Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA TELEKOMUNIKACJI

PRACA DYPLOMOWA

An IEEE 802.11 Channel Access Simulator based on the Python SimPy Library

Implementacja symulatora dostępu do kanału radiowego sieci IEEE 802.11 z użyciem biblioteki SimPy języka Python

Autor:	Paweł Topór
Kierunek studiów:	Teleinformatyka
Typ studiów:	Stacjonarne
Opiekun pracy:	dr hab. inż. Szymon Szott

Kraków, 2021

I would like to thank my supervisor, PhD Szymon Szott for great support and a lot of patience during reading and improving this thesis.

Contents

List of Figures

List of Listings

List of Tables

1. Introduction

Nowadays wireless communication is the most commonly used method of communication. It is present in the day to day life of the majority of world population. Especially one of the most well known types of it, which are Wireless Local Area Networks (WLANs). This field of interest is rapidly growing and still developing new solutions to ensure users best performance in data transmission. The organisation responsible for creating and maintaining standards for wireless communication in computer networks is the Institute of Electrical and Electronics Engineers (IEEE). The name of the standard which defines the methods and techniques of WLAN data transmission is 802.11. It focuses on bottom two layers of the OSI/ISO model. Those layers are medium access control (MAC) and physical layer (PHY). The first one is responsible for controlling channel access and sending data.

The motivation for this thesis is the fact that the most valuable parameters for users in the wireless networks is the highest possible data throughput. They want to be able to upload and download any data without waiting too long. The purpose of this thesis is to create easy to use simulation model which easily generate results for given transmission parameters. This model should allow users for an easy way to predict behavior of their configurations and for researchers to study 802.11 performance.

The goal of this thesis is to implement and validate a practical model of IEEE 802.11a standard. The model has to be able to simulate the channel behavior depending on the set of initial channel access parameters and number of the transmitting stations. This thesis will cover the implementation of the 802.11a [1] extension with the discrete event simulator Python [2] library, SimPy [3]. The network throughput and collision probability are two main values which are validated to prove the correct implementation mentioned extension.

The work contains five chapters. Chapter 2 contains the theoretical background which is needed to understand certain behaviours and mechanisms for the channel access in the 802.11a standard as well as description of four other implementations which were used to validate the results of this thesis. The third chapter focuses on explaining the whole implementation created for this thesis, the structure and execution process. There is also information about the technical aspects (about chosen programming language and libraries used). In the next chapter the reader can find the validation process which carefully validates the implementation against all the possible metrics and describes the conclusions. In the last chapter there is a summary about whether the main goal was reached and the main findings of the work.

2. Background

The standard 802.11 was built over the IEEE 802 set of local area network (LAN) protocols and extends it with other ones. It focuses on media access control (MAC) and physical layer (PHY) which implements a wireless local area network (WLAN) in various frequencies, including 2.4 GHz, 5 GHz, 6 GHz, and 60 GHz frequency bands with different channel widths. The first 802.11 standard was released in 1997 which was just a beginning of evolution of wireless short distance communication as every couple of years we can see new amendments to the standard. Every new standard wanted to implement something new or to improve the existing solutions with better throughput, unreliability, mobility or security. In this work the main focus is on one of the first standard which is 802.11a, especially on the DCF function used by medium access control layer.

2.1. Channel Access in IEEE 802.11

IEEE uses electromagnetic waves to transport data from one place to another. This technique is very unreliable and needs other mechanisms to control access to shared channel which allows only to handle one communication at once. To properly complete a transmission there is need of mechanism which will coordinate access to channel as this medium allows only one communication at once. For this is responsible the Medium Access Control sub-layer which supports communication of multiple stations in the shared medium. There are few techniques to archive that but we will focus on most widely used which is Distributed Coordination Function (DCF) which is based on Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). The name Carrier Sense Multiple Access with Collision Avoidance suggests how this mechanism works. In the first place it senses the channel checking if is in the idle state. Multiple access means that there is one common channel accessible by multiple stations. And collision avoidance is responsible for decreasing the probability of collisions. The basic concept of this function is shown in Figure 2.1.

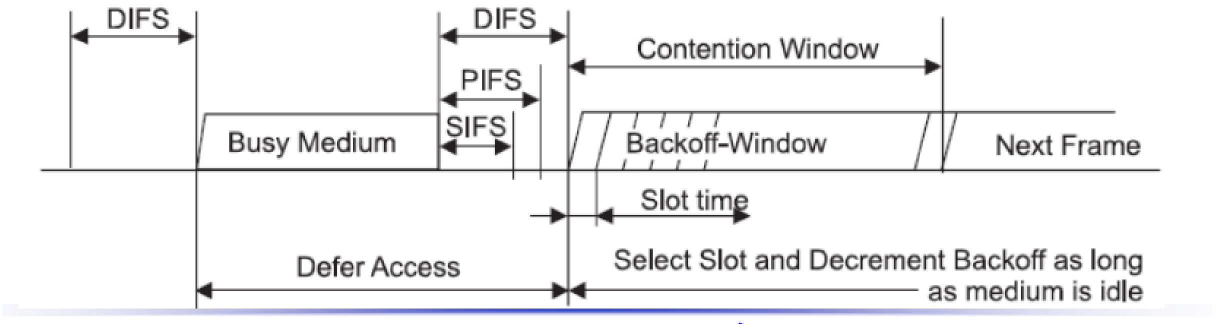


Figure 2.1: DCF function in 802.11a[4]

In the mechanism presented by Figure 2.1 we observe that it uses different waiting periods. The Slot Time defines the duration of the one slot in μs and depends on the physical layer characteristics. SIFS stands for short inter frame spacing and is used for the frames with the highest priority like ACK. It is the shortest frame spacing with length and the duration of that spacing is defined by the physical layer and calculated based on the propagation time, switching time between transmitting and receiving state and other transmitter delays. The DIFS is DCF inter frame spacing responsible for the frames with the lowest priority and due to that fact its duration is the longest and is used before sending for example date frames. Its time is the sum of the SIFS and two time slot times. The backoff-window is the random time during which the station needs to sense if the channel is idle. It is calculated from drawn random value from range $0 \; to \; 2^n - 1$, where the n is the number of failed transmissions in a row. The upper limit of the range is set to CWmax if the calculated limit would be greater then this parameter.

Depending on which type of frame has to be sent (data frame or control frame), the control rate or date rate is chosen along with SIFS or DIFS waiting period. Then it senses the channel for time defined by those frames. If the channel was idle for the whole time, DCF can start waiting backoff time. If it was not it has to start waiting chosen duration again. The backoff time is drawn from calculated range. The more failed transmissions during last subsequent attempts the bigger is the range. Then it starts counting down the drawn time as well as sensing the channel state. If the channel becomes busy it saves the left time and waits for idle medium. After counting down has completed the function can start transmitting the frame. After successful transmission the destination station should return an acknowledgement (ACK) frame. If the the frame was delivered successfully the station can prepare another frame. If the station did not receive the ACK within an ACK Timeout duration it tries the transmission again for the same frame.

Every time the frame was not sent successfully its backoff range from which the value can be drawn increases. This leads to situation when the frame with a lot of failed transmissions has lower chance to be sent before other stations. To prevent such situation the retry limit parameter was introduced. It counts the unsuccessful transmissions of the frame and when the limit was reached the frame is dropped. This prevents from blocking the station from sending another waiting frame in the queue. The missing frame has to be handled by the higher layer of the OSI/ISO stack.

2.2. Discrete Event Simulation

The implementation of the channel access method described above in this thesis relies on the discrete event simulation approach implemented in the SimPy library. To properly understand this simulation approach there is need to explain what exactly the discrete events are. First an event is an action which leads to the system state change. In this situation the event can be understood for example as packet arrival or departure, the backoff waiting started or finished. As all these actions can be completed on their own, specific time they can be called discrete. Because the system changes its state only when an event occurs, the time between these events can just be skipped and the focus (event processing) can be put only on specific moments which leads to speeding up and simplification of the whole simulation.

A simple workflow of the DCF function is illustrated in Figure 2.2 and described in Table 2.1 .

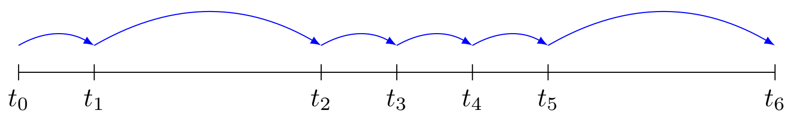


Figure 2.2: The concept of skipping between discrete events

Time of event	Event name	Event description
t_1	Frame generated	Frame generated and starts to wait difs time + backoff
t_2	Frame sent	Backoff was decremented successfully and frame was sent
t_3	Frame arrived	Frame arrived and target station start to wait sifs time before ACK frame
t_4	ACK sent	Target station waited sifs time and ACK frame was sent
t_5	ACK received	ACK received, new frame generated and difs time + backoff starts
t_6	Frame sent	Backoff was waited successfully and frame was sent
$*$	$*$	$*$
$*$	$*$	$*$
t_n	$*$	$*$

Table 2.1: Example event definitions

2.3. Existing Implementations

To validate the results of DCF-SimPy, four other implementations were considered. Each of them implements DCF using a different approach and focuses on different aspects. These implementations are already known to be working properly so it is a good way to validate the implementation of the algorithm described in this work.

2.3.1. ns-3

The ns-3 [5] discrete-event network simulator allows to easily implement any IP based network model like Wi-Fi. The simulation code which fully implements the behavior of an 802.11a network with a full stack of protocols was provided by the thesis Supervisor. The code can simulate the behavior of the network based on the values of input parameters like CW, number of stations, and simulation duration. It provides information about collision probability and throughput in the simulated network which meets the requirements to validate DCF-SimPy. The code was ran using two releases of ns-3: 3.30.1 and 3.31 (the latter was released during the writing of this theses). The code is attached in Appendix **??**.

2.3.2. Matlab Throughput

This model implementation was written by Kamil Słowik in Matlab [6]. It covers the whole process of constructing frames and their duration as governed by the 802.11a protocol. It defines the wrapping of the raw payload into the PPDU frame by adding every additional information needed by each of the layers (PHY and MAC). Besides that is defines the duration of frames like SIFS, DIFS, ACK, and PPDU depending on the data and control transmission rate. This allows calculating the theoretical maximum throughput of a network with only one station transmitting and thus to validate the DCF-SimPy throughput implementation in the most simple scenario (one transmitting station in the channel). The code is attached in Appendix **??**.

2.3.3. Matlab Wi-Fi Simulator

It is the Matlab implementation of the simulator for the Wi-Fi behavior regarding to the initial parameters of the transmitting stations. The code covers the implementation of the DCF function used in the 802.11 standard. The implementations allow to modify input parameters like number of stations, CWmin, CWmax, data length. As a result it returns a metric along with the ones that are helpful to validate the implementation from this thesis, collision probability and throughput. The code was provided by the thesis Supervisor, who used it in [7], and is attached in Appendix **??**.

2.3.4. Wi-Fi AirTime Calculator

The Wi-Fi AirTime Calculator Excel spreadsheet was created by Gjermund Raaen [8] to calculate the achievable throughput for 802.11a/n/ac/ax standards with one transmitting station. This sheet allows to manipulate many parameters to adjust each scenario for the mentioned standards. It is based fully on mathematical equations to calculate the result.

3. Implementation

This chapter describes the implementation of the DCF function used to simulate behavior of channel access in 802.11a networks. Everything was written in Python from scratch with the help of libraries like pandas, matplotlib, numpy, and scipy. However, the core library used to implement the whole algorithm was SimPy which is a generic framework for discrete event simulation. To validate the results, four other implementations were used which were already confirmed to be working correctly. DCF-SimPy code is attached as Appendix **??**.

The structure block representing the workflow of the DCF-SimPy implementations is shown in Figure 3.1.

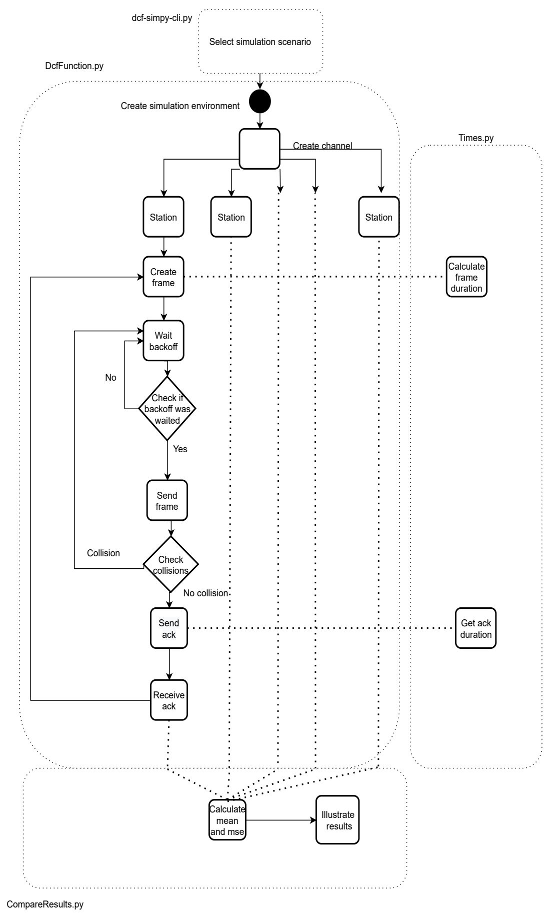


Figure 3.1: Workflow diagram

3.1. SimPy Library

The SimPy library allows for process-based discrete-event simulations. This means that the channels, stations, and frames are simulated as processes based on Python generator functions. With this it is possible to queue events and wait for them to execute in an ordered way (e.g., wait backoff time, wait for frame to arrive). Additionally, the library adds many shared resources providing a way of communication between processes, e.g., a station can notify others that it has started a transmission. The basic concept of this library can be described as follows:

1. There is an environment which is responsible for the scheduling and processing of events.
2. Inside an environment we can define independent processes responsible for generating events.
3. Events are scheduled tasks which are responsible for waiting some discreet time, accessing shared resources, and interrupting other events.
4. Communication between processes inside an environment can be done by using shared resources, the state of which is accessible at any time within this environment.

In the DCF function there is one main process which represents the channel (Listing 3.1). It uses two shared resources, simpy.Resource and simpy.PreemptiveResource. These resources are used by all stations to define the state of the channel. The first one stores all currently transmitting stations and the second one is preempted by the station with the longest frame which means that it contains one station with the longest single transmission time.

```python
@dataclass()
class Channel:
    tx_queue: simpy.PreemptiveResource  # lock for the stations with the
     ↪  longest frame to transmit
    tx_lock: simpy.Resource  # channel lock (locked when there is ongoing
     ↪  transmission)
    n_of_stations: int  # number of transmitting stations in the channel
    backoffs: Dict[int, Dict[int, int]]
    tx_list: List[Station] = field(
        default_factory=list
    )  # transmitting stations in the channel
    back_off_list: List[Station] = field(
        default_factory=list
    )  # stations in backoff phase
    failed_transmissions: int = 0  # total failed transmissions
    succeeded_transmissions: int = 0  # total succeeded transmissions
    bytes_sent: int = 0  # total bytes sent
```

Listing 3.1: Channel representation

Each of the stations is represented by its own process, which is responsible for sending frames based on the state of the shared resources. It schedules events (frames) to wait for various durations: backoff slots, SIFS, DIFS, or the frame (DATA or ACK) transmission. After the events are scheduled, the SimPy environment orders them and executes in the correct order. Listing 3.2 shows the representation of a station and Listing 3.3 the representation of a frame. Is can be seen that there are variables responsible for setting the transmission parameters. Besides that there are some variables used to store the results of the simulation which later will be saved in the csv file.

```python
class Station:
    def __init__(
        self,
        env: simpy.Environment,
        name: str,
        channel: dataclass,
        config: Config = Config(),
    ):
        self.config = config
        self.times = Times(config.data_size, config.mcs)
        self.name = name  # name of the station
        self.env = env  # current environment
        self.col = random.choice(colors)  # color of output
        self.frame_to_send = None  # the frame object which is next to send
        self.succeeded_transmissions = 0  # all succeeded transmissions for
          ↪  station
        self.failed_transmissions = 0  # all failed transmissions for
          ↪  station
        self.failed_transmissions_in_row = (
            0  # all failed transmissions for station without succeeded
              ↪  transmissions
        )
        self.cw_min = config.cw_min  # cw min parameter value
        self.cw_max = config.cw_max  # cw max parameter value
        self.channel = channel  # channel object
        env.process(self.start())  # simulation process
        self.process = None  # waiting backoff process
```

Listing 3.2: Station representation

```python
@dataclass()
class Frame:
    frame_time: int  # time of the frame
    station_name: str  # name of the owning it station
    col: str  # output color
    data_size: int  # payload size
    t_start: int  # generation time
    number_of_retransmissions: int = 0  # retransmissions count
    t_end: int = None  # sent time
    t_to_send: int = None  # how much time it took to sent successfully
```

Listing 3.3: Frame representation

3.2. Frame Transmission Time

The basis of any DCF implementation requires the proper calculation of frame transmission duration. In the 802.11a standard, when sending each frame, there is additional data overhead (coming from the PHY and MAC layers), time intervals (SIFS, DIFS, backoff), and mechanisms allowing to confirm the correctness of the transmission. Knowing the size of individual elements, we can calculate the duration of a single frame transmission, which is necessary for the correct simulation of the channel access algorithm. Table 3.1 presents the constant values of 802.11a used in the remainder of this thesis.

Parameter	Value
Slot duration	9 μs
SIFS	16 μs
DIFS	34 μs
Preamble duration	16 μs
ACK_timeout	45 μs
ACK size	18 B
MAC header	40 B
CWmin	15 slots
CWmax	1023 slots
Retry limit	7

Table 3.1: Constant PHY/MAC parameters of 802.11a.

The duration of a successful transmission, when there is no collision, can be defined as:

$$TxTime_s = DIFS + Backoff + PPDUframeTime + SIFS + ACKframeTime \tag{3.1}$$

where:

- *TxTime_s* – total duration of a single successful transmission [μs],
- *DIFS* – duration of DCF Interframe Space [μs],
- *Backoff* – random duration in the range $[0, CW] \times Slot$ [μs],

- *PPDUframeTime* – duration of PPDU frame transmission [μs],
- *SIFS* – duration of Short Interframe Space [μs],
- *ACKframeTime* – duration of Acknowledgment frame transmission [μs].

Meanwhile, the duration of a transmission when a collision occurred, can be defined as:

$$TxTime_c = DIFS + Backoff + PPDUframeTime + ACKTimeout \tag{3.2}$$

where:

- *TxTime_c* – total duration of a single failed transmission [μs]
- *ACKTimeout* – waiting time for the Acknowledgment frame [us]

To specify the PPDUframeTime we need to know the duration of three elements: Preamble, Signal and CPSDU.

$$PPDUFrameTime = PreambleTime + SignalTime + CPSDUTime \tag{3.3}$$

where:

- *SignalTime* – time of OFDM signal which depends on the physical control rate (Signal/Control Rate),
- *CPSDUTime* – time of CPSDU frame and depends on the data control rate (CPSDU/Data Rate).

The data and control rate for an 802.11a transmission depend on the modulation and coding scheme used (Table 3.2). In this work, the highest data rate (54 Mb/s) and highest control rate (24 Mb/s) are used which represents a typical configuration of 802.11a networks.

Modulation	Coding rate	Data rate (Mb/s)
BPSK	1/2	6*
BPSK	3/4	9
QPSK	1/2	12*
QPSK	3/4	18
16-QAM	1/2	24*
16-QAM	3/4	36
64-QAM	2/3	48
64-QAM	3/4	54

Table 3.2: Data rates available in 802.11a. Rates available as control rates are denoted by an asterisk.

SignalTime can be calculated by dividing the signal length (24 B) by the control rate and CPSDU is the sum of Overhead, PSDU and PAD where:

- *Overhead* – Service + Tail in CPSDU frame (22 B),
- *PSDU* – MACFrame, MACOverhead + Payload,
- *Pad* – Complement to ensure integer values of symbols.

There remains only the need to calculate the value of Pad; this can be done using the following equation:

$$Pad = ceil((Service + PSDU + Tail)/DBPS)) \cdot DBPS - (Service + PSDU + Tail) \tag{3.4}$$

where:

– *DBPS* - Bits of data per symbol

Having all this information the frame duration can be finally calculated (Listing 3.4).

```
        msdu = self.payload * 8  # [b]
        # MacFrame
        mac_frame = Times.mac_overhead + msdu  # [b]
        # PPDU Padding
        ppdu_padding = math.ceil(
            (Times._overhead + mac_frame) / self.n_data
        ) * self.n_data - (Times._overhead + mac_frame)
        # CPSDU Frame
        cpsdu = Times._overhead + mac_frame + ppdu_padding  # [b]
        # PPDU Frame
        ppdu = self.ofdm_preamble + self.ofdm_signal + cpsdu /
         ↪  self.data_rate  # [us]
        ppdu_tx_time = math.ceil(ppdu)
```

Listing 3.4: Calculation of DATA frame duration

After each correct transmission the destination station needs to send the confirmation about receiving the data correctly. To do this the ACKFrame is sent. The duration of that frame is the sum of the constant values and can be defined as follows:

$$ACKFrameTime = PreambleTime + SignalTime + AckLength/ControlRate \tag{3.5}$$

After substitution of the variables in the equation with proper values the ACKFrameTime in 802.11a with a control rate of 6 Mb/s takes always 59 μs to transmit.

3.3. Backoff Mechanism

This chapter describes the implementation off the backoff mechanism in the DCF algorithm which is responsible for deciding when the frame should be transmitted. It detects if the channel is idle and counts down the waiting period to minimise the collision probability. The first function is responsible for generating the backoff time based on the CWmin, CWmax parameters and the retry counter (the maximum number of unsuccessful transmissions in a row), Listing 3.5.

```python
    def generate_new_back_off_time(self, failed_transmissions_in_row):
        upper_limit = (
            pow(2, failed_transmissions_in_row) * (self.cw_min + 1) - 1
        )  # define the upper limit basing on  unsuccessful transmissions
         ↪  in the row
        upper_limit = (
            upper_limit if upper_limit <= self.cw_max else self.cw_max
        )  # set upper limit to CWmax if is bigger then this parameter
        back_off = random.randint(0, upper_limit)  # draw the backoff value
        self.channel.backoffs[back_off][
            self.channel.n_of_stations
        ] += 1  # store drawn value for future analyzes
        return back_off * self.times.t_slot
```

Listing 3.5: Backoff time generation implementation

First, it is necessary to define the upper backoff limit regarding the previous collision count and check if it is not larger than CWmax; if so the limit is set to the value of CWmax. After this a number is drawn from the range between 0 and upper limit using the random Python library. The drawn backoff is stored for analyzing the obtained values. Since up till now the value of backoff has been in slots it is multiplied by the slot duration to return the value in μs.

Listing 3.6. shows the implementation of the backoff mechanism. The main part of this mechanism relies mostly on simpy.Resource(env, capacity=1) which works like a lock on the channel. When there are ongoing transmissions this lock prevents other stations from starting counting backoff as the channel is not idle. When the lock becomes free all stations can start waiting their remaining backoff after an additional DIFS time. When any station finishes the waiting period it firstly informs other stations waiting their own backoff that the channel becomes busy by sending the interrupt signal (it throws exceptions in their processes) and starts the transmission process. Interrupted stations have to stop counting down and have to join the waiting queue managed by the mentioned lock. To simulate the delay of sensing the channel, the remaining backoff is reduced by one slot.

```python
    def wait_back_off(self):
        back_off_time = self.generate_new_back_off_time(
            self.failed_transmissions_in_row
        )  # generate the new backoff time
        while back_off_time > -1:
            try:
                with self.channel.tx_lock.request() as req:  # wait for the lock/idle channel
                    yield req
                back_off_time += Times.t_difs  # add DIFS time
                log(self, f"Starting to wait backoff: ({back_off_time})␣...")
                start = self.env.now  # store the current simulation time
                self.channel.back_off_list.append(
                    self
                )  # join the list off stations which are waiting backoffs
                yield self.env.timeout(
                    back_off_time
                )  # join the environment action queue
                log(self, f"Backoff waited, sending frame...")
                back_off_time = -1  # leave the loop
                self.channel.back_off_list.remove(
                    self
                )  # leave the waiting list as backoff was waited successfully
            except simpy.Interrupt:  # handle the interruptions from transmitting stations
                log(self, "Waiting was interrupted, waiting to resume backoff...")
                back_off_time -= (
                    self.env.now - start
                )  # set the backoff to the remaining one
                back_off_time -= 9  # simulate the delay of sensing the channel state
```

Listing 3.6: Backoff waiting mechanism implementation

3.4. Collision Detection

This chapter describes how collisions are detected in the DCF function algorithm which allows to decide if there is a need for retransmission or the frame was delivered successfully. When a station starts a transmission, it joins the tx_list which will allow to decide if there was a collision or not. After the station finishes transmitting it checks if there are other stations in the list. If there are, that means that all these transmissions were not successful and there is a need to retry them. Otherwise, it means that the channel was used only by one station and there was no collision. This behavior is presented by Listing 3.7.

```python
def check_collision(self):  # check if the collision occurred
    if (
        len(self.channel.tx_list) > 1
    ):  # check if there was more then one station transmitting
        self.sent_failed()
        return False
    else:
        self.sent_completed()
        return True
```

Listing 3.7: Implementation of collision detection

3.5. Obtaining and Analyzing Results

One of the most important aspects of each implementation is validating the results. Appropriate parameters must be collected, processed, and visualized. On how this is done depends how easy and accurate the correctness of the whole algorithm can be confirmed and how much it can be relied upon. To validate the correctness of this algorithm the following metrics have been collected:

– Successful transmissions - how many transmissions during one simulation were successful

– Failed transmissions - how many transmissions during one simulation were unsuccessful

– Throughput - what was the final throughput during the simulation

– Collision probability - what was the collision probability during the simulation

– Drawn backoffs - the list of drawn backoff values during the simulation

The above statistics were collected during the run time of the algorithm, summarized and saved in a comma separated (csv) file along with other settings like CWmin, CWmax, simulation time and Number of stations. Sample results are shown in Table 3.3.

Figure 3.2 shows the whole process of collecting, processing, and visualizing the results.

TIME	CW-MIN	CW-MAX	N	SEED	P-COLL	THR	FAILED-TX	SUC-TX
2020-10-20 06:52:47	15	1023	1	1	0.0000	28.050432	0	2382
2020-10-20 06:52:50	15	1023	2	1	0.1145	28.427264	312	2414
2020-10-20 06:52:51	15	1023	3	1	0.1750	28.474368	513	2418
2020-10-20 06:52:52	15	1023	4	1	0.2309	27.968	713	2375
2020-10-20 06:52:53	15	1023	5	1	0.2689	27.626496	863	2346
2020-10-20 06:52:53	15	1023	6	1	0.2967	27.249664	976	2314
2020-10-20 06:52:54	15	1023	8	1	0.3440	26.660864	1187	2264
2020-10-20 06:52:54	15	1023	7	1	0.3370	26.707968	1153	2268
2020-10-20 06:52:54	15	1023	9	1	0.3723	26.390016	1329	2241

Table 3.3: Sample results of DCF-SimPy simulation

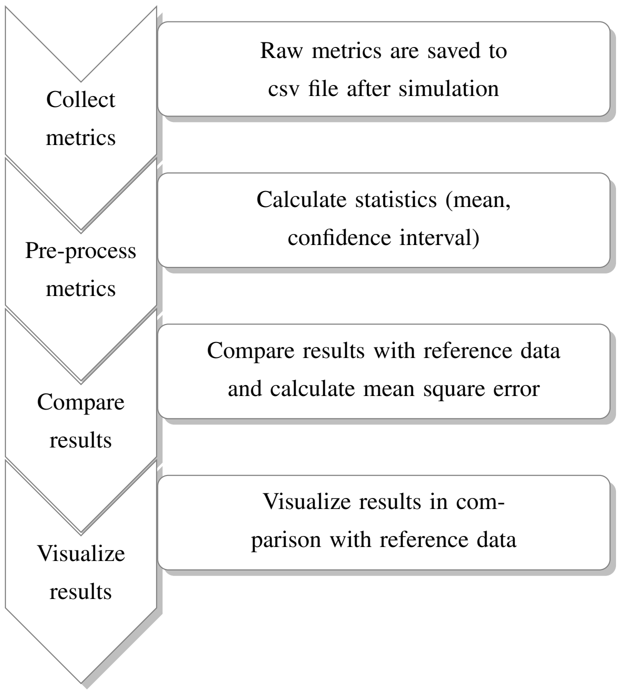

Collect metrics
Raw metrics are saved to csv file after simulation
Pre-process metrics
Calculate statistics (mean, confidence interval)
Compare results
Compare results with reference data and calculate mean square error
Visualize results
Visualize results in comparison with reference data

Figure 3.2: Process of visualizing results

From the raw results the statistics (average, standard deviation) per station are calculated and appended to the reference data. To measure the quality of results, the mean square error is generated and saved to evaluate the algorithm. This whole processing is done using the Numpy library which simplifies and speeds up the whole process. After this processing, a chart can be finally generated to show all potential imperfections of the algorithm.

4. Validation

In this chapter the DCF-SimPy implementation is going to be validated by comparing it to the implementations described in Section 3.6 which have been already proven to be working correctly. The implementation can be consider to be working properly when the collision probability and the throughput for each number of transmitting stations is almost identical to the reference data from other implementations. Besides these two parameters the initial frame duration and backoff distribution were checked as well. All the results are placed on charts to get clear view of any deviations. Where it was reasonable the mean square error (MSE) was calculated to get error as a value. It is the difference between DCF-SimPy and other implementations. All the simulation parameters were set according to standard 802.11a and are located in Table 3.1. The data payload size was set to 1472 B, so that with the IP and UDP headers, the MAC payload was exactly 1500 B. The highest possible data and control rate was used during the simulation which is respectively 54 Mb/s and 24 Mb/s.

All simulations were performed following these assumptions:

– Station always has frame to send

– No hidden stations

– No fragmentation

– Perfect channel conditions

– Simple DCF without RTS/CTS

– Only unicast transmission

– No frame aggregation

– Channel is block until the longest frame was sent

– 95% confidence intervals

4.1. Program Execution

This implementation uses "click" Python library which allows to create a command line interface for Python programs. It was used to prepare the functions to execute the DCF function in different scenarios like:

– Changing number of transmitting stations

– Changing CW parameters

– Changing MCS value

– Changing payload size

– Single simulation

Requirement for running any of the simulation is having Python >= 3.6 installed. There is also a need to install all of the additional libraries from requirements.txt file by running for example: `python install -r requirements.txt`. After that the simulation can be executed.

To see all available simulation scenarios the following command has to be executed:

```
python3 dcf-simpy-cli.py  --help

Usage: dcf-simpy-cli.py [OPTIONS] COMMAND [ARGS]...

Options:
  -v, --verbose  Enable informational logging, use second time for debugging
                 logs.

  -h, --help     Show this message and exit.

Commands:
  run-changing-cw
  run-changing-mcs
  run-changing-payload
  run-changing-stations
  single-run
```

This will print all possible scenarios with the execution instructions. Next command will show the detail information about selected scenario:

```
python3 dcf-simpy-cli.py  run-changing-stations --help
Usage: dcf-simpy-cli.py run-changing-stations [OPTIONS]

Options:
  -r, --runs INTEGER             Runs per stations number.
  --stations-start INTEGER       Starting number of stations.  [required]
  --stations-end INTEGER         Ending number of stations.  [required]
  -t, --simulation-time FLOAT    Duration of the simulation per stations number
                                 in s.

  -p, --payload-size INTEGER     Size of payload in B.
  --cw-min INTEGER               Size of cw min.
  --cw-max INTEGER               Size of cw max.
  --r-limit INTEGER              Number of failed transmissions in a row.
  --seed INTEGER                 Seed for simulation.
  -s, --skip-results             If provided, results are not saved.
  --skip-results-show            If provided, results are not shown, to show
                                 results you can't skip-results.

  -m, --mcs-value INTEGER        Value of mcs.
  -h, --help                   Show this message and exit.
```

And at the end the selected simulation can be run by using selected values:

```
python3 dcf-simpy-cli.py  run-changing-stations -r 5 \
--stations-start=2 --stations-end=4 -t 1

SEED = 0 N=2 CW_MIN = 15 CW_MAX = 1023  PCOLL: 0.1128 THR: ...
SEED = 0 N=3 CW_MIN = 15 CW_MAX = 1023  PCOLL: 0.1751 THR: ...
SEED = 0 N=4 CW_MIN = 15 CW_MAX = 1023  PCOLL: 0.2087 THR: ...
```

4.2. Frame Duration

The first step in validating DCF-SimPy is to compare the frame duration of this algorithm. Each of the required frames was described in Section 3.2. At the beginning the single data and acknowledgement frame duration was compared. The results are located in Table 4.1.

Model Name	Data duration [μs]	ACK duration[μs]
ns-3.30.1	248	28
ns-3.31	248	28
Analitical model	245	23
WiFi AirTime	244	22
DCF-SimPy	245	23

Table 4.1: Frame duration comparison

It allows to validate the implementation of PHY and MAC layer overhead in comparison to other reference values. This is the core on which later the other metrics will rely on. Data duration is similar in all cases, in ns-3 it is longer for around 3-4 μs caused by the value of OFDM preamble which in ns-3 is set to 20 μs instead of to 16 μs like in the other simulations. There is the same situation with the ACK duration.

After the frame duration is validated it is important to proceed with throughput comparison. The scenario with 1 transmitting station in channel and Backoff set to 0 (CWmin=0 and CWmax = 0) is going to be examined. By setting the Backoff parameter to 0 the randomness factor has been eliminated which means that the results should be the same. Obtained throughput results have been visualized in Figure 4.1. The simulation with one transmitting station will check only the implementation of the correct transmission frames as there are no collisions in such conditions and frames for the situation when the collision occurs are never sent. It means that only the correct frame transmissions was validated (Equation 3.1) however in the failed transmission case only SIFS+ACKframeTime changes to ACKTimeout (Equation 3.2). These are constant values which are defined by 802.11a so the that difference is predefined and can be skipped from validation.

In this section all the charts are without any confidence intervals because there is no random factor caused by backoff mechanism or frame collisions.

The DCF-SimPy results are close to the Analytical Model and Wi-Fi AirTime, a difference is caused only due to the fact of rounding error. In Analytical Model the precision is the highest and is up to 10e-12, in DCF-Simy 10e-7 and the Wi-Fi AirTime 10e-0. The lower throughput of the ns-3 simulator is caused because it implements more realistic network model which leads to decreasing of performance. All other implementations are implementing only DCF function with no other mechanisms which explains higher throughput.

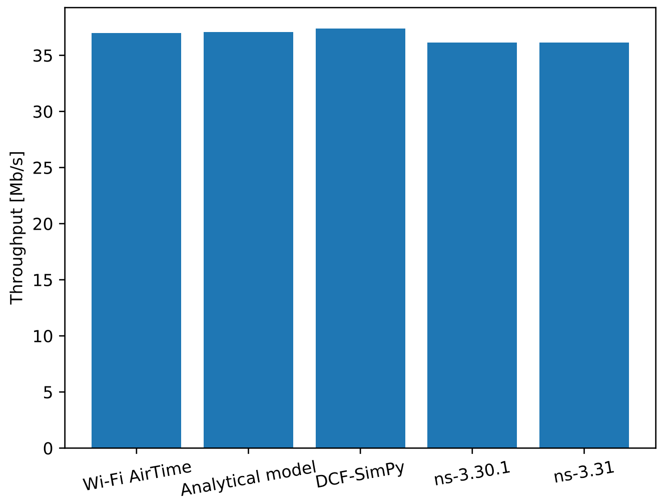


Figure 4.1: Throughput comparison

4.3. Throughput

All of the functions and mechanisms in 802.11 are mostly introduced to obtain the best performance of data transmission, which means that the main factor is to get the highest possible throughput within any number of transmitting stations. That is why the next important metric to validate is the aggregate network throughput. The results were generated for transmitting stations in the common channel from range 1 to 10. Simulation was set to 100 seconds and was retried 10 times for each number of stations. The rest of the parameters were set to the values from Table 3.1 and Table 3.2. To validate the results the ns-3 implementations were used and analytical model as only those three are able to calculate the throughput. In ns-3 and DCF-SimPy, the throughput depends on the drawn backoff and collision occurrences which is a random factor, the MSE for those was calculated for each run per station count. Analytical model produces repeatable, unchanged results each time so the confidence intervals in this case were omitted. Table 4.2 presents values of the mean square error, and it can be noticed that DCF-SimPy results are the closest to the ns-3.31 implementation which should give the results with the best quality.

Model Name	MSE for DCF-SimPy
ns-3.30.1	2.09E-01
ns-3.31	2.18E-02
Analitical model	5.90E-01

Table 4.2: MSE for DCF-SimPy of throughput

As the MSE are small the chart for Figure 4.2 was scaled in order to better show the confidence intervals and visualize the differences between the implementations. Again it can be clearly see that DCF-SimPy overlaps with the ns-3.31 simulator and the results are between ns-3.30.1 and the Analytical model. The analytical model produces higher values which is reasonable as this is the most enlarged model which uses only mathematical equations to generate results.

Figure 4.3 shows the throughput in the normal scale, the lines of the results for each of the implementations are almost overlapping each other, the overall differences in final throughput are not significant and all the lines are decreasing after reaching two stations smoothly without any deviations.

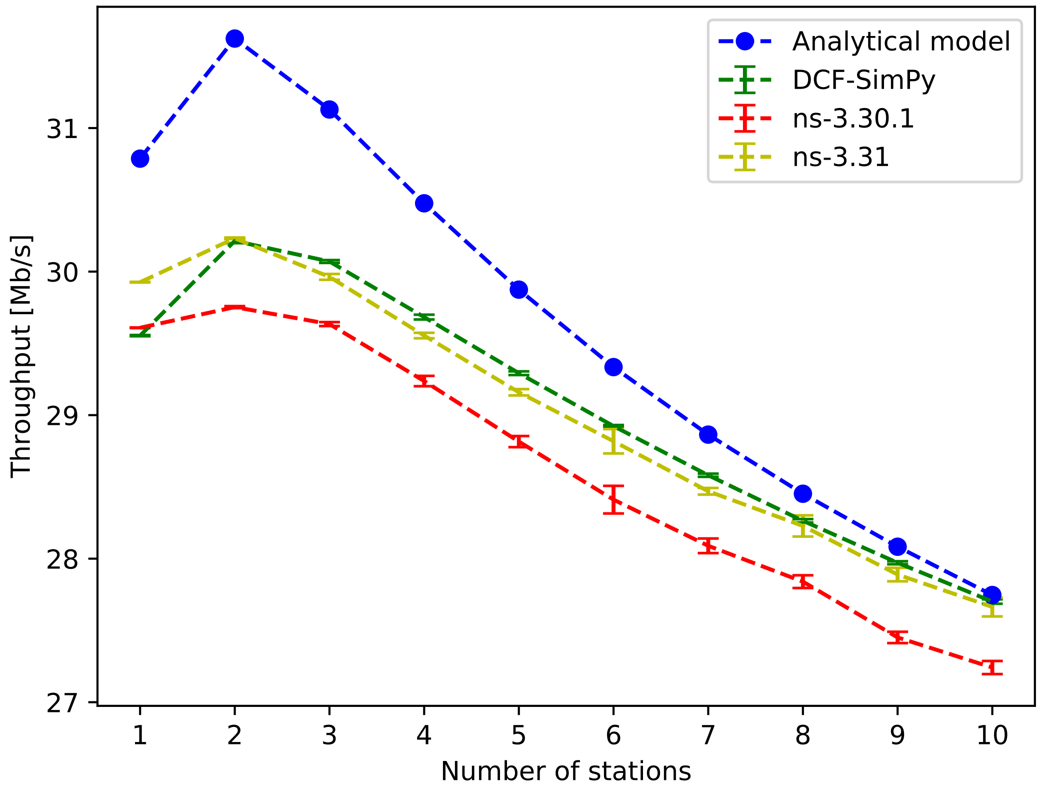


Figure 4.2: Throughput per station count comparison simplified

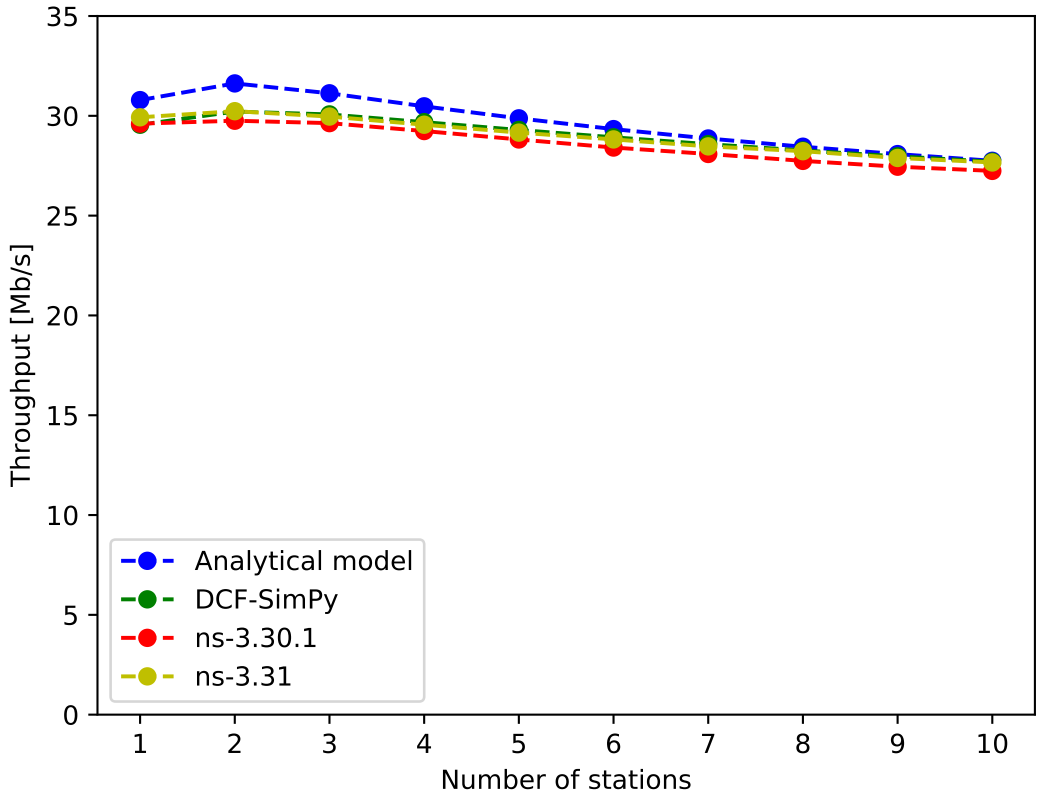


Figure 4.3: Throughput station count comparison

The quality of the obtained results is good. It can be seen that Analytical model generated the best throughput with fast decreasing values after two transmitting stations. The results of ns-3.31 are the closest ones to the DCF-SimPy which is good behavior. In ns-3.31 there was a lot of bug fixes and thanks to that the results of that version show much better performance in comparison to the previous one. ns-3.30.1 had probably some bugs because it simulated lower throughput than other implementations. It can be seen that values from ns-3.31 simulator are similar to the DCF-SimPy but with constant difference of 0.1 Mb/s. These means that the only difference between these two implementations in caused by the initial difference in frame duration.

4.4. Collision Probability

One of the most important values is collision probability. It shows how well the exponential, random back off mechanism allows to avoid collisions. With the increasing amount of transmitting stations in the common channel, the collision probability should be increasing without any unexpected deviations. To validate DCF-SimPy the results were compared with two other implementations, and one analytical model. All of those have been described in Section 3.6. In all simulations, the transmitting station count was varying from 1 to 10. The simulation time was set to 100 seconds and was executed 10 times for each station count to generate more reliable results. All other simulation parameters were set according to standard 802.11a and are located in Table 3.1. After all of the runs, the mean and the mean square error (MSE) were calculated.

According to the data from Table 4.3 which shows the mean square error of SCF-SimPy implementation in comparison to other ones, it can be noticed that this error in the worst case (ns-3.31) has only value 1.20E-04. In general, the MSE shows that the results are close to the reference data.

Model Name	MSE for DCF-SimPy
ns-3.30.1	7.56E-06
ns-3.31	1.20E-04
Analitical model	4.22E-05
Matlab simulation	7.73E-05

Table 4.3: MSE for DCF-SimPy of collision probability

Figure 4.4 visualizes the collision probability of all of the implementations. It can be seen that only ns-3.31 differs from other results including DCF-SimPy. All other simulations generate results similar to the implementation from this thesis.

Regarding to the values of MSE from Table 4.3 and Figure 4.4 the conclusions are following:

– In Analytical model for lower number of station the similar results to DCF-SimPy are generated. Starting with 6 stations and more is starts to show higher collision probability due to a fact that it does not implement retry limit mechanism which is responsible for lowering the chance of collisions when there is more transmitting stations in the channel.
– For ns-3.30.1 simulation there is almost no difference in results in comparison to DCF-SimPy.
– ns-3.31 changed something in the implementation which caused that the collision probability to decreased. It is hard to define what caused that but this is desirable behavior.

Overall the collision probability in all of the implementations is similar and without any deviations or anomaly behaviors which means that DCF-SimPy implements properly working DCF function algorithm.

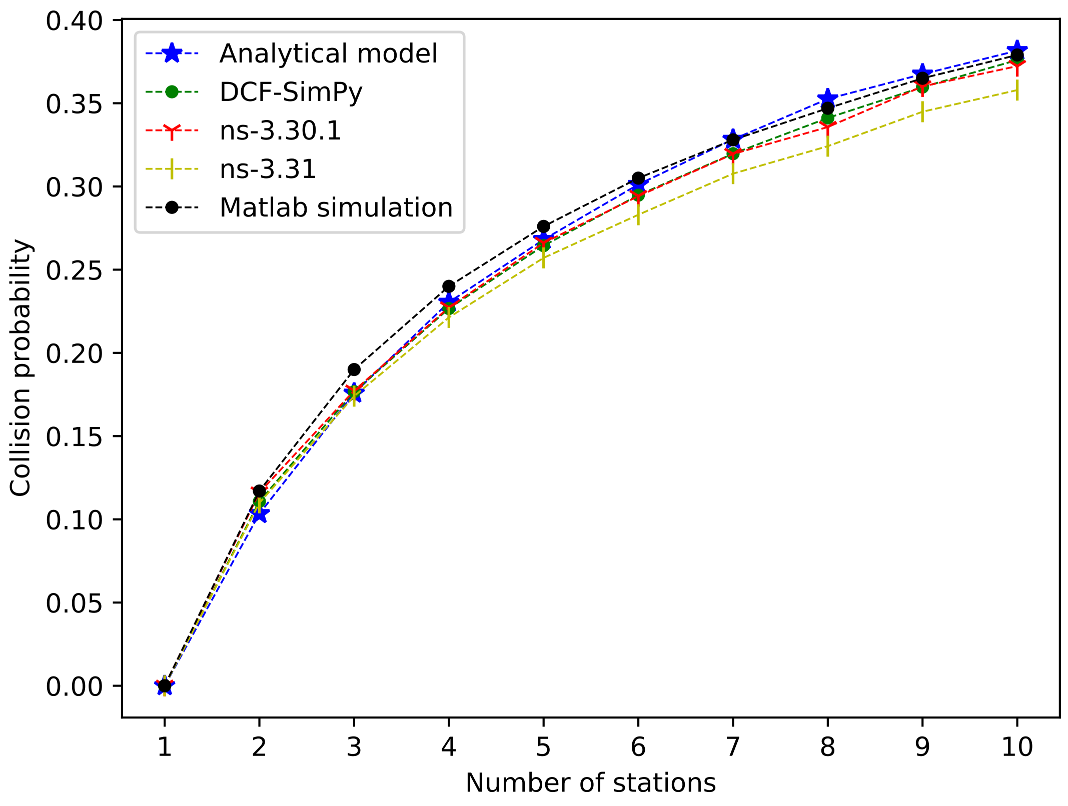
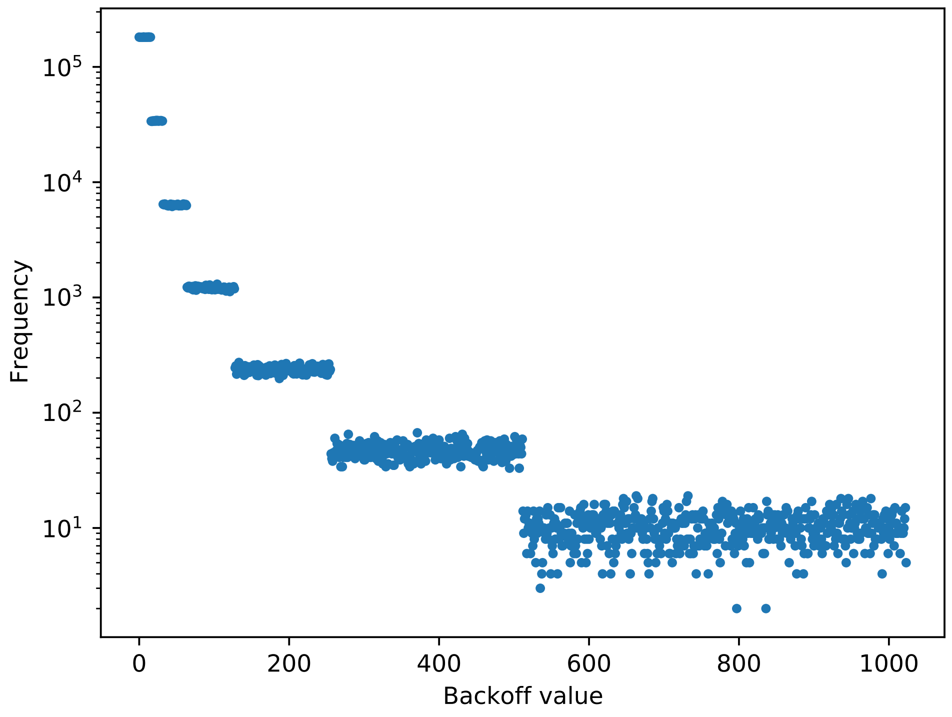


Figure 4.4: Collision probability per station count comparison

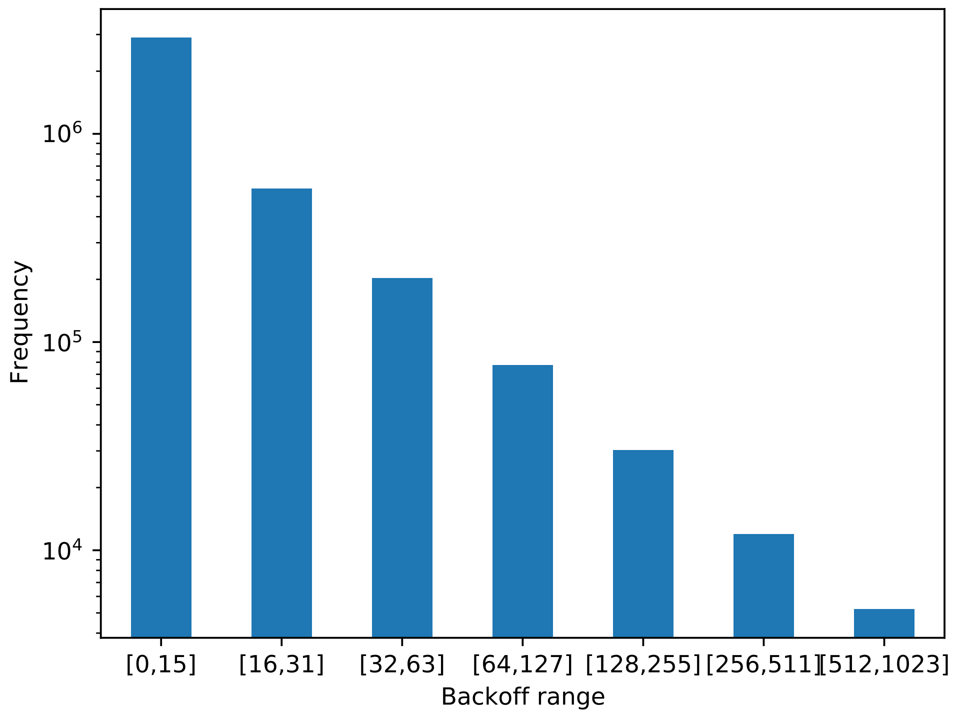
4.5. Initial CW Size

This section will be responsible for visualizing the backoff distribution during the simulation. It will not be compared to other implementations as this is more likely just show what backoff values were drawn. The collected metrics are from running 10 simulations per each station number in range from 1 to 10 and simulations time set to 100 seconds. All other parameters were set using the values from Table 3.1. Figure 4.5a shows the distribution of backoff in logarithmic scale without any aggregation. The backoff is drawn from range 0 to $\min(2^n - 1, CWmax)$ where the n is a number of the failed transmissions in a row. The chart presents that this rule is perfectly met. There is definitely more backoff values for n=1 than for the n=2,3,4..., each cluster of values appears less frequently with increasing value of n parameter from the equation.

For better visualization of drawn backoff, Figure 4.5b shows the aggregated ranges in logarithmic scale. It is a good representation to show how the DCF functions works to avoid collisions and keep good performance.



(a) Individual results



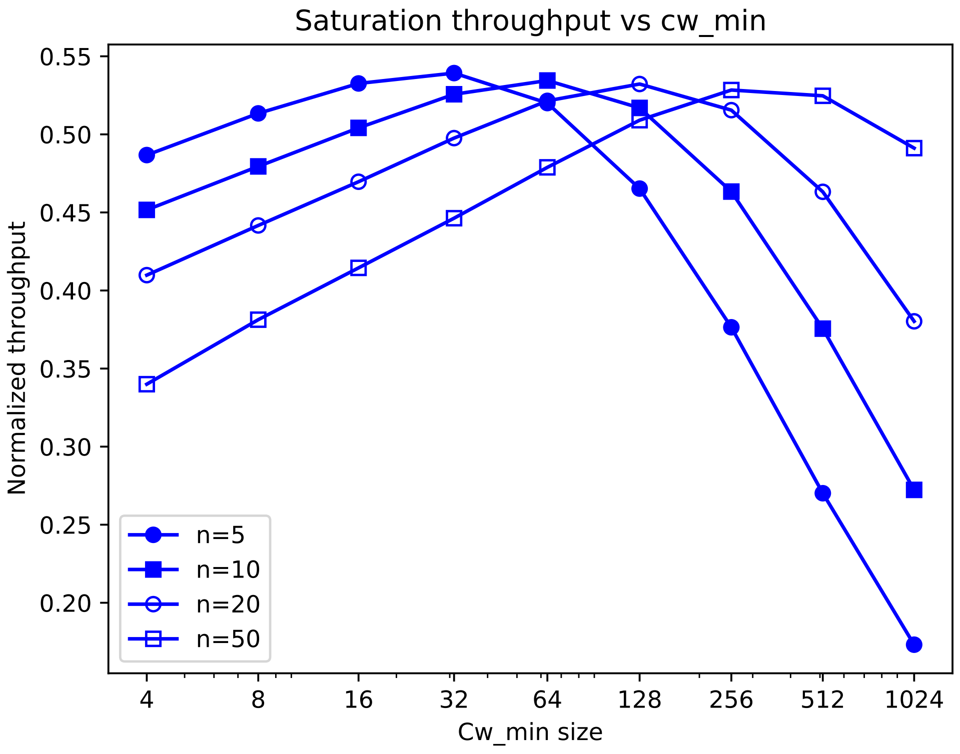
(b) Aggregation to power of 2 ranges

Figure 4.5: Drawn backoff values

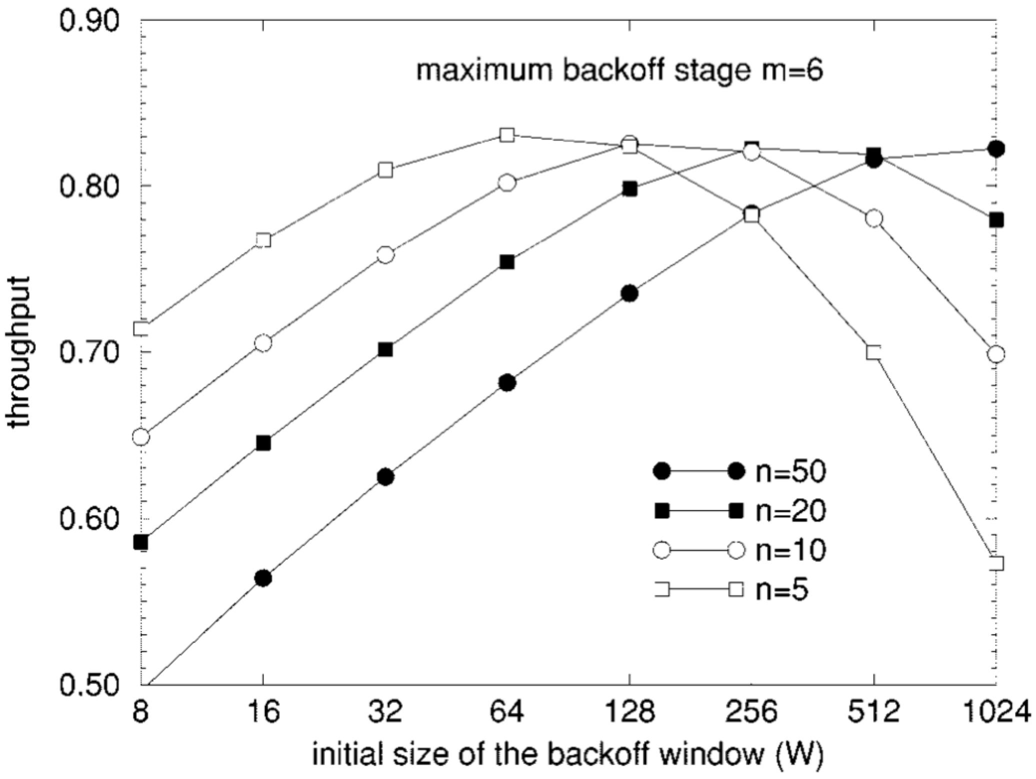
4.6. Impact of CW Size

This section will demonstrate the throughput performance in comparison to different values of the CWmin parameter (other than the default 16). All the parameters were set according to Table 3.1 except the CWmin which was changing from 4 to 1024 and was calculated from following equation: CWmin = power(2,n), where n is natural value in range from 2 to 10. There was one more change in performing of the simulation, the station were not changing from 1 to 10 but 4 scenarios were used with 5, 10, 20, and 50 transmitting stations. Obtained results are in Figure 4.6a.

This type of simulation can demonstrate the best CWmin value depends of number of currently transmitting station in the common channel. For example when there are 5 transmitting stations stations CWmin value which allows to reach the highest performance is 32, but when there are 50 stations in common channel the best value of CWmin is 256. It can be observed that when having more stations it is better to use higher value of CWmin. This is logical, the higher CW the less chance of collision. But there is also other side of the coin, higher CW means also that the single transmission lasts longer and the performance can decrease. So there is need to be careful with setting this parameter, changing its value can either increase performance or decrease it. In general for higher number of transmitting stations it is better to use higher value of CWmin. Figure 4.6b show the same simulation performed by Giuseppe Bianchi [9]. He visualized the results in the same way with only change in CWmin range, he was changing it in range from 8 to 1024. Bianchi's version of visualization uses different throughput normalization method. It can be seen that the chart is similar to the one from this thesis.



(a) DCF SimPy model



(b) Bianchi model

Figure 4.6: Throughput in comparison to different CWmin and transmitting station number

4.7. Impact of Different MCS

In this scenario DCF-SimPy is going to be validated against changing modulation and coding scheme (MCS) values. Eight different values of MCS were examined. Each of them has its own Control Rate and Data rate, all the values are shown in the Table 4.1.

In this scenario we consider 10 transmitting stations within common channel. All values were set according to Table 3.1 and simulation time was set to 100 s with 10 different runs to measure the confidenceity level. The results were compared to the ns-3.31 implementations. Figure 4.7 shows the obtained throughput values in comparison to different MCS parameter settings.

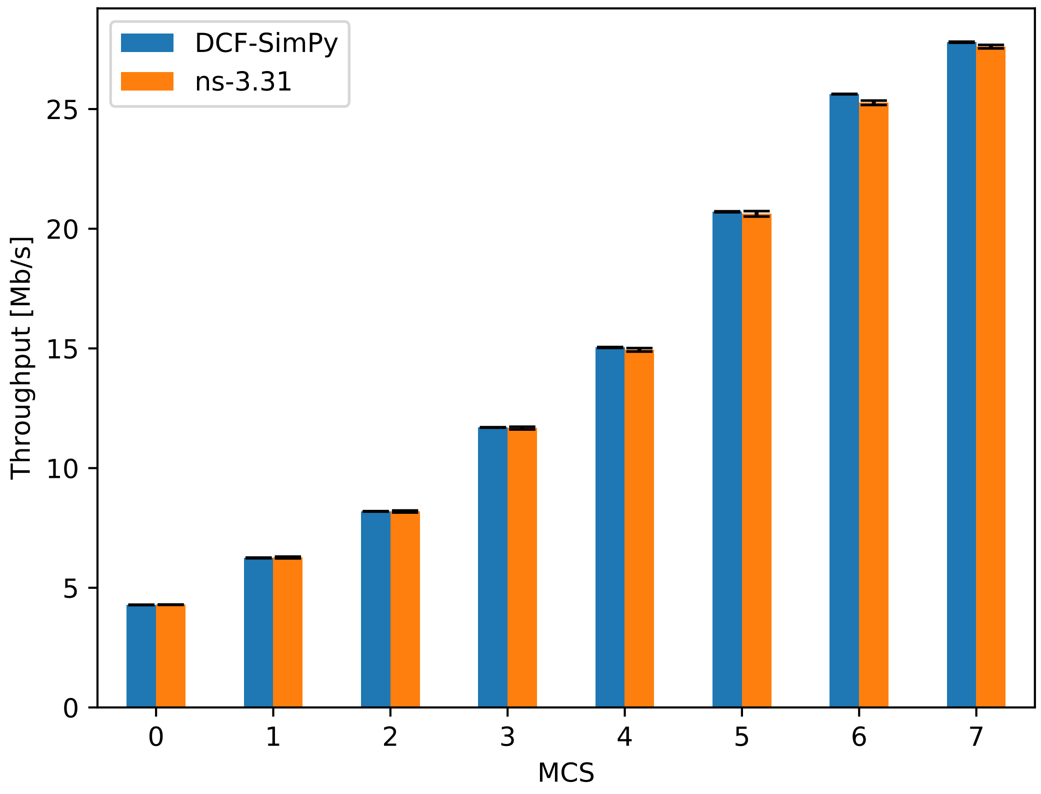


Figure 4.7: Throughput comparison with changing MCS value

At the top of each bar there is confidenceity level but is almost unnoticeable because all the result were similar for each of the runs. We would expect to get a little lower values for the ns-3.31 implementations according to the results obtained in the Section 4.2. And without any surprises the DCF-SimPy consistently generates higher throughput for each of the MCSs which proves correct implementation in this thesis.

4.8. Impact of Different Payload

Last scenario to check was the situation when again there are 10 constantly transmitting stations in common channel. All parameters are set according to Table 3.1 and this time the payload size is going to be changing from 100 B do 2000 B with step of 100 B.

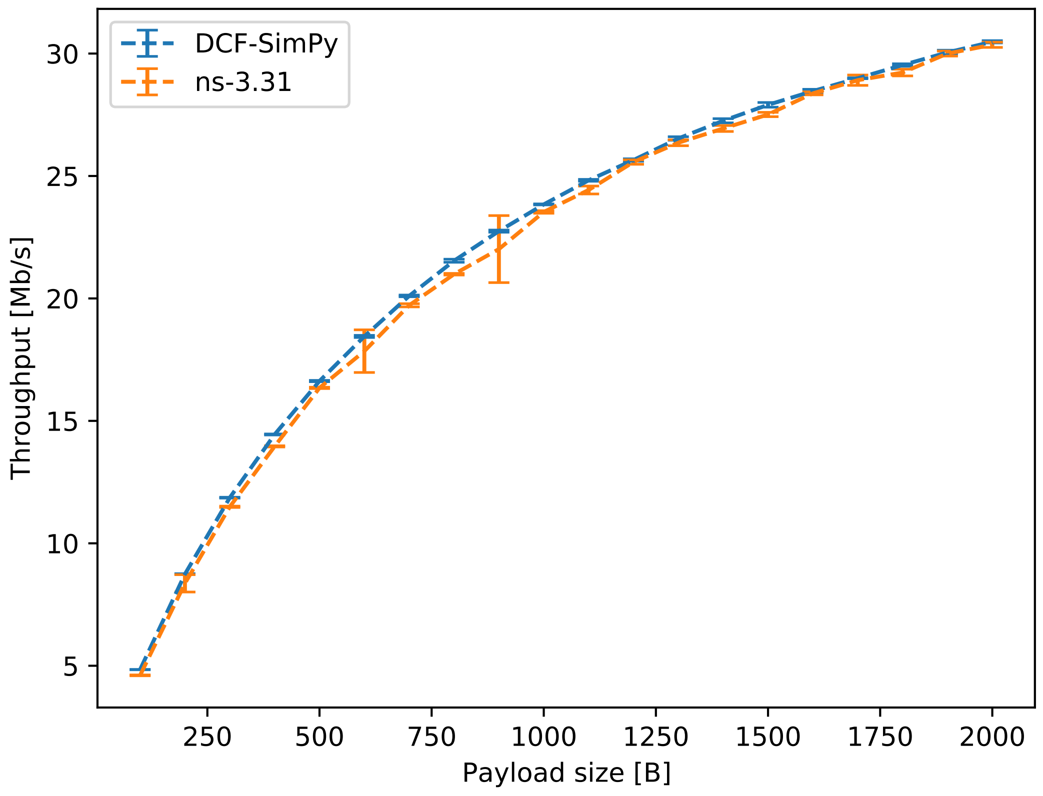


Figure 4.8: Throughput comparison with changing payload size

In Figure 4.8 it can be observed that meeting the expectations DCF-SimPy generates a little better throughput which is totally correct behavior. On the confidence level bars it can be noticed that ns-3.31 simulation sometimes generates values with larger deviation while DCF-SimPy is much more stable.

5. Conclusions

The main goal of this thesis was to implement the channel access of the 802.11a standard using discrete event simulator based on the Python SimPy library and validate the results. The goal was reached and it was possible to create simple implementation of DCF function from that standard. The implementation allows to simulate the behavior of the network with various scenarios and parameters. It can be easily used to check the expected results of the simulated environment. The implementation does not contain any known bugs or unwanted behaviors. After reaching the goal the assumptions are that the SimPy library can can be used for simulation of the 802.11a networks and with not much effort can be adopted to many different scenarios which can help to understand common use cases and problems in the real word. The library API interface was easy to use after some initial preparation thanks to well documented methods.

There are other implementation using other technologies but none of those is perfect, each of them contains some pros and cons. One can have a good performance but simplifies a lot of mechanism and the other one can implement full stack of techniques but can have worst performance and higher complexity to use it. The SimPy implementation seems to fit into in the middle of those cases, it has good performance, implements most of the standard mechanism and is easy to use and to modify.

During the validation process against other implementations it was confirmed that DCF-SimPy correctly implements DCF function. All obtained results were as expected. Even some of the simulations showed that the results were more constant and predictable than in the reference data. Furthermore the performance of this implementation is comparable to the ns-3 simulator.

In the future the implementation can be extended with RTS/CTS mechanism to increase the possible scenarios simulation. Addition of allowing the transmission in the channel where the stations are out of transmission range other would increase the reality of the simulations. Furthermore, frame fragmentation and aggregation could be added as well. There is a lot features which can be implemented to the functions from this thesis, the 802.11 family of standards is very wide and many mechanisms and algorithms are used. In the future new scenarios can be covered and validated.

Bibliography

[1] M. S. Gast, *802.11 Wireless Networks: The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2005, ISBN: 0596100523.

[2] M. Lutz, L. Lewin, and F. Willison, *Programming Python*, ser. Nutshell handbook. O'Reilly, 2001, ISBN: 9780596000851. [Online]. Available: https://books.google.pl/books?id=c8pV-TzyfBUC.

[3] T. SimPy, *SimPy API Reference*, accessed 2020-09-08. [Online]. Available: https://simpy.readthedocs.io/en/latest/api_reference/index.html.

[4] dr hab. inż. Marek Natkaniec, *Bezprzewodowe systemy teleinformatyczne*, AGH University, lecture notes, 2018.

[5] *Ns-3 reference manual*, accessed 2020-09-14. [Online]. Available: https://www.nsnam.org/docs/release/3.9/manual.pdf.

[6] K. Słowik, *Performance Evaluation of IEEE 802.11n/ac Networks with Frame Aggregation*, AGH University, diploma notes, 2017.

[7] K. Kosek-Szott, A. L. Valvo, S. Szott, P. Gallo, and I. Tinnirello, *Downlink channel access performance of nr-u: Impact of numerology and mini-slots on coexistence with wi-fi in the 5 ghz band*, 2020. arXiv: 2007.14247 [cs.NI].

[8] G. Raaen, *The WiFi AirTime Calculator*, accessed 2020-09-16. arXiv: 2007.14247. [Online]. Available: https://gjermundraaen.com/thewifiairtimecalculator/.

[9] G. Bianchi and I. Tinnirello, "Remarks on IEEE 802.11 DCF performance analysis," *IEEE Communications Letters*, vol. 9, no. 8, pp. 765–767, 2005. DOI: 10.1109/LCOMM.2005.1496609.