Coexistence Study on Different Medium Access Mechanisms Using a Software Defined Radio Testbed

Alexander Pastor

Bachelor Thesis

October 24, 2017

Examiners

Prof. Dr. Petri Mähönen
Prof. Dr.-Ing. Marina Petrova

Supervisors

Prof. Dr. Petri Mähönen
Peng Wang, M.Sc.
Andra Voicu, M.Sc.

Institute for Networked Systems
RWTH Aachen University

The present work was submitted to the Institute for Networked Systems

Coexistence Study on Different Medium Access Mechanisms Using a Software Defined Radio Testbed

Bachelor Thesis

presented by
Alexander Pastor

Prof. Dr. Petri Mähönen
Prof. Dr.-Ing. Marina Petrova

Aachen, October 24, 2017

(Alexander Pastor)

Acknowledgements

We thank...

Contents

ABSTRACT

Abstract here.

1

INTRODUCTION

Introduction here.

2

BACKGROUND

In this chapter the theoretical foundations for the succeeding work are treated. Firstly, the MAC layer is introduced in the context of the OSI reference model. Successively, a glance on a number of different MAC protocols and mechanisms is taken, while analyzing performance with respect to the challenges and goals in wireless transmission. The chapter concludes with describing the advantages of software-defined radio (SDR) and how GNU Radio can be used to create SDR.

2.1 MAC PROTOCOLS

2.1.1 MAC Layer in the OSI Model

The OSI model is a layered architecture that divides a telecommunication system into several manageable layers. The second of seven layers featured by the original model is the data link layer, which in turn is split into the upper logical link control (LLC) and the lower medium access control (MAC) sublayers. Table 2.1 gives a short overview of the layers' responsibilities.

Layer	Responsibilities
Physical Layer	dealing with mechanical, electrical and timing interfaces of data transmission
MAC Sublayer	controlling medium access and frame synchronization
LLC Sublayer	multiplexing to enable different network protocols coexist, flow control and error control
Network Layer	routing and congestion control
Transport Layer	transmission reliability, same-order-delivery, congestion avoidance
Session Layer	token management, dialog control, synchronization
Presentation Layer	abstracting syntax and semantics of transmission, encryption
Application Layer	user application protocols, such as http, ftp, smtp and many more

TABLE 2.1: Layers in the OSI model

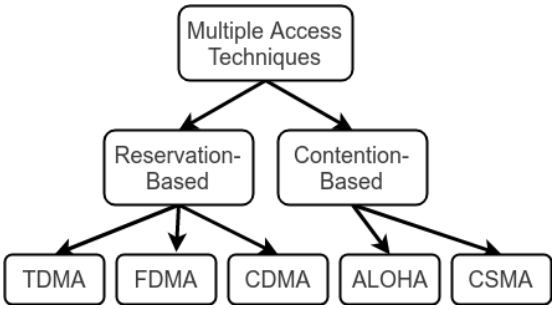


FIGURE 2.1: Classification of MAC techniques according to [1]

2.1.2 Classification of MAC Protocols

Traditional MAC protocols can be classified into one of two subgroups: reservation-based and contention-based. According to [2] the appropriate choice of MAC protocol depends on a plethora of design-drivers such as requirements concerning throughput, latency, energy consumption and traffic patterns.

2.1.3 Reservation-Based MAC Protocols

Reservation-based protocols may implement an array of desirable features, but require knowledge of network topology in order to allow each node to communicate with every other on the basis of a schedule. These features include reduced collisions, fairness among nodes or multiple transmissions at the same time.

TDMA is a representative protocol in this group, which divides time into frames. Each frame is subdivided into slots, where each node is assigned to a unique slot during which it may transmit. As a result we obtain collision-free transmission, predictable scheduling delays, high throughput in heavy load situations and fairness among nodes. However, both the knowledge of topology and tight synchronization require large overheads or expensive hardware. As a result, TDMA becomes less attractive for large-scale networks [2].

2.1.4 ALOHA

ALOHA is arguably the most simple MAC protocol. Whenever a user wants to send data he just does so. The higher the channel load, i.e. sending requests per time unit, the more likely collisions will occur, which render all transmitted information useless.

The question is how likely it is that a collision does not occur. In other words, how efficient is an ALOHA channel? Making a statement requires a few preliminary assumptions as shown in [3]:

1. We are taking a look at pure ALOHA.
2. We simplify the calculation by assuming a fixed frame length.
3. The number of packets generated during a frame time is a poisson-distributed random variable X.

4. The channel load G comprises of two portions: "new" and retransmitted frames.

The probability mass function of the Poisson distribution and thus the probability of k frames being generated during a given frame time amounts to:

$$Pr(X = k) = \frac{G^k \cdot e^{-G}}{k!} \tag{2.1}$$

The probability of zero frames being generated during the transmission of the frame is $Pr(X = 0) = e^{-G}$ (assumption 3). If no collision occurs during the transmission of frame F no other frame was sent off during that transmission. Conversely, F itself did not collide with a frame sent off prior to F. We conclude that the vulnerability period during which collision may corrupt data is two frame times (assumption 2).

The probability that no frame other than the frame to be transmitted is generated during the two frame time vulnerability period is $P_0 = e^{-2G}$. The throughput S is given by $S = GP_0 = Ge^{-2G}$.

The maximum throughput is achieved when $\frac{\partial S}{\partial G} \stackrel{!}{=} 0$:

$$\begin{aligned} \frac{\partial S}{\partial G} &= \frac{\partial}{\partial G} G e^{-2G} && (2.2)\\ &= e^{-2G}(1 - 2G) && (2.3)\\ &\stackrel{!}{=} 0 && (2.4)\\ \Leftrightarrow \quad G &= 0.5 && (2.5) \end{aligned}$$

This means that for $G = 0.5$ the throughput S reaches its maximum $S_{\text{ALOHA,max}} = \frac{1}{2e} \approx 0.18$. This result is very reasonable, since the transmission of a frame is vulnerable for the duration of two frame times, so the maximum is achieved when sending exactly every second slot, where a slot is equivalent to the frame time.

As an aside, the throughput can be doubled with slotted ALOHA. In contrast to pure ALOHA, slotted ALOHA allows transmission only at the beginning of slots, which effectively halves the vulnerability period to only one slot, since frames transmitted prior to a frame F cannot interfere with F anymore. Thus, $S_{\text{ALOHA,max}} = \frac{1}{e} \approx 0.36$, reached at $G = 1$. However, this comes at the cost of an additional frame delay of t_{slot} in the worst case and $\frac{t_{\text{slot}}}{2}$ in the average case and the need for synchronization.

As shown in figure 2.2 ALOHA's performance is discouraging and improvements over ALOHA were found.

2.1.5 CSMA

Main problem of ALOHA is the negligence of concurrent traffic in the channel. A solution to this problem is offered by the "listen before talk" (LBT) mechanism, which means in order to avoid collisions we make a clear channel assessment (CCA) and refrain from sending should it be busy. This is the simple, yet effective basic idea of carrier sensing multiple access (CSMA) which comes in three flavors, as depicted in figure 2.3 which will be discussed next.

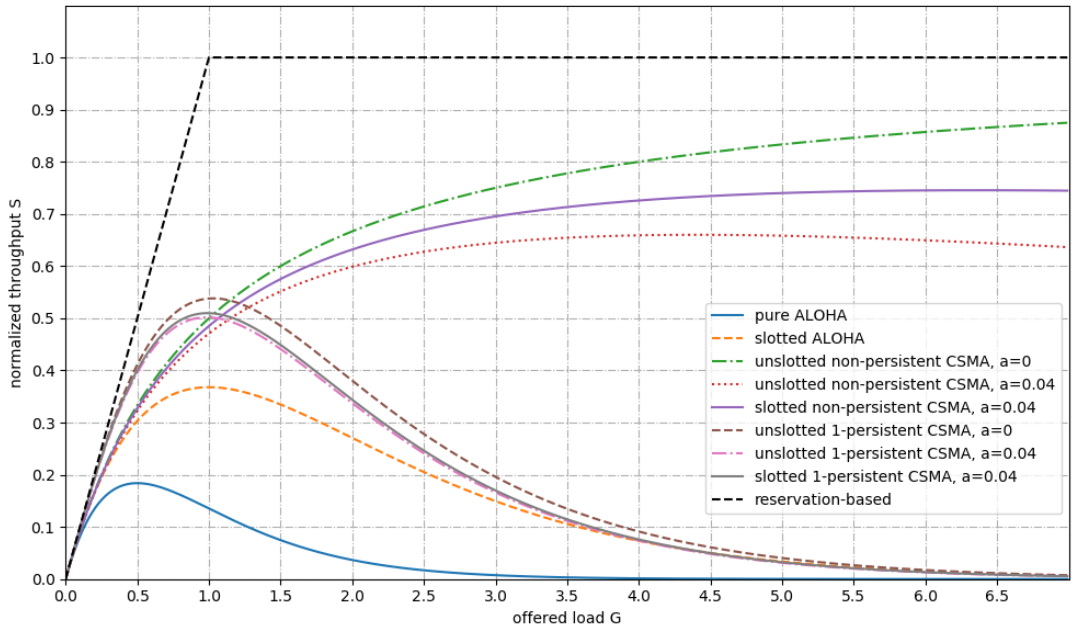


FIGURE 2.2: Normalized throughput over offered load according to formulae in [1], [2], [3], with $a = \tau/T_p$, where τ is the maximum propagation delay and T_p the packet transmission time and under the assumptions 2-4 made in section 2.1.4.

2.1.5.1 *1-Persistent CSMA*

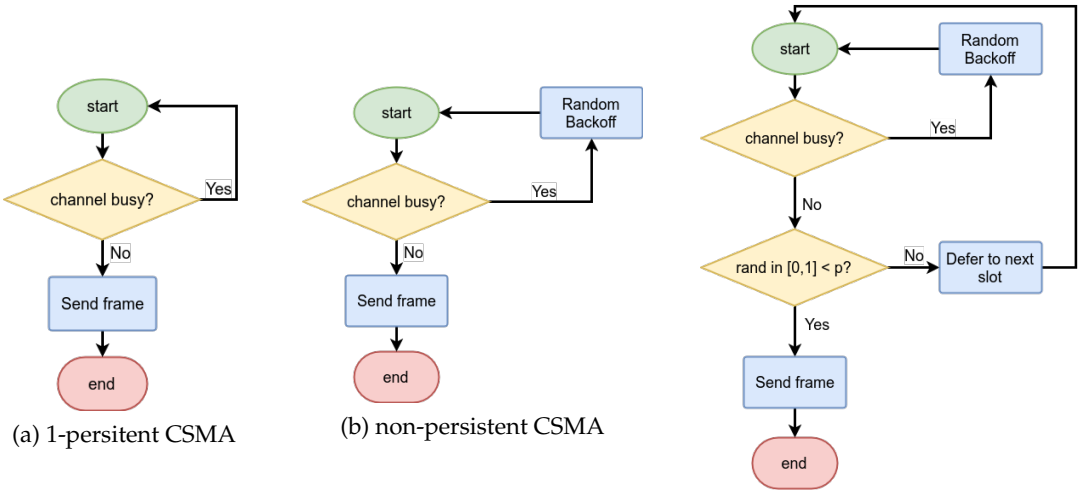
When the channel is busy 1-persistent CSMA waits until the channel becomes idle. As soon as the channel is found idle a frame is transmitted with a probability of 1, hence 1-persistent CSMA. If the frame collides with another, the node waits for a random backoff time and then the whole process is started all over again.

Despite being a substantial improvement over ALOHA, this protocol has at least two problems:

- Provided propagation delay is zero or negligible, collisions can still occur. Imagine a three-node-scenario with nodes A, B and C. A is transmitting, while B and C are waiting for their turn. Once A finished transmission B and C will lunge onto the channel like a pack of wolves leading to collision.
- If propagation delay is not negligible the protocol suffers from an additional problem. In this scenario A has just begun sending. B will assume the channel is idle and send off his frame, since, due to the propagation delay, B has not yet heard of A. This is why propagation delay may significantly hamper this protocol's performance.

2.1.5.2 *Non-Persistent CSMA*

In order to alleviate 1-persistent CSMA's problem with several nodes trying to seize the channel as soon as it becomes idle a less greedy attempt is made with non-persistent



(a) 1-persitent CSMA (b) non-persistent CSMA (c) p-persistent CSMA

FIGURE 2.3: The three flavors of CSMA

CSMA. Instead of continuously sensing the channel until it becomes idle nodes wait a random backoff time until they listen again. As a result, this protocol leads to better channel utilization with the downside of higher delays.

2.1.5.3 P-Persistent CSMA

P-persitent CSMA is a protocol for slotted channels. Whenever a node A wishes to send off a packet the channel is sensed. If the channel is found idle it transmits its packet with a probability of p. With a probability $1 - p$ it defers the transmissions to the next slot. This process is repeated until one either the packet is sent off or the channel is found busy again. In the latter case A acts as though a collision had taken place and waits a random time until starting again.

This flavor of CSMA can be regarded as a compromise between 1-persistent CSMA and non-persistent CSMA, where the choice of p determines the greediness. The smaller p, the less greedy and thus the closer p-persistent CSMA approximates non-persistent behavior. An appropriate choice of p can get the best out of both worlds: minimal delays as in 1-persistent CSMA, as well as high channel efficiency as in non-persistent CSMA.

2.1.6 CSMA with Collision Detection

A way to further improve CSMA-family protocols is to immediately cancel transmissions once a collision is detected. There is no point in continuing these transmissions, as the transmitted data is lost in any case and aborting the transmission saves bandwidth, time and energy.

CSMA/CD is used on wired LANs and serves as basis of the wide-spread Ethernet. However, this mechanism is not extensively made use of in wireless networks. Concerning the reason, it is cardinal to understand that collision detection is an analog

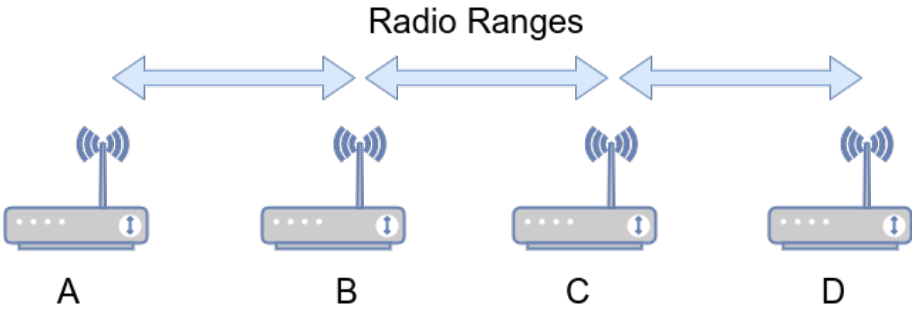


FIGURE 2.4: Setup to explain the hidden and exposed node problem. Each node can only reach its neighbors.

process. A collision is detected by comparing the received and transmitted signal's energy or pulse width, which premises transmission and reception taking place simultaneously. This condition is seldom met for wireless nodes, which are mostly half-duplex. The reason for this lies in the conservation of energy, since wireless signal spreads spherically around its origin and thus degrades with the second order of distance. Furthermore, wireless channels are typically much more noisy than their wired counterparts and suffer from multipath fading. To make up for the loss in signal strength we would have to employ expensive signal processing in order to recover fainter signals. Alternatively, we could increase the transmit power, but this increases interference with other nodes, as well as electricity consumption.

2.1.7 Challenges for Wireless MAC Protocols

Wireless MAC protocols have to tackle a few problems that do not occur in wired data exchange. Among them are the hidden node and the exposed node problem, which will be discussed by reference to 2.4. Further challenges, such as energy limitations will also be delineated.

2.1.7.1 The Hidden Node and the Exposed Node Problem

Suppose that the node's radio range is limited to the neighboring nodes and A would like to transmit to B. If C just started transmitting A won't hear C and falsely assume that the channel is idle and start transmitting. This is the hidden node problem.

For the same configuration, in another scenario B would like to send to A and C is already transmitting to D. B refrains from sending despite collisions would only take place between B and C, where it does not matter. This is the exposed node problem.

2.1.7.2 Further Challenges

Further challenges to MAC protocol design include the power conservation when faced with constrained power resources, as in wireless sensor networks (WSN) where devices rely on batteries for their supply with power. Attempts to mitigate waste of energy have been made in several specialized, duty-cycle based MACs such as Sensor MAC, Timeout MAC and Berkeley MAC as in more detail shown in section 2.1.9.

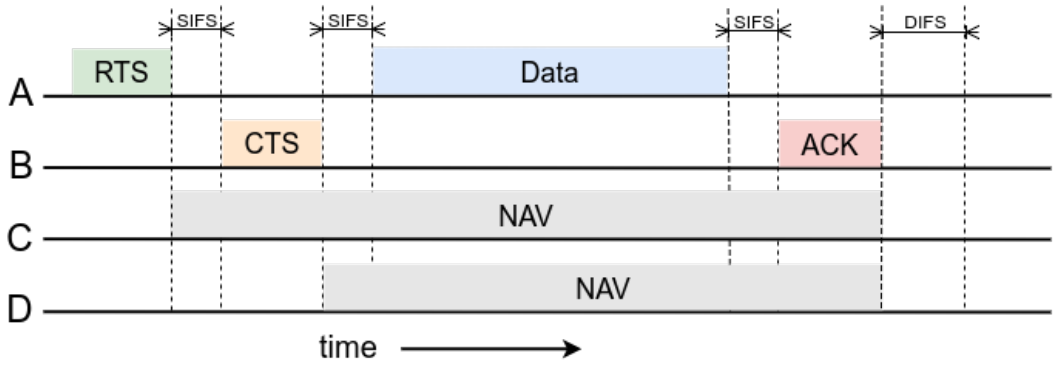


FIGURE 2.5: Virtual carrier sensing in CSMA/CA, as described in [3] and [4]

On the same page, due to constrained energy resources, WSN are especially susceptible to denial of sleep attacks, a special form of denial of service (DoS) attack, drastically increasing energy consumption and thus reducing the system's lifetime. It is due to this fact that security is paramount in biomedical or military fields of application.

2.1.8 CSMA with Collision Avoidance

802.11 is a set of physical layer (PHY) and MAC specifications for wireless local area networks (WLANs). When the dominant mode of operation, the so-called distributed coordination function (DCF) is employed CSMA/CA is used in the MAC layer.

Beside physical carrier sensing previously simply referred to as carrier sensing another mechanism, namely virtual carrier sensing in combination with RTS/CTS exchange is (optionally) employed to mitigate the trouble caused by hidden nodes.

In order to explain these mechanisms we refer to the setup of figure 2.4 with a slight modification in so far as that each node's radio range shall span across two neighboring nodes in both directions. That is to say, A can hear B and C, but not D and so on. Figure 2.5 visualizes the chain of events whose explanation follows.

A wants to send to B, hence issues a request to send (RTS). Every node receiving the RTS is shut down, except for B that in response to the RTS creates a clear to send (CTS) frame. Not only A receives this CTS frame, but also D, a hidden node from A's point of view. Upon reception of CTS D is silenced as well. Therefore, RTS/CTS is addressing the hidden node problem. RTS/CTS are frames of 30 bytes length containing the length of the frame in this case A wants to transmit. Based on this length C and D setup the so-called network allocation vector (NAV), which are node-internal timers reminding C and D that the channel is still in use. This mechanism is called virtual carrier sensing due to the fact that no physical process is involved in obtaining the channel status. Shutting down nodes has the beneficial side-effect of reducing overhearing and therefore reduces energy consumption.

As further depicted in figure 2.5 there are named intervals of specified length between each of the frames. Varying lengths of these interval types serve the purpose of prioritizing certain frames over others.

The short interframe spacing (SIFS) is the interval until the next control frame or next fragment (of a fragmented data frame) may be sent. SIFS is designed to allow one party out of the two parties in dialog send off their frame without interference by another node. The longer interval DCF interframe spacing (DIFS) is the interval after which any station may try to seize the channel for their transmission.

For the sake of completeness, we briefly mention two previously consciously left out intervals, namely point coordination function interframe spacing (PIFS) and extended interframe spacing (EIFS). If 802.11 is operates in an alternative mode of operation, where the base station acts as point coordinator of traffic the standard prescribes an interval of length PIFS to allow the base station to send certain control (beacon and poll) frames. EIFS is used to report the reception of a bad or unknown frame and due to this action's low priority is the longest interval among the mentioned four.

2.1.9 Duty-Cycle-Based MAC Protocols

In duty-cycle-based MAC schemes nodes repeatedly alternate between active and sleep phases to reduce idle listening and thus energy consumption. Due to increased contention during active phases these protocols are mostly designed for limited contention traffic situations as in WSNs. The share of an active period in a cycle is called duty factor. Sources to this section were [2] and [5].

2.1.9.1 Sensor MAC

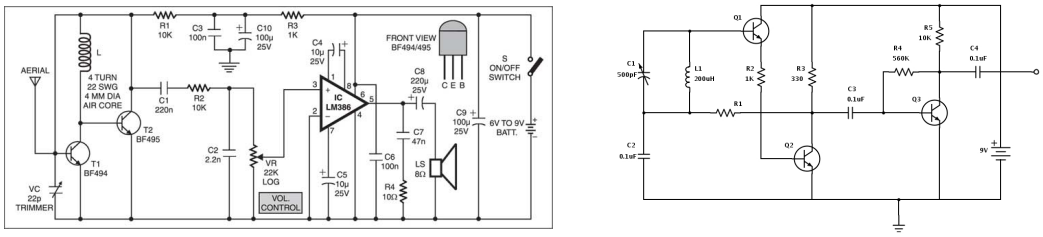
In SMAC the active period is divided into to a synchronization and a data transmission phase. During sync phase nodes transmit SYNC packets. Nodes receiving SYNC packets adopt the schedule carried by the packet and broadcast into their neighborhood. Nodes that follow the same schedule form a virtual cluster. Borderline nodes between virtual clusters adopt multiple schedules and thus have an increased duty factor. During contention period SMAC features the RTS/CTS exchange and fragments data frames, which are transmitted in a burst to reduce collision likelihood. The duty factor per schedule is predetermined on the basis of expected load as the result of an optimization problem on the competing goals of reducing idle listening and contention. The higher the duty factor the more idle-listening and the less contention occurs.

2.1.9.2 Timeout MAC

While TMAC shares the same principle of schedule establishment with SMAC nodes adaptively vary duty factors depending on expected traffic. Furthermore, TMAC shifts all communication to the beginning of the active period. This allows nodes to sleep earlier should no traffic be detected during a certain time period. In variable load situations TMAC saves as much as five times more energy compared to SMAC at the cost of increased latency.

2.1.9.3 Berekeley MAC

Still, TMAC maintains common active phases at high energy expenses. BMAC drops the requirement of maintaining common active phases. Instead payload is preceded



(a) FM Receiver

(b) AM Receiver

FIGURE 2.6: Simple DIY radio receiver circuit diagrams

by extended preambles such that every receiver is able to reliably detect packets. This has the effect of shifting energy expenses from the receiving to the sending side, which saves energy in low load applications such as surveillance. In BMAC CCA is based on outlier detection, instead of thresholding like in CSMA further reducing energy use [6].

2.2 SOFTWARE DEFINED RADIO

2.2.1 Purpose of Software Defined Radio

Traditional radio equipment is "hardware-defined", i.e. that the signal processing runs on a specialized electrical circuit. This has the potential advantages of efficient energy use and cheap production at the cost of limited flexibility in operation.

In SDR signal processing components such as filters, amplifiers, modulators, detectors and many more are implemented in software and mostly run on general-purpose processors, sometimes in combination with DSPs and FPGAs.

While the limitations of hardware-defined radios are acceptable for a number of applications, such as e.g. self-made radio receivers as shown in figure 2.6, it is very desirable to get rid of these limitations for rapid prototyping of new technologies including but not limited to cognitive radio, software-defined antennas and wireless mesh networks. In the case of this thesis SDR simplifies studying the influence of different MAC mechanisms.

2.2.2 What is GNU Radio?

The GNU Radio (GR) project is dedicated to the evolution of a free and open-source SDK enabling both the creation of actual software-defined radio, as well as simulated signal processing. Written in C++ and Python, GNU Radio also comes with the intuitive graphical software GNU Radio Companion (GRC) that allows creating block diagrams called flowgraphs simply by connecting signal processing blocks into a directed graph. Its audience is not merely limited to research and industry, but also encompasses academia, government and private users.

A proprietary, well-documented alternative to GNU Radio is LabVIEW developed by National Instruments. LabVIEW takes a purely graphical approach similar to GRC

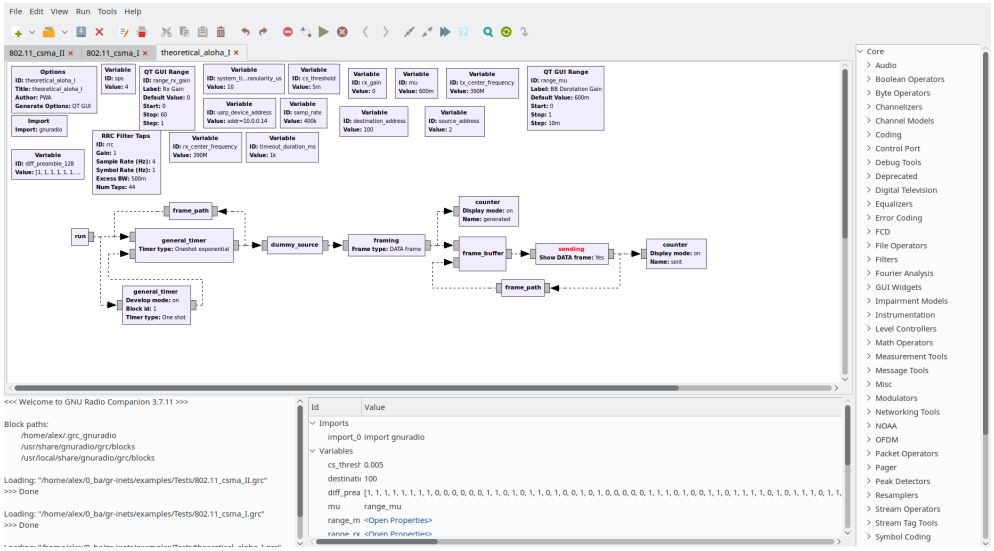


FIGURE 2.7: GNU Radio Companion GUI

relying on block diagrams, but lacks the freedom of user-defined block creation with a programming language such as C++ or Python.

Mathworks MATLAB/Simulink also provides a communication systems toolbox, however the institute's devices are not on the list of officially supported devices [7].

2.2.3 Core Concepts of GNU Radio

2.2.3.1 Flowcharts and Blocks

The two most basic concepts of GNU Radio are flowgraphs and blocks. As mentioned in 2.2.2 flowgraphs are directed graphs, whose nodes are functional blocks and whose vertices determine the direction of data flow [8].

The behavior of these blocks is programmed in either Python or C++, where the latter is recommended for performance-critical applications, which is also why the blocks in our flowgraphs are all written in C++. If performance is less critical Python is a superior choice since it is more concise and allows faster prototyping as there is no need for compilation. Each block generally serves exactly one purpose for the sake of modularity. Blocks in turn can be composed of an arbitrary number of inner blocks, making extensive use of the modularity and hiding implementation complexity from the user, much like a blackbox in electrical circuits. These composed blocks are called hierarchical blocks. In our case the complete PHY layer is hidden in hierarchical blocks called "sending" and "receiving".

Blocks are connected through ports, which can either be input or output ports. Depending on which types of ports a block has, it can either be a source, sink or neither of the former. Each input port only consumes data of a specific data type. Similarly, each output port only produces data of a specific data type. The set of types ranges from integers, floating point and complex numbers to messages and a bunch

of others. Since each block implements a certain function these ports can be regarded as input parameters and return values of a function, respectively.

2.2.3.2 Message Passing and Stream Tags

When designing packet-based protocols, such as MAC protocols it is of tremendous importance to be able to detect packet data unit (PDU) boundaries. For this purpose GR provides an asynchronous message passing system. A synchronous alternative is to attach so-called stream tags to the "infinite" stream of data. The former method is the right choice when designing MAC protocols due to the asynchronous nature of packet delivery [8] [9].

2.2.3.3 Polymorphic Types and SWIG

Polymorphic types (PMT) are opaque data types that enable safe information exchange across blocks by serving as generic containers of data. Self-evidently, the original data type must be retained as a PMT class member. For thread-safety reasons PMT are immutable. We make extensive use of PMTs when passing messages. As an aside, note that the python PMT class has some powerful tools unavailable its C++ counterpart, making use of Python's weak typing [9].

SWIG (simplified wrapper and interface generator) is a software that helps to connect code written in C or C++ to a variety of scripting languages, such as in our case Python. This is achieved by generating a Python module from the C/C++ code with the help of an interface file. This "compatibility layer" is necessary, because blocks can be written in either Python or C++ as mentioned earlier.

3

Related Work

Related Work here.

4

MEASUREMENT METHODOLOGY

This chapter is dedicated to answering the questions of what was measured and how results were obtained. Firstly, the measurement setup is discussed. Subsequently, the GNU Radio flowgraphs are explained. Secondly, with reference to the flowgraphs measurement metrics are formally defined in view of the necessity to verify the recorded data. Lastly, an overview of the semi-automatic measurement script system designed to automate, therefore accelerate the process of file system management, data processing and result plotting is given.

4.1 MEASUREMENT SETUP

The setup consists of two USRP2s from Ettus Research and two USRP 2920s from National Instruments. Where the former pair is programmed as receiver and sniffer and the latter as senders as depicted in 4.1. Each USRP was connected to a gigabit switch through a LAN cabel. The scripts running on the devices were launched from the institute's computer with the IP 134.130.223.151, which was remote controlled from my private laptop. Hereafter, we will call the node pair 10.0.0.9-10.0.0.6 link 1 and 10.0.0.3-10.0.0.6 link 2. Table 4.1 contains other necessary information to reproduce the measurement results.

4.2 GNU RADIO FLOWGRAPHS

4.2.1 Receiver and Sniffer

Figure 4.2.1 shows the two-way handshake receiving logic. After frame integrity is checked and type is confirmed to be data frame an acknowledgment is generated. The `frame_probe` blocks record the times when the frames reach certain positions in the flowgraph representing the occurrence of events such as frame reception, passed or failed frame integrity check and more. Note that the address check is disabled so that the receiver may receive frames from any sender.

The sniffer consists only of a single `frame_probe` block, which records detected power above noise level during the whole measurement. The sniffer provides valuable insight of what is actually going on in the channel from a "neutral" point of view. Neutral in the sense of:

- A clear distinction between the senders can be made according to the energy levels, since the sniffer is located between the senders and transmission gains were chosen accordingly.
- Sensing the channel is possible during the whole measurement time, because the sniffer is never sending.

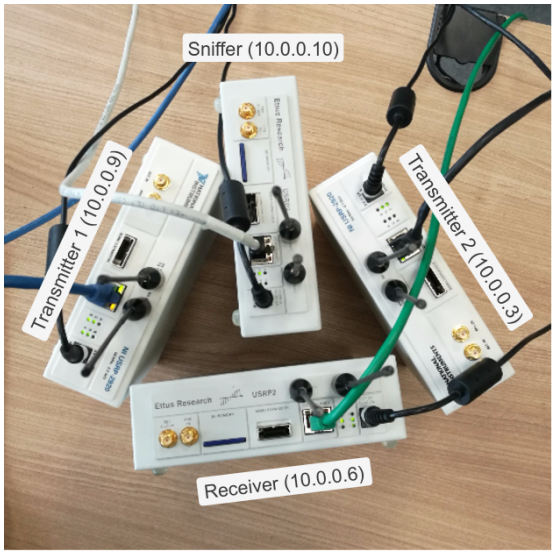


FIGURE 4.1: Photo of the measurement setup after many less successful tries due to different technical difficulties such as randomly failing devices, increased latency, asymmetrical RX/TX gains and more

Function	TX Gain	RX Gain	Source Address	Dest. Address	IP Address
Receiver	4dB	10dB	X	any	10.0.0.6
Sniffer	0	0	any	any	10.0.0.10
Transmitter 1	5dB	0	Y	X	10.0.0.9
Transmitter 2	9dB	0	Z	X	10.0.0.3

TABLE 4.1: Setup parameters

In a nutshell, the sniffer is a valuable debugging and verification tool as described in more detail in section 4.3.

4.2.2 *Pure ALOHA Transmitter*

The `run` block enables us to start several senders exactly at the same time, which is useful if we execute the flowgraphs manually without the automated measurement scripts. Payload is generated in the `dummy_source` block, packed into a frame in the `framing` block and buffered in the `frame_buffer` block. The interval between generated frames is determined a `general_timer` block, which we trigger either constant or exponentially distributed intervals. Self-reception is prevented by shutting down the receiver when about to send a frame through the sending block. As soon as the data packet is sent off the `timeout` block receives a copy of the data frame. If the timeout timer is reset by a recevied ACK before it runs out the next frame in the buffer is dequeued, otherwise the data is forwarded to the `resend_check`

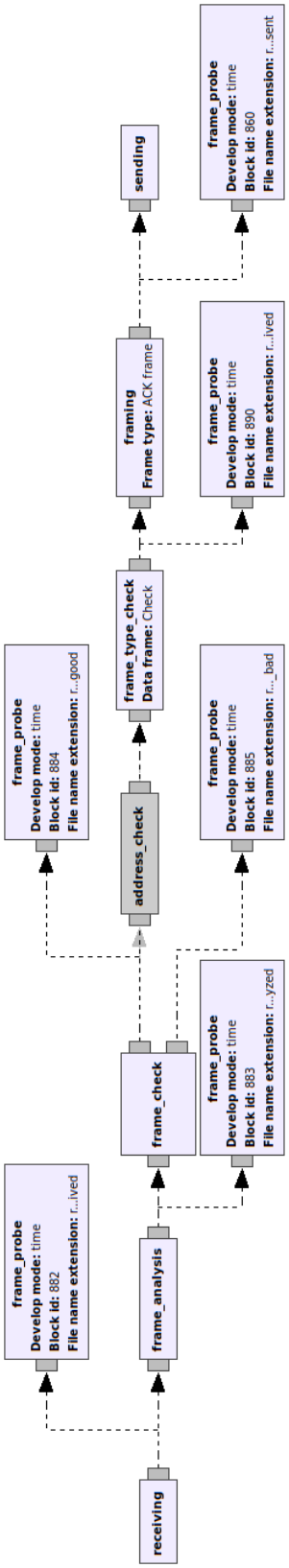
block. If the maximum number of retransmissions, in our case 6 has not been reached a retransmission is issued, otherwise the frame is dropped without substitution.

4.2.3 CSMA Transmitter

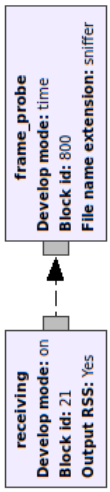
The CSMA transmitter is based on the ALOHA transmitter, but features extra mechanisms as described in section 2.1.5, which will now be exclusively discussed. The flowgraph aims at resembling 802.11 DCF and features CCA through thresholding in the `carrier_sensing` block. Despite the fact that this block has the feature of adaptively determining an appropriate carrier sensing threshold we chose a fixed value of 0.002 power units (PU) [1]. This choice was made to make sure that ALOHA transmission power levels were not confused with noise during the adaptive CSMA noise floor detection period.

DIFS and SIFS are realized through `general_timer` blocks with the respective values. The design, as depicted, does not feature the RTS/CTS exhange.

[1] Power unit is a linear-scale unit read out via the UHD driver.



(a) Receiver



(b) Sniffer

FIGURE 4.2: GRC Receiver Flowgraphs

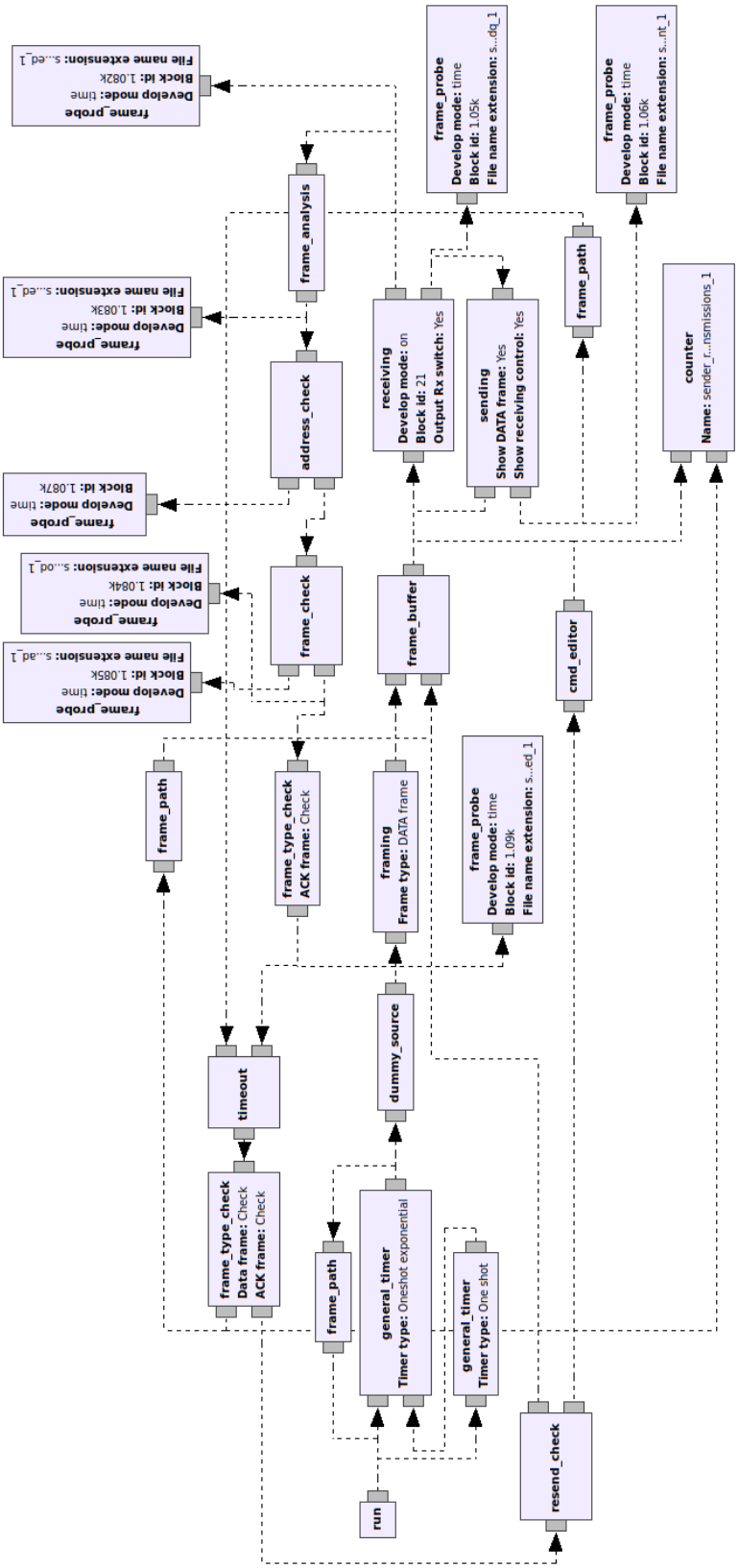


FIGURE 4.3: GRC Pure ALOHA Transmitter Flowgraph

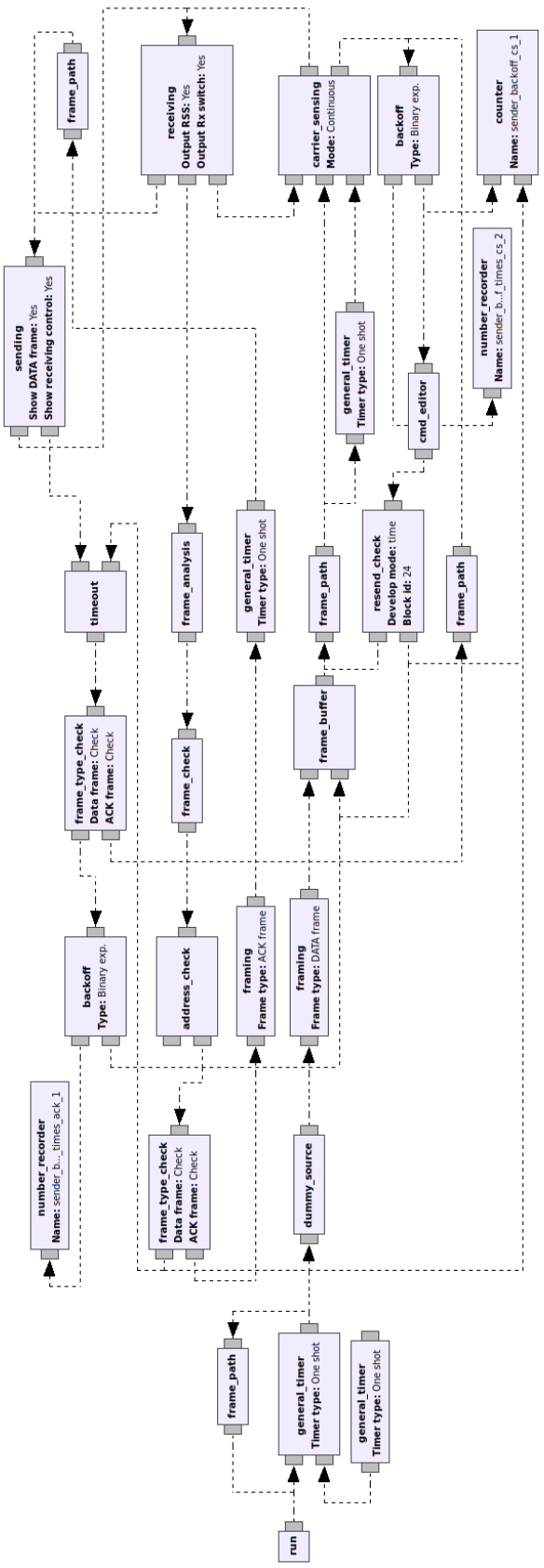


FIGURE 4.4: GRC CSMA Transmitter Flowgraph

4.3 MEASUREMENT METRICS

All recorded metrics will be defined in this section. Furthermore, we will describe the way how they were obtained and how we verified them. All metrics were originally captured with at least one of the following blocks: `frame_probe`, `counter`, `number_recorder` and `time_probe` as depicted in figures 4.2.1, 4.2.2 and 4.2.3.

4.3.1 Throughput

We define throughput as the mean useful data [2] transmission rate in the unit kbit/s. We obtain this metric simply by counting the number of ACKs received at the sender, multiply it with the frame length of 8 kbits and divide it by the measurement duration. The calculations are done in `throughput.py` making use of the CLI tool `wc` to count lines. Joint and single throughputs are our main metrics to judge a protocol's efficiency or how well a certain combination of protocols can coexist under different conditions, respectively.

4.3.2 Round-Trip Time

We define round-trip time as the mean time from the buffer dequeuing a data frame until ACK reception. If the variable `rtt_mode` in `rtt.py` is set to `rtt` then this calculation excludes retransmitted frames. As an experimental feature if `rtt_mode` is set to `frame_delay` instead then retransmitted frames are taken into account. Note, however that when the latter mode is chosen correct numbers are only guaranteed if the maximum number of retransmissions is never once reached, since there is no currently implemented way of telling if the acknowledgment belongs to the current or the next data frame in that case. We will, however, throughout the whole theis refer to the actual round-trip time according to its original definition disregarding retransmitted frames.

4.3.3 Backoff Times

The script `backoff.py` sums up three different backoff times: Firstly, the backoff due to failed [3] carrier sensing. Secondly, we capture the backoff times due to successful transmission to give other nodes a chance to seize the channel. Lastly, we sum up the two to obtain the total backoff. The total backoff duration in our view reflects the efficiency of the CSMA protocols in dependency on the parameters DIFS, SIFS and backoff slot length [4]. We chose a minimum contention window of $2^5 = 32$ backoff slots with binary exponential increase.

4.3.4 Packet Durations & Channel Occupation

The channel occupation chart as in figure **??** provides an approximated logical view on the channel in the fashion of a Gantt chart. Blue patches represent data frames,

[2] payload and headers disregarding retransmissions

[3] by failed we mean a negative CCA, i.e. a busy channel.

[4] E.g. for two CSMA transmitter it would be ideal to have both transmitters back off for around 50% of the transmission time to give each other a chance to transmit.

red patches ACKs and black patches the reception of ACKs. The chart is only a (good) approximation of the channel because the width of the patches are fixed and defined in `channel_occupation.py`. The duration of DATA and ACK frames was previously recorded with the `time_probe` blocks as the difference between the times when the frame was dequeued from the buffer and when it was received by the receiver. As expected, the frame durations were very stable. A data frame took 40ms to transmit and an ACK frame took 7ms. The variation of frame length was in the range of 1-2 milliseconds for data frames and in the sub-millisecond range for ACK frames. Depending on the time-limits chosen for the plot this may be well below the plot's resolution, which is why the time axis of such plot will be limited to a range of a few seconds at most.

4.3.5 Channel Energy Level

The energy levels (denoted in a linear-scale, non-negative power unit) in the channel observed by the sniffer (and processed in `sniffer.py`) help us to verify a multitude of metrics. We can verify frame durations, round-trip time, backoff time (for saturated traffic) and logical channel occupation. Even collisions are clearly visible and which sender caused them. Throughput ratio among senders can easily be verified by opting in data representation as CDF, e.g. if two identical MAC protocols run under identical circumstances then we expect a CDF with a step roughly at $y = 0.5$.

4.4 MEASUREMENT SCRIPT SYSTEM

The elaborate script-system was an integral part of the work and enables future users to much more quickly gain results based on automation, since it is no longer necessary to manually execute flowgraphs, manage captured files, run data processing scripts, sync files with github ... Instead everything is automatically done for them. Literally the transmission of every frame and data processing step can be traced back with the log files. Furthermore, to accommodate the need for comparison a retrospective evaluation of any set of measurements `belated_evaluation.py` was created. The user only needs to add a few lines to the script as shown in listing 4.1.

```
1 measurement     = [714,715,728,646]
2 links           = [1,2,1,2]
3 boxplot_xticks  = [
4     "CSMA\nDIFS=15ms\nSIFS=0ms\nBO=0ms\nLink 1 @ 450MHz",
5     r"unsaturated ALOHA ~ Poisson($1/\lambda=200ms$), Link 2 @
      450MHz",
6     "CSMA\nDIFS=15ms\nSIFS=0ms\nBO=0ms\nLink 1 @ 450MHz\n Baseline",
7     "unsaturated ALOHA\nLink 2 @ 450MHz\n Baseline",
8 ]
```

LISTING 4.1: Evaluation of measurements with `belated_evaluation.py`. In `links` we denote the link we used in the corresponding measurement (compare figure 4.1)

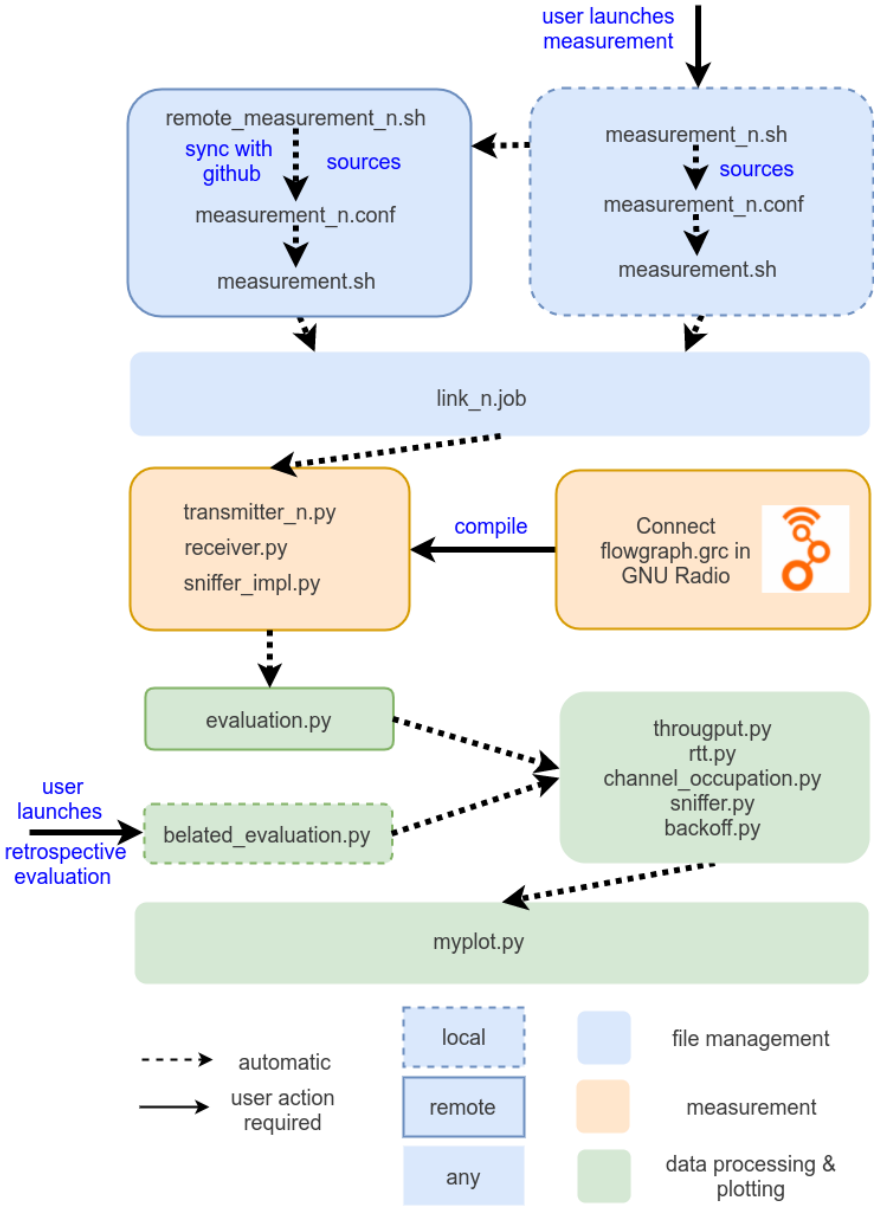


FIGURE 4.5: The three-phase measurement script system. Starting with the user calling `measurement_n.sh`, where n is the ID of the link, general settings are "imported" from `measurement_n.conf`. If the user sets `remote_measurement` to 1 in the conf file then `remote_measurement_n.sh` synchronizes the files on the remote machine with the github repository, then executes `measurement_n.sh` remotely. Subsequently, `measurement.sh` for link n works on open "jobs" from the `jobs_open_n` repositories and optionally puts them into the `jobs_done_n` directory after completion. In the job files important variables such as duration, repetitions and flowgraph scripts of the measurement are defined. Any variable set in the conf file can be overwritten in the job file since they both are just exporting variables and were separated for semantic reasons only. After the measurement concluded `evaluation.py` coordinates data processing which eventually leads to plotting based on Matplotlib as defined in `myplot.py`.

5

MEASUREMENT RESULTS

For this chapter different combinations of two links employing pure ALOHA, non-persistent CSMA and 1-persitent-CSMA-like MAC protocols were run on the same channel. In-depth analysis of different metrics and a consecutive protocol assessment are provided.

5.1 SAME PROTOCOL COMBINATIONS

5.1.1 Pure ALOHA

5.1.2 Non-Persistent CSMA With Large Parameter Set

5.1.3 Non-Persistent CSMA With Small Parameter Set

5.1.4 Non-Persistent CSMA With Medium Parameter Set

5.1.5 DIFS-Only

5.2 DIFFERENT PROTOCOL COMBINATIONS

5.2.1 ALOHA and Non-Persistent CSMA

5.2.2 Unsaturated ALOHA and Non-Persistent CSMA

5.2.3 Inhomogeneous Non-Persistent CSMA

5.2.4 DIFS-Only and ALOHA

5.2.5 DIFS-Only and Non-Persistent CSMA

6

Conclusions and Future Work

Conclusions and Future Work here.

A

Bash and Python Scripts

This appendix aims at giving in-depth insight in the scripts used in the three phases of the measuring, data processing and plotting process. The basic principle, however, is depicted in section 4.4. Minor edits were made for format and aesthetic reasons.

```
1  echo "remote_measurement is set to "$remote_measurement"."
2
3  function setup_remote_connection
4  {
5    reset
6    sshpass -p "inets" ssh -$remote_flags $remote_user@$remote_ip
       "bash -s" < remote_measurement_$link.sh
7  }
8
9  function prepare_measurement
10 {
11     reset
12     measurement_counter=0
13     ## let's make sure all the directories exist
14     printf "\nchecking if paths exists...\n"
15
16     #let's first make absolutely sure the raw data source path
       exists
17     if [ ! -d $raw_data_source_path ];
18       then
19         mkdir -p $raw_data_source_path
20         echo $raw_data_source_path" created."
21       else
22         rm -r $raw_data_source_path/*
23     fi
24
25     if [ -d $plot_directory_path ];
26       then
27         echo $plot_directory_path" already existed!"
28         cd $plot_directory_path
29         # create measurement directory
30         while [ -d $measurement_counter ]; do
31             measurement_counter=$(($measurement_counter+1))
32         done
33         export measurement_counter;
34     fi
35
36     if [ -d $log_path ];
```

```
37        then
38          echo $log_path" already existed!"
39        else
40          mkdir -p $log_path
41          echo $log_path" directory created."
42      fi
43
44      mkdir -p $plot_directory_path/$measurement_counter
45      echo $plot_directory_path/$measurement_counter" directory
        created."
46
47      mkdir -p $data_source_path/$measurement_counter
48      echo $data_source_path/$measurement_counter" directory created."
49
50      mkdir -p $jobs_open_path
51      mkdir -p $jobs_done_path
52
53      ## let's check if measurement script is defined
54      # if $measurement_scripts undefined:
55      # go through directory and list all python files
56      if [ -z ${measurement_scripts+x} ];
57        then
58          echo  "no measurement scripts set,
59                going through files inside of $locate_base_path."
60          echo "please add a the full path of one of the files to
        \$scritps."
61          #locate -r "$locate_base_path" | grep "\.py$"
62          echo "terminated. ding dong"
63          exit -1
64      fi
65
66      printf "\n"
67  }
68
69  function measure
70  {
71    local prematurely_aborted=0
72
73    for ((x = 1 ; x <= $measurement_repetitions ; x += 1)); do
74
75      # get pid to later kill it
76      for i in "${measurement_scripts[@]}"
77      do
78        python $measurement_script_path/$i &
79      done
80
81      for ((y = $timer ; y > 0 ; y -= 1)); do
82        echo "measurement $x/$measurement_repetitions complete in $y
        second(s)."
83        if [ $check_if_prematurely_aborted -eq 1 ];
84          then
85            if $(ps -p ${measurement_scripts_pid[*]}) | grep
        ${measurement_scripts_pid[*]};
```

```
86            then
87              :
88            else
89              prematurely_aborted=1
90              echo  "Scripts were killed prematurely. Measurement
    may be incomplete."
91              break
92          fi
93      fi
94      sleep 1
95    done
96
97    kill $(jobs -p)
98
99    # save this measurement's data to special folder
100   mkdir -p $data_source_path/$measurement_counter/$x
101   echo  "measurement $x raw data directory created
    $data_source_path/$measurement_counter/$x/."
102
103   echo $raw_data_source_path
104   echo $(ls $raw_data_source_path | egrep "*_$link.txt")
105
106   cd $raw_data_source_path
107   mv -v $(ls | egrep "*_$link.txt")
    $data_source_path/$measurement_counter/$x/
108   cp -v $(ls | egrep "sniffer")
    $data_source_path/$measurement_counter/$x/
109   if [ "$receiver_mode" == "single" ];
110     then
111       cp -v $(ls | egrep "receiver")
    $data_source_path/$measurement_counter/$x/
112   fi
113   echo  "measurement $x raw data moved to
    $data_source_path/$measurement_counter/$x/."
114   printf "\n"
115   if [ $prematurely_aborted -eq 1 ];
116     then
117       if [ $plot_if_prematurely_aborted -eq 0 ];
118         then
119           echo "plotting if measurement prematurely aborted set
    to false."
120           echo "terminated."
121           exit -1
122       fi
123   fi
124
125 done
126
127 #exit remote connection
128 if [ $remote_measurement -eq 1 ]; then
129   echo "remote_measurement is set to "$remote_measurement"."
130   exit
131 fi
```

```
132 }
133
134 function plot
135 {
136   ## plot the results
137   echo "now processing results..."
138
139   # call the plotting scripts as data
140   #echo "starting to generate plots..."
141   echo "plotting python should be: "$plot_py" ("$os")."
142
143   for i in ${plot_scripts[@]}; do
144     bash -c "$plot_py $plot_py_path/$i"
145   done
146
147   echo "+------------------+"
148   echo "|plotting completed|"
149   echo "+------------------+"
150 }
151
152 function cleanup
153 {
154     ##cleaning up the mess you created!
155     #kill all child proceesses
156     echo "staring cleanup..."
157     echo "killing all lingering child processes..."
158     killall -9 -g $0
159     cd $this_path
160     exit
161 }
162 trap cleanup sighup sigint sigkill;
163 trap "cd $this_path" exit;
164
165 function main
166 {
167   # clear up console
168   #reset
169   # check if jobs_open directory is empty
170   if [ ! "$(ls -a $jobs_open_path)" ]; then
171     echo "there seem to be no open jobs. measuring with default
      parameters."
172     prepare_measurement
173     #take measurements
174     measure | tee -a $log_path/default_$measurement_counter.log
175     # create plot if desired
176     if [ $plot_enabled -eq 1 ]; then
177       plot | tee -a $log_path/default_$measurement_counter.log; fi
178   else
179     prepare_measurement
180     echo "open jobs detected! let's get to work..."
181     jobs=$jobs_open_path/*
182     for job in $jobs; do
183       source $job;
```

```
184       job_name=$(echo $job | rev | cut -d"/" -f1 | rev )
185       log=$log_path/$job_name"_"$measurement_counter.log
186       #echo $job_name
187       cat $job | tee -a $log
188       cat measurement_$link.conf | tee -a $log
189       measure | tee -a $log
190       if [ $plot_enabled -eq 1 ]; then
191         plot | tee -a $log
192       fi
193       if [ $move_after_job_done -eq 1 ]; then
194         cp $job $plot_directory_path/$measurement_counter/
195         mv $job $jobs_done_path/
196       fi
197       export measurement_counter=$((measurement_counter++))
198     done
199   fi
200 }
201
202 if [ $debug_mode -eq 1 ]; then
203   echo "+-----------------+"
204   echo "|debug mode active|"
205   echo "+-----------------+"
206 fi
207
208 if [ $remote_measurement -eq 1 ]; then
209   # call to main included here
210     setup_remote_connection
211   else
212     main
213 fi
```

LISTING A.1: measure.sh

```
1  import numpy as np
2  import myplot
3  import os
4
5  import rtt
6  import throughput as tp
7  import channel_occupation
8  import backoff
9  import sniffer
10
11 # From Bash
12 measurement = [int(os.environ["measurement_counter"])]
13 links = [int(os.environ["link"])]
14 repetitions = int(os.environ["measurement_repetitions"])
15 data_source_path = os.environ["data_source_path"]
16 plot_path =
       os.environ["plot_directory_path"]+"/"+os.environ["measurement_counter"]+"/"
17 plot_type = ["cdf", "boxplot"]
```

```
throughput_data_files =
    os.environ["throughput_data_files"].split(",")
rtt_data_files = os.environ["rtt_data_files"].split(",")
co_data_files = os.environ["co_data_files"].split(",")
sniffer_data_files = os.environ["sniffer_data_files"].split(",")
retxs_data_files = os.environ["retxs_data_files"].split(",")
show_plot = int(os.environ["show_plot_after_measurement"])
rtt_mode = os.environ["rtt_mode"]
max_retxs = 6
eval_mode = "live"
timer = int(os.environ["timer"])
receiver_mode = os.environ["receiver_mode"]

#From Python
plot_pdf = False
boxplot_xticks = [ "measurement "+str(index) for index in
    measurement ]
legend_labels = [ tick.replace("\n", ", ") for tick in
    boxplot_xticks ]

custom_legend_coordinates = {
    "rtt":                  [0.24,0.85,"upper left"],
    "packet_loss":          [1,0,"lower right"],
    "retxs":                [1,0,"lower right"],
    "throughput":           [1,0,"lower right"],
    "diagnosis_sender":     [1,0,"lower right"],
    "diagnosis_receiver":   [1,0,"lower right"],
    "backoff":              [1,0,"lower right"],
    "channel_occupation":   [1,0,"lower right"],
    "sniffer":              [1,0,"lower right"]
}

create_plots = {
    "rtt":                   False,
    "packet_loss":           False,
    "retxs":                 False,
    "throughput":            True,
    "diagnostic":            True,
    "backoff":               True,
    "channel_occupation":    True,
    "sniffer":               True
}

channel_occupation_mode = {
    "occupation_mode":   ["overview", "zoom"],
    "zoom":              [5,7],
    "zoom_mode":         "interval",
    "zoom_interval":     2
}

sniffer_settings =  {
    "sniffer_mode":             ["physical", "smoothed"],
    "link":                     1,
```

```
    "zoom":                     [0.0,timer*repetitions],
    "zoom_mode":                "interval",
    "zoom_interval":            2,
    "smoothing_difference":     0.0001,
    "smoothing_derivative":     0.01,
    "smoothing_range":          [0.0010,0.0013]
}

#Unimplemented, use later
annotations_below   = []
annotations_other   = []

eval_dict = {
    "measurement":              measurement,
    "repetitions":              repetitions,
    "data_source_path":         data_source_path,
    "xticks":                   boxplot_xticks,
    "legend":                   legend_labels,
    "annotations_below":        annotations_below,
    "annotations_other":        annotations_other,
    "throughput_data_files":    throughput_data_files,
    "retxs_data_files":         retxs_data_files,
    "rtt_data_files":           rtt_data_files,
    "show_plot":                show_plot,
    "legend_coordinates":       custom_legend_coordinates,
    "create_plots":             create_plots,
    "links":                    links,
    "rtt_mode":                 rtt_mode,
    "channel_occupation_mode":  channel_occupation_mode,
    "co_data_files":            co_data_files,
    "sniffer_data_files":       sniffer_data_files,
    "sniffer_settings":         sniffer_settings,
    "timer":                    timer,
    "plot_pdf":                 plot_pdf
}

for index,a_plot_type in enumerate(plot_type):
    if plot_type[index] == "cdf":
        grid                = True
    else:
        grid                = True

    eval_dict["plot_type"]  = [plot_type[index]]
    eval_dict["plot_path"]  = plot_path
    eval_dict["grid"]       = grid

    if create_plots["backoff"] == True:
        print("Creating backoff plot!")
        backoff.backoff(**eval_dict).plot()
    if create_plots["rtt"] == True:
        print("Creating rtt plot!")
        rtt.rtt(**eval_dict).plot()
    if create_plots["throughput"] == True:
```

```
121         print("Creating throughput plot!")
122         tp.tp(**eval_dict).plot()
123 
124 # The plots with only one plot type!
125 if create_plots["channel_occupation"] == True:
126     print("Creating channel occupation plot!")
127     channel_occupation.channel_occupation(**eval_dict)
128 if create_plots["sniffer"] == True:
129     print("Creating sniffer energy plot!")
130     sniffer.sniffer(**eval_dict)
131 
132 print("Done.")
```

LISTING A.2: evaluation.py

B

ABBREVIATIONS

AM amplitude modulation

BMAC Berkeley MAC

CCA clear channel assessment

CDMA code division multiple access

CDF cumulative distribution function

CLI command line interface

CSMA carrier sense multiple access

CSMA/CA CSMA with collision avoidance

CSMA/CD CSMA with collision detection

CTS clear to send

DCF distributed coordination function

DIFS DCF interframe spacing

DIY do it yourself

DOS denial of service

DSP digitial speech processor

EIFS extended interframe spacing

FDMA Frequency Division Multiple Access

FM frequency modulation

FPGA field programmable gate array

GNU GNU is not Unix

GR GNU Radio

GRC GNU Radio Companion

GUI graphical user interface

LabVIEW Laboratory Virtual Instrumentation Engineering Workbench

LAN local area network

LLC logical link control

LBT listen before talk

MAC medium access control

MATLAB Matrix Laboratory

NAV network allocation vector

PCF point coordination function

PDU packet data unit

PHY physical (layer)

PIFS PCF interframe spacing

PU power units

PMT polymorphic type

QPSK quadrature phase-shift keying

RF radio frequency

RTS request to send

RTT round-trip time

RX receiving/reception

SDK software development kit

SMAC Sensor MAC

SDR software defined radio

SDU service data unit

SIFS short interframe spacing

SWIG simplified wrapper and interface generator

TDMA time division multiple access

TMAC Timeout MAC

TX transmitting/transmission

UHD USRP hardware driver

USRP universal software radio peripheral

WLAN wireless LAN

WSN wireless sensor networks

BIBLIOGRAPHY

[1] V. Garg, *Wireless Communications & Networking*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] A. Bachir, M. Dohler, T. Watteyne, and K. K. Leung, "Mac essentials for wireless sensor networks," *IEEE Communications Surveys Tutorials*, vol. 12, no. 2, pp. 222–248, Second 2010.

[3] A. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall Professional Technical Reference, 2002.

[4] M. S. Gast, *802.11 Wireless Networks: The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2005.

[5] I. Demirkol, C. Ersoy, and F. Alagoz, "Mac protocols for wireless sensor networks: a survey," *IEEE Communications Magazine*, vol. 44, no. 4, pp. 115–121, April 2006.

[6] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 95–107. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031508

[7] "Usrp support package from communications system toolbox," https://de.mathworks.com/hardware-support/usrp.html, accessed: 2017-10-06.

[8] "Tutorials core concepts," https://wiki.gnuradio.org/index.php/TutorialsCoreConcepts, accessed: 2017-10-03.

[9] "Gnu radio manual and c++ api reference 3.7.10.1," https://gnuradio.org/doc/-doxygen, accessed: 2017-10-04.

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift