package org.contikios.cooja.contikimote.interfaces;
31.
32.
33. import java.util.*;
34. import javax.swing.*;
35. import org.apache.log4j.Logger;
36. import org.jdom.Element;
37.
38.
39. import java.awt.BorderLayout;
40. import java.awt.event.ActionEvent;
41. import java.awt.event.ActionListener;
42. import java.util.Observable;
43. import java.util.Observer;
44. import java.lang.*;
45. import javax.swing.Box;
46. import javax.swing.JButton;
47. import javax.swing.JLabel;
48. import javax.swing.JPanel;
49.
50. import org.contikios.cooja.COOJARadioPacket;
51. import org.contikios.cooja.Mote;
52. import org.contikios.cooja.RadioPacket;
53. import org.contikios.cooja.Simulation;
54. import org.contikios.cooja.contikimote.ContikiMote;
55. import org.contikios.cooja.interfaces.PolledAfterActiveTicks;
56. import org.contikios.cooja.interfaces.Position;
57. import org.contikios.cooja.interfaces.Radio;
58. import org.contikios.cooja.mote.memory.VarMemory;
59. import org.contikios.cooja.radiomediums.UDGM;
60. import org.contikios.cooja.util.CCITT_CRC;
```

```
61.
62.
63.
64.
65. import org.contikios.cooja.*;
66. import org.contikios.cooja.contikimote.ContikiMoteInterface;
67. import org.contikios.cooja.interfaces.PolledAfterAllTicks;
68. import org.contikios.cooja.interfaces.PolledBeforeAllTicks;
69. import org.contikios.cooja.mote.memory.SectionMoteMemory;
70.
71. @ClassDescription("Dummy Interface")
72. public class DummyInterface extends Radio implements ContikiMoteInterface, PolledBeforeAllTick
    s, PolledAfterAllTicks {
73.   private static Logger logger = Logger.getLogger(DummyInterface.class);
74.
75.   private ContikiMote mote;
76.   private VarMemory myMoteMemory;
77.
78.
79. /**
80.    * Transmission bitrate (kbps).
81.    */
82.   private double RADIO_TRANSMISSION_RATE_kbps;
83.
84.   private RadioPacket packetToMote = null;
85.
86.   private RadioPacket packetFromMote = null;
87.
88.   private boolean radioOn = true;
89.
90.   private boolean isTransmitting = false;
91.
92.   private boolean isInterfered = false;
93.
94.   private long transmissionEndTime = -1;
95.
96.   private RadioEvent lastEvent = RadioEvent.UNKNOWN;
97.
98.   private long lastEventTime = 0;
99.
100.  private int oldOutputPowerIndicator = -1;
101.
102.  private int oldRadioChannel = -1;
103.
104.
105.
106.
107.
108.
109.  public DummyInterface(Mote mote) {
110.RADIO_TRANSMISSION_RATE_kbps = mote.getType().getConfig().getDoubleValue(
111.        DummyInterface.class, "RADIO_TRANSMISSION_RATE_kbps");
112.
113.
114.    this.mote = (ContikiMote) mote;
115.    this.myMoteMemory = new VarMemory(mote.getMemory());
116.
117.    radioOn = myMoteMemory.getByteValueOf("simRadioHWOnDummy") == 1;
118.  }
119.
```

```
120.  public static String[] getCoreInterfaceDependencies() {
121.
```

```
122.     return new String[] { "dummy_interface" };
123.   }
124.
125.   public void doActionsBeforeTick() {
126.     logger.debug("Java-part of dummy interface acts BEFORE mote tick:");
127.   }
128.
129.   public void doActionsAfterTick() {
130.
131.     long now = mote.getSimulation().getSimulationTime();
132.
133.     /* Check if radio hardware status changed */
134.     if (radioOn != (myMoteMemory.getByteValueOf("simRadioHWOnDummy") == 1)) {
135.       radioOn = !radioOn;
136.
137.       if (!radioOn) {
138.         myMoteMemory.setByteValueOf("simReceivingDummy", (byte) 0);
139.         myMoteMemory.setIntValueOf("simInSizeDummy", 0);
140.         myMoteMemory.setIntValueOf("simOutSizeDummy", 0);
141.         isTransmitting = false;
142.         lastEvent = RadioEvent.HW_OFF;
143.       } else {
144.         lastEvent = RadioEvent.HW_ON;
145.       }
146.
147.       lastEventTime = now;
148.       this.setChanged();
149.       this.notifyObservers();
150.     }
151.     if (!radioOn) {
152.       return;
153.     }
154.
155.     /* Check if radio output power changed */
156.     if (myMoteMemory.getByteValueOf("simPowerDummy") != oldOutputPowerIndicator) {
157.       oldOutputPowerIndicator = myMoteMemory.getByteValueOf("simPowerDummy");
158.       lastEvent = RadioEvent.UNKNOWN;
159.       this.setChanged();
160.       this.notifyObservers();
161.     }
162.
163.     /* Check if radio channel changed */
164.     if (getChannel() != oldRadioChannel) {
165.       oldRadioChannel = getChannel();
166.       lastEvent = RadioEvent.UNKNOWN;
167.       this.setChanged();
168.       this.notifyObservers();
169.     }
170.
171.     /* Ongoing transmission */
172.     if (isTransmitting && now >= transmissionEndTime) {
173.       myMoteMemory.setIntValueOf("simOutSizeDummy", 0);
174.       isTransmitting = false;
175.       mote.requestImmediateWakeup();
176.
177.       lastEventTime = now;
178.       lastEvent = RadioEvent.TRANSMISSION_FINISHED;
179.       this.setChanged();
180.       this.notifyObservers();
181.       /*logger.debug("----- CONTIKI TRANSMISSION ENDED -----");*/
182.     }
183.
184.     /* New transmission */
185.     int size = myMoteMemory.getIntValueOf("simOutSizeDummy");
186.     if (!isTransmitting && size > 0) {
```

```
187.      packetFromMote = new COOJARadioPacket(myMoteMemory.getByteArray("simOutDataBufferDummy",
     size + 2));
188.
189.      if (packetFromMote.getPacketData() == null || packetFromMote.getPacketData().length == 0
    ) {
190.        logger.warn("Skipping zero sized Contiki packet (no buffer)");
191.        myMoteMemory.setIntValueOf("simOutSizeDummy", 0);
192.        mote.requestImmediateWakeup();
193.        return;
194.      }
195.
196.      byte[] data = packetFromMote.getPacketData();
197.      CCITT_CRC txCrc = new CCITT_CRC();
198.      txCrc.setCRC(0);
199.      for (int i = 0; i < size; i++) {
200.        txCrc.addBitrev(data[i]);
201.      }
202.      data[size] = (byte)txCrc.getCRCHi();
203.      data[size + 1] = (byte)txCrc.getCRCLow();
204.
205.      isTransmitting = true;
206.
207.      /* Calculate transmission duration (us) */
208.      /* XXX Currently floored due to millisecond scheduling! */
209.      long duration = (int) (Simulation.MILLISECOND*((8 * size /*bits*/) / RADIO_TRANSMISSION_
    RATE_kbps));
210.      transmissionEndTime = now + Math.max(1, duration);
211.
212.      lastEventTime = now;
213.      lastEvent = RadioEvent.TRANSMISSION_STARTED;
214.      this.setChanged();
215.      this.notifyObservers();
216.      //logger.debug("----- NEW CONTIKI TRANSMISSION DETECTED -----");
217.
218.      // Deliver packet right away
219.      lastEvent = RadioEvent.PACKET_TRANSMITTED;
220.      this.setChanged();
221.      this.notifyObservers();
222.      //logger.debug("----- CONTIKI PACKET DELIVERED -----");
223.    }
224.
225.    if (isTransmitting && transmissionEndTime > now) {
226.      mote.scheduleNextWakeup(transmissionEndTime);
227.    }
228.  }
229.
230. /* Packet radio support */
231.  public RadioPacket getLastPacketTransmitted() {
232.    return packetFromMote;
233.  }
234.
235.  public RadioPacket getLastPacketReceived() {
236.    return packetToMote;
237.  }
238.
239.  public void setReceivedPacket(RadioPacket packet) {
240.    packetToMote = packet;
241.  }
242.
243.  /* General radio support */
244.  public boolean isRadioOn() {
245.    return radioOn;
246.  }
247.
248.  public boolean isTransmitting() {
249.    return isTransmitting;
```

```
250.   }
251.
252.   public boolean isReceiving() {
253.     return myMoteMemory.getByteValueOf("simReceivingDummy") == 1;
254.   }
255.
256.   public boolean isInterfered() {
257.     return isInterfered;
258.   }
259.
260.   public int getChannel() {
261.     return myMoteMemory.getIntValueOf("simRadioChannelDummy");
262.   }
263. public void signalReceptionStart() {
264.     packetToMote = null;
265.     if (isInterfered() || isReceiving() || isTransmitting()) {
266.       interfereAnyReception();
267.       return;
268.     }
269.
270.     myMoteMemory.setByteValueOf("simReceivingDummy", (byte) 1);
271.     mote.requestImmediateWakeup();
272.
273.     lastEventTime = mote.getSimulation().getSimulationTime();
274.     lastEvent = RadioEvent.RECEPTION_STARTED;
275.
276.     myMoteMemory.setInt64ValueOf("simLastPacketTimestampDummy", lastEventTime);
277.
278.     this.setChanged();
279.     this.notifyObservers();
280.   }
281.
282.   public void signalReceptionEnd() {
283.     if (isInterfered || packetToMote == null) {
284.       isInterfered = false;
285.       packetToMote = null;
286.       myMoteMemory.setIntValueOf("simInSizeDummy", 0);
287.     } else {
288.       myMoteMemory.setIntValueOf("simInSizeDummy", packetToMote.getPacketData().length - 2);
289.       myMoteMemory.setByteArray("simInDataBufferDummy", packetToMote.getPacketData());
290.     }
291.
292.     myMoteMemory.setByteValueOf("simReceivingDummy", (byte) 0);
293.     mote.requestImmediateWakeup();
294.     lastEventTime = mote.getSimulation().getSimulationTime();
295.     lastEvent = RadioEvent.RECEPTION_FINISHED;
296.     this.setChanged();
297.     this.notifyObservers();
298.   }
299. public RadioEvent getLastEvent() {
300.     return lastEvent;
301.   }
302.
303.   public void interfereAnyReception() {
304.     if (isInterfered()) {
305.       return;
306.     }
307.
308.     isInterfered = true;
309.
310.     lastEvent = RadioEvent.RECEPTION_INTERFERED;
311.     lastEventTime = mote.getSimulation().getSimulationTime();
312.     this.setChanged();
313.     this.notifyObservers();
314.   }
315.
```

```
316.   public double getCurrentOutputPower() {
317.     /* TODO Implement method */
318.
319.     return 0;
320.   }
321.
322.   public int getOutputPowerIndicatorMax() {
323.     return 100;
324.   }
325.
326.   public int getCurrentOutputPowerIndicator() {
327.     return myMoteMemory.getByteValueOf("simPowerDummy");
328.   }
329.
330.   public double getCurrentSignalStrength() {
331.     return myMoteMemory.getIntValueOf("simSignalStrengthDummy");
332.
333.
334.   }
335.
336.   public void setCurrentSignalStrength(double signalStrength) {
337.     myMoteMemory.setIntValueOf("simSignalStrengthDummy", (int) signalStrength);
338.   }
339. public void setLQI(int lqi){
340.     if(lqi<0) {
341.       lqi=0;
342.     }
343.     else if(lqi>0xff) {
344.       lqi=0xff;
345.     }
346.     myMoteMemory.setIntValueOf("simLQIDummy", lqi);
347.   }
348.
349.   public int getLQI(){
350.     return myMoteMemory.getIntValueOf("simLQIDummy");
351.   }
352.
353.   public Position getPosition() {
354.     return mote.getInterfaces().getPosition();
355.   }
356.
357.
358.
359.
360.   public JPanel getInterfaceVisualizer() {
361.     JPanel panel = new JPanel(new BorderLayout());
362.     Box box = Box.createVerticalBox();
363.
364.     final JLabel statusLabel = new JLabel("");
365.     final JLabel lastEventLabel = new JLabel("");
366.     final JLabel channelLabel = new JLabel("");
367.     final JLabel ssLabel = new JLabel("");
368.     final JButton updateButton = new JButton("DummyRadio");
369.
370.     box.add(statusLabel);
371.     box.add(lastEventLabel);
372.     box.add(ssLabel);
373.     box.add(updateButton);
374.     box.add(channelLabel);
375.
376.     updateButton.addActionListener(new ActionListener() {
377.       public void actionPerformed(ActionEvent e) {
378.         ssLabel.setText("Signal strength (not auto-updated): "
379.             + String.format("%1.1f", getCurrentSignalStrength()) + " dBm");
380.       }
381.     });
```

```
382.
383.    final Observer observer = new Observer() {
384.      public void update(Observable obs, Object obj) {
385.        if (isTransmitting()) {
386.          statusLabel.setText("Transmitting");
387.        } else if (isReceiving()) {
388.          statusLabel.setText("Receiving");
389.        } else {
390.          statusLabel.setText("Listening");
391.        }
392.
393.        lastEventLabel.setText("Last event: " + getLastEvent());
394.        ssLabel.setText("Signal strength (not auto-updated): "
395.            + String.format("%1.1f", getCurrentSignalStrength()) + " dBm");
396.        if (getChannel() == -1) {
397.          channelLabel.setText("Current channel: ALL");
398.        } else {
399.          channelLabel.setText("Current channel: " + getChannel());
400.        }
401.      }
402.    };
403.    this.addObserver(observer);
404.
405.    observer.update(null, null);
406.
407.    panel.add(BorderLayout.NORTH, box);
408.    panel.putClientProperty("intf_obs", observer);
409.    return panel;
410.  }
411.
412.  public void releaseInterfaceVisualizer(JPanel panel) {
413.  }
414.
415.  public Collection<Element> getConfigXML() {
416.   ArrayList<Element> config = new ArrayList<Element>();
417.
418.           Element element;
419.
420.           /* Radio bitrate */
421.           element = new Element("bitrate");
422.           element.setText("" + RADIO_TRANSMISSION_RATE_kbps);
423.           config.add(element);
424.
425.           return config;
426.  }
427.
428.  public void setConfigXML(Collection<Element> configXML, boolean visAvailable) {
429. for (Element element : configXML) {
430.                 if (element.getName().equals("bitrate")) {
431.                         RADIO_TRANSMISSION_RATE_kbps = Double.parseDouble(element.getText());

432. logger.info("Radio bitrate reconfigured to (kbps): " + RADIO_TRANSMISION_RATE_kbps);
433.                 }
434.         }
435.  }
436.public Mote getMote() {
437.    return mote;
438.  }
439.
440.  public String toString() {
441.    return "DummyInterface";
442.  }
443.
444.}
```

8.4 MoteInterfaceHandler (Showing only the parts i have added)

```
                                        (.....)
    private Position myPosition;
    private Radio myRadio;
    private Radio myRadioDummy;
    private PolledBeforeActiveTicks[] polledBeforeActive = null;
    private PolledAfterActiveTicks[] polledAfterActive = null;
    private PolledBeforeAllTicks[] polledBeforeAll = null;
    private PolledAfterAllTicks[] polledAfterAll = null;

/*---------------------------------------------------------------------------*/
                                        (.....)
/*---------------------------------------------------------------------------*/
    * Returns the radio interface (if any).
    *
    * @return Radio interface
    */
   public Radio getRadio() {
     if (myRadio == null) {
       myRadio = getInterfaceOfType(Radio.class);
     }
     return myRadio;
   }
   public Radio getDummyRadio() {
     if (myRadioDummy == null) {
       myRadioDummy = getInterfaceOfType(DummyInterface.class);
     }
     return myRadioDummy;
   }
/*---------------------------------------------------------------------------*/
                                        (.....)
/*---------------------------------------------------------------------------*/

```

8.5 RadioMedium (Showing only the parts I have added)

```
/*---------------------------------------------------------------------------*/
                                        (.....)
/*---------------------------------------------------------------------------*/

 public UDGM(Simulation simulation) {
    super(simulation);
    random = simulation.getRandomGenerator();
    dgrm = new DirectedGraphMedium() {
      protected void analyzeEdges() {
        /* Create edges according to distances.
         * XXX May be slow for mobile networks */
        clearEdges();
        for (Radio source: UDGM.this.getRegisteredRadios()) {
          Position sourcePos = source.getPosition();
          for (Radio dest: UDGM.this.getRegisteredRadios()) {
            Position destPos = dest.getPosition();
            /* Ignore ourselves */
            if (source.getPosition().equals(dest.getPosition()))
            {
```

```
20.
21.               continue;
22.             }
23.
24.             double distance = sourcePos.getDistanceTo(destPos);
25.
26.             if (distance < Math.max(TRANSMITTING_RANGE, INTERFERENCE_RANGE))
27.             {
28.               /* Add potential destination */
29.               addEdge(
30.                   new DirectedGraphMedium.Edge(source,
31.                       new DGRMDestinationRadio(dest)));
32.             }
33.
34.
35.
36.           }
37.         }
38.         super.analyzeEdges();
39.       }
40.     };
41.
42.  public RadioConnection createConnections(Radio sender) {
43.     RadioConnection newConnection = new RadioConnection(sender);
44.
45.     /* Fail radio transmission randomly - no radios will hear this transmission */
46.
47.
48.
49.
50.     /* Calculate ranges: grows with radio output power */
51.     double moteTransmissionRange = TRANSMITTING_RANGE
52.     * ((double) sender.getCurrentOutputPowerIndicator() / (double) sender.getOutputPowerIndica
    torMax());
53.     double moteInterferenceRange = INTERFERENCE_RANGE
54.     * ((double) sender.getCurrentOutputPowerIndicator() / (double) sender.getOutputPowerIndica
    torMax());
55.
56.     /* Get all potential destination radios */
57.     DestinationRadio[] potentialDestinations = dgrm.getPotentialDestinations(sender);
58.     if (potentialDestinations == null) {
59.       return newConnection;
60.     }
61.
62.     /* Loop through all potential destinations */
63.     Position senderPos = sender.getPosition();
64.     for (DestinationRadio dest: potentialDestinations) {
65.       Radio recv = dest.radio;
66.
67.
68.       if (sender.getPosition().equals(recv.getPosition())){
69.
70.               continue;
71.       }
72.
73.
74.
75.       /* Fail if radios are on different (but configured) channels */
76.       if (sender.getChannel() >= 0 &&
77.           recv.getChannel() >= 0 &&
78.           sender.getChannel() != recv.getChannel()) {
79.
80.         /* Add the connection in a dormant state;
81.            it will be activated later when the radio will be
82.            turned on and switched to the right channel. This behavior
83.            is consistent with the case when receiver is turned off. */
```

```
84.           newConnection.addInterfered(recv);
85.
86.           continue;
87.         }
88.         Position recvPos = recv.getPosition();
89.
90.         /* Fail if radio is turned off */
91. //        if (!recv.isReceiverOn()) {
92. //          /* Special case: allow connection if source is Contiki radio,
93. //           * and destination is something else (byte radio).
94. //           * Allows cross-level communication with power-saving MACs. */
95. //          if (sender instanceof ContikiRadio &&
96. //              !(recv instanceof ContikiRadio)) {
97. //            /*logger.info("Special case: creating connection to turned off radio");*/
98. //          } else {
99. //            recv.interfereAnyReception();
100.//            continue;
101.//          }
102.//        }
103.
104.       double distance = senderPos.getDistanceTo(recvPos);
105.       if (distance <= moteTransmissionRange)
106.        {
107.         /* Within transmission range */
108.
109.
110.         if (!recv.isRadioOn()) {
111.
112.           newConnection.addInterfered(recv);
113.           recv.interfereAnyReception();
114.         } else if (recv.isInterfered()) {
115.
116.           // Was interfered: keep interfering
117.           newConnection.addInterfered(recv);
118.         } else if (recv.isTransmitting()) {
119.
120.           newConnection.addInterfered(recv);
121.         } else if (recv.isReceiving()) {
122.
123.           // Was receiving, or reception failed: start interfering
124.           newConnection.addInterfered(recv);
125.           recv.interfereAnyReception();
126.
127.           // Interfere receiver in all other active radio connections
128.           for (RadioConnection conn : getActiveConnections()) {
129.             if (conn.isDestination(recv)) {
130.               conn.addInterfered(recv);
131.             }
132.           }
133.
134.         } else {
135.
136.           /* Success: radio starts receiving */
137.           /*
138.           System.out.println("Source " + sender.getPosition() + " sender til " + recv.getPosit
    ion());
139.           System.out.println("SourceRADIO " + sender.toString() + " sender til RecvRADIO " + r
    ecv.toString());
140.          Clock clock = Clock.systemDefaultZone();
141.           System.out.println(clock.instant());
142.
143.         */
144.          newConnection.addDestination(recv);
145.         }
146.       }
147.        else if (distance <= moteInterferenceRange) {
```

```
        /* Within interference range */
        newConnection.addInterfered(recv);
        recv.interfereAnyReception();
      }
    }

    return newConnection;
  }
/*---------------------------------------------------------------------------*/
                                    (.....)
/*---------------------------------------------------------------------------*/
```

8.6 TSCH EB scanning (Showing only the parts I have added)

```
/*---------------------------------------------------------------------------*/
                                    (.....)
/*---------------------------------------------------------------------------*/


    if(current_channel == 0 || now_time -
 current_channel_since > TSCH_CHANNEL_SCAN_DURATION) {

// Choosing a random channel for radio 1
      /* Pick a channel at random in TSCH_JOIN_HOPPING_SEQUENCE */
      uint8_t scan_channel = TSCH_JOIN_HOPPING_SEQUENCE[
          random_rand() % sizeof(TSCH_JOIN_HOPPING_SEQUENCE)];

    do
    {
// Choosing a random channel for radio 2 and making sure that
they are listening on to different channels.
      scan_channeldummy=TSCH_JOIN_HOPPING_SEQUENCE[
          random_rand() % sizeof(TSCH_JOIN_HOPPING_SEQUENCE)];
    }
    while(scan_channel==scan_channeldummy);

      if(current_channel != scan_channel) {
// Setting channel on both radios
        NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, scan_channel);
        NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNELDummy, scan_channeldummy);
        current_channel = scan_channel;
      }
      current_channel_since = now_time;
    }

    /* Turn radio on and wait for EB */
    NETSTACK_RADIO.on();
/*---------------------------------------------------------------------------*/
                                    (.....)
/*---------------------------------------------------------------------------*/
```

9 Appendix C

9.1 Java-Script for 2-hop simulation

```
1.  TIMEOUT(100000000000);
2.  while(true)
3.  {
4.
5.          if(msg.contains("Ready"))
6.  // When the node has finished sending all of its UDP packages,
    it prints out "Ready". Then I log all information before
    closing the simulation.
7.          {
8.
9.                  log.log(abc+"\n");
10.                 log.log(server+"\n");
11.                 log.log(client+"\n");
12.
13.                 // plugin saves duty cycle information from all the nodes in the simulation.
14.                 plugin = mote.getSimulation().getCooja().getStartedPlugin("PowerTracker");
15.                 log.log("PowerTracker: Extracted statistics:\n" + plugin.radioStatistics() + "\n");
16.                 log.testOK(); /* Report test success and quit */
17.
18.         }
19.         else
20.         {
21.                 if(msg.contains("Client Energest CPU"))
22.                 {
23. // If the message sent from the node contains the message Client Energest
    CPU, I save data in a variable called client. This will be overwritten
    every time the client prints CPU information because I'm only interested in
    the latest CPU data.
24.                   client = msg;
25.                 }
26.                 else if(msg.contains("Server Energest CPU"))
27.                 {
28.                    // Same as above, it only stores information about the server.
29.                    server = msg;
30.                 }
31.                 else
32.                 {
33.                    abc = msg;
34.                 }
35.         }
36.
37.
38.
39.
40.      YIELD(); // Wait for the next time the node prints something
41.
42. }
```

9.2 Java-Script for joining the simulation

```
1.  TIMEOUT(100000000000);
2.  while(true)
3.  {
4.
5.          if(msg.contains("Time = "))
6.           {
7.             log.log(msg+"\n");
8.             log.testOK(); /* Report test success and quit */
9.  // Joining node printes out «Time» followed by timestamp. This will be
    saved, and the test will end.
10.          }
11.         else
12.         {
13.         }
14.       YIELD();
15.
16. }
```

9.3 Java-Script for Reliability and latency

```
1.  TIMEOUT(100000000000);
2.
3.  tx1 = "0"
4.  tx2 = "0"
5.  tx3 = "0"
6.  tx4 = "0"
7.  tx5 = "0"
8.  tx6 = "0"
9.  tx7 = "0"
10. WAIT_UNTIL(msg.contains("time")) //
11. while(true)
12. {
13.          // When the node has finished sending all of its UDP packages, it
    prints out "Ready". Then I log all information before closing the
    simulation.
14.
15.         if(msg.contains("Ready"))
16.          {
17.          // Logs the number of packages received with one, two, three etc. transmissions.
18.
19.  log.log("Tx1 = "+ tx1 +" Tx2 = "+tx2+ " Tx3 = "+tx3 + " Tx4 = "+tx4 + " Tx5 = "+tx5 + " Tx6 = " + tx6 + "
    Tx7 = "+tx7+"\n");
20.                  log.log(total+"\n");
21.                  plugin = mote.getSimulation().getCooja().getStartedPlugin("PowerTra
    cker");
22.                  log.log("PowerTracker: Extracted statistics:\n" + plugin.radioStati
    stics() + "\n");
23.                  log.log(Server+"\n");
24.                  log.log(Client+"\n");
25.                  log.log(radio+"\n")
26.                  log.testOK(); /* Report test success and quit */
27.          }
28.
29.          // Logs the number of received packages, used to calculate PDR
30.
31.          else if (msg.contains("Reciving package = "))
32.          {
33.              total=msg;
34.          }
35.
36.
37.          else if(msg.contains("Server Energest CPU:"))
38.          {
39.                Server = msg;
40.          }
41.          else if(msg.contains("Client Energest CPU:"))
42.          {
43.                Client = msg;
44.          }
45.          else if(msg.contains("tx"))
46.          {
47.
    // The sentences below count how many of the transmissions need one, two, or three
    transmissions.
48.
49.                 if(msg.contains("tx=1"))
50.                  {
51.             tx1++;
52.                  }
53.                 else if(msg.contains("tx=2"))
```

```
                {
                        tx2++;
                }
               else if(msg.contains("tx=3"))
                {
                        tx3++;
                }
               else if(msg.contains("tx=4"))
                {
                        tx4++;
                }
               else if(msg.contains("tx=5"))
                {
                        tx5++;
                }
                else if(msg.contains("tx=6"))
                {
                        tx6++;
                }
                else if(msg.contains("tx=7"))
                {
                        tx7++;
                }
            }
            // Saves radio distribution

          else if(msg.contains("Radio1 ="))
                {
                        radio = msg;
                }


      YIELD();

}
```