Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektorniki i Telekomunikacji

KATEDRA TELEKOMUNIKACJI

PRACA DYPLOMOWA

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

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

Autor:	Cecylia Borek
Kierunek studiów:	Teleinformatyka
Typ studiów:	Stacjonarne
Opiekun pracy:	dr hab. inż. Szymon Szott

Kraków, 2021

I would like to thank my supervisor, Szymon Szott, for providing guidance, encouragement and feedback throughout writing the thesis.

Contents

List of Figures

List of Tables

1. Introduction

Wireless networks have become ubiquitous in recent years. Though not as secure or reliable as wired networks, they are an inseparable part of our lives, in which we are constantly on the move. Wireless networks gained enormous popularity mainly by allowing the mobility and flexibility of users. We may be provided with Internet access without the need to be physically connected. Additionally, we can use the Internet in places like cafes, airports or shopping centers, where it would not be possible to connect through a wire. WLANs (Wireless Local Area Networks) are also of great use in offices, universities, and households, where even though we could be using a fixed connection, most people opt for wireless one, as it is much more comfortable and flexible.

While being very convenient for end users, wireless networks pose many challenges for their providers and developers. A radio interface introduces many problems that do not exist in fixed networks [2]. A radio wave meets many obstacles on its path, which disturb its propagation and cause interference. There is also much more noise present in the wireless medium than in the wired one. Radio spectrum is a limited resource, which makes its efficient utilization a crucial aspect. Additionally, radio is a truly broadcast medium, which means only one station at a time may transmit, while others need to listen.

The IEEE 802.11 standard, released and maintained by the Institute of Electrical and Electronics Engineers – IEEE – describes and specifies the functionality of the physical (PHY) and medium access control (MAC) layer of WLANs. These two layers are the most important in wireless connectivity, addressing the issues and challenges of the wireless medium. The scope of this work is the MAC layer and, more precisely, the channel access mechanism.

The channel access mechanism of an IEEE 802.11 network is one of the aspects that distinguishes wireless networks from wired ones. Due to characteristics of the radio interface methods used in traditional fixed networks could not be incorporated into wireless networks. The IEEE 802.11 standard contains a detailed description of the methods and algorithms used in the medium access procedure. Scrutinizing and looking closely into this procedure helps us better understand the wireless network, its crucial features, advantages, and limits.

The goal of this thesis is to create a simulator of the channel access procedure of the IEEE 802.11 network. This is realised with the use of the Python NumPy library. The simulator should be configurable, allowing for different scenarios and test cases. Results of the simulation are meant to be confronted with other widely used network simulators and theoretical expectations. The simulator is created with the thought of simple studies or research as its main use cases.

The thesis is structured as follows. Chapter 2 consists of a basic description of the IEEE 802.11 network and its channel access mechanism defined in the standard. It also contains a description of various existing implementations of the IEEE 802.11 channel access procedure which are later used in the validation process. Chapter 3 describes the simulator concept, initial assumptions, and simulator implementation. Chapter 4 presents the validation scenarios, their results, and simple performance analysis of the simulator. Finally, Chapter 5 summarizes the work.

2. Channel Access in IEEE 802.11 Networks

This chapter presents the basic theoretical knowledge necessary to understand the thesis. The description focuses on the channel access procedure in WLANs. It looks into the special characteristics, requirements and challenges of the wireless transmission medium, as well as algorithms and solutions used. It also contains an overlook of some of the already existing implementations of the channel access procedure.

2.1. CSMA with Collision Avoidance in Wireless Networks

CSMA/CA, which stands for Carrier-Sense Multiple Access with Collision Avoidance, is the main and mandatory channel access method of the IEEE 802.11 standard [4]. The carrier sensing mechanism is realized by every station listening to the channel before an attempt to transmit, in order to check whether any transmissions are already happening at a given moment. Collision avoidance is done by a station waiting for a random period of time before actually starting the transmission, after sensing the channel idle.

The collision avoidance is a crucial part of the channel access procedure in wireless networks. In wired networks, CSMA with collision detection is used. Whenever a station senses a collision in the channel, it stops transmitting and retransmits the packet after some random time. This, however, is not possible in wireless networks. The station's antenna is not able to simultaneously transmit data and listen to the channel's state, as the interference coming from the transmission significantly disturbs the received signal. In wireless networks the only way to determine that a collision occurred is by the sender not receiving the acknowledgement (ACK) frame from the endpoint, during a certain amount of time. This, however, is time consuming, as the station transmits the whole packet and then waits for the period called ACK Timeout before assuming a collision occurred.

2.2. DCF Function Description

WLANs based on the IEEE 802.11 standard utilize the Distributed Coordination Function (DCF) as a medium access method. The DCF function implements the carrier sensing mechanism by measuring the instantaneous signal power in the channel. For collision avoidance, DCF uses the Binary Exponential Backoff algorithm (BEB) which is described in detail in Section 2.3. In DCF, before a station may start a transmission, it has to wait for the DIFS (DCF Interframe Space) interval while checking the channel status. If, during that interval, the channel becomes busy, the station delays its transmission. Otherwise it waits for another period – the backoff period – and, if no other transmission is detected during that time, it starts a transmission after its backoff counter reaches zero. In case of a detected transmission during the station's backoff period, it stops its counter and resumes to count it down after the next DIFS interval.

DCF also implements a positive acknowledgement mechanism. The acknowledgement frame is sent out to the sender in case of a properly received data frame. The receiver waits for the SIFS (Short Interframe Space) interval, from the end of the data frame, before sending the acknowledgement frame. Both DIFS and SIFS intervals are dependent on the medium and fulfil the dependence: $DIFS > SIFS$, which makes the acknowledgement frames of higher priority than data frames. The default values of DIFS and SIFS for the IEEE 802.11a standard are presented in Table 3.2.

2.3. Binary Exponential Backoff Algorithm

As mentioned in Section 2.2, DCF incorporates BEB as a collision avoidance mechanism. The backoff period, for which a station has to wait after the DCF interval (DIFS), minimizes the amount of collisions. Without it, all stations would begin their transmission after waiting for the time of a DCF interval. The backoff period is randomly selected by each station after the channel has been idle for the DIFS interval and is stored in the station's backoff counter. In case a station already has a non zero value in its backoff counter, from the previous contention round, it does not select a new one. The backoff period can be expressed by the equation:

$$Backoff = Random() \cdot SlotTime$$

where $Random()$ is a pseudorandom integer from the range $[0, CW]$. CW is an ordered set of integer powers of two, minus one, beginning with CW_{min} and ending with CW_{max}, and can be represented as:

$$CW = \{min(2^j \cdot (CW_{min} + 1) - 1, CW_{max}) : j \in \langle 0, M \rangle\}$$

where j denotes the current retransmission and M is the maximum number of retransmissions, after which a frame is dropped.

After an unsuccessful transmission, a station takes the next value from the CW set, until reaching CW_{max}. If a transmission is successful, the station's current value of CW is reset back to CW_{min}. For the IEEE 802.11a/n/ac/ax standards, the default values of CW_{min} and CW_{max} are, respectively, 15 and 1023.

2.4. Implementations of IEEE 802.11 Channel Access

There are many different implementations of the IEEE 802.11 channel access mechanism. Apart from the ones used in real networks, as a part of their software, there are also many simulators and models, whose purpose is to conduct performance studies. This chapter presents a few of the latter, chosen to best fit our study case. Later, in Chapter 4, they are used to validate the simulator created for this thesis, by comparing different statistics and results.

2.4.1. ns-3

ns-3 is a free, open-source, discrete-event network simulator [6]. It allows flexible configuration of a full-stack network simulation, so that it can be utilized for different scenarios and network characteristics. In this work, the release 3.31, from 27th of June 2020 was used. For the purpose of validating the DCF-NumPy results, ns-3 simulator was configured as to stay in accordance with the examined DCF-NumPy simulator (i.e., in accordance with parameters specified in Table 3.1). As input parameters, the number of stations and simulation time were provided. For each simulation round, the simulator returned the aggregate throughput and mean probability of collision. Each ns-3 simulation was run 10 times for each number of contending stations. Later, the results with the same number of contending stations were grouped, and the means of throughput and collision probability were obtained. The time for each simulation was constant and equal to 100 s. The source code of the simulator, used in this work, can be found in Appendix B.

2.4.2. Bianchi's Analytical Model

The analytical model is a Matlab implementation of Bianchi's model proposed and described in [1]. The Matlab function takes as input the number of contending stations – N, as well as CW_{min} and CW_{max}. It returns the probability of collision, which is calculated by numerically solving one of Binachi's equations:

$$p_{coll} = 1 - (1 - \tau)^{N-1},$$

where τ is the probability that a station transmits in a randomly chosen slot time. The code for this model was obtained, with permission, from Ilenia Tinnirello of the University of Palermo and is included in Appendix E.

2.4.3. Frame Duration Matlab Model

The *Frame Duration Matlab Model* is a model created as an MSc project [8]. The main focus in there is put on very accurate calculations of frame duration for IEEE 802.11a/n/ac standards (taking into account frame aggregation for 802.11n/ac networks). It allows various configurations of MAC layer parameters. Due to its high precision in obtaining frame duration and the whole transmission duration, it was used to validate the calculation of transmission time in DCF-NumPy simulation. The source code for this model was provided by the supervisor and can be found in Appendix D.

2.4.4. WiFi AirTime Calculator

Another model used to validate the transmission time calculation of the DCF-NumPy simulation is the *Wi-Fi AirTime Calculator* spreadsheet [7]. The spreadsheet provides scenarios for IEEE 802.11a/n/ac as well as the IEEE 802.11ax standards. For every standard it allows modifying various parameters as to fit to specific network configuration. The spreadsheet returns the Wi-Fi transmission time, calculated based on the given standard's ruleset.

2.4.5. Matlab Coexistence Model

Last model used for validation is the *Matlab Coexistence Model*, described in [5]. It is a Monte Carlo simulation model, implemented in Matlab. It implements similar approach as the one in DCF-NumPy simulation, which iterates through consecutive contention rounds. At each round the channel access procedure is simulated and it is determined if a collision occurred or not. The model's function takes, among others, the number of contending stations, CW_{min} and CW_{max} as arguments and returns the probability of collision. It is also meant to simulate the environment, where Wi-Fi, New Radio Unlicensed (NR-U), and License Assisted Access (LAA) nodes coexist, however, for the purpose of validating DCF-NumPy only the Wi-Fi stations were configured. The source code of the model can be found in Appendix C.

2.4.6. Conclusions

As it can be seen, all of the presented implementations offer a slightly different approach and put focus on different aspects of the channel access mechanism and wireless transmission. The ns-3 simulator is a very complex one. It allows the configuration of many parameters and creation of many different test scenarios. However, because of its complexity, it is harder to configure and requires its user to be well acquainted with its structure. For some basic research and studies it may be a too complicated tool. Bianchi's analytical model is based on numerically solving an equation to obtain the probability of collision. Therefore we do not get the network throughput, which often is a useful and desired performance parameter. The *Frame Duration Matlab Model's* main focus is on calculating the frame duration and network throughput with high precision. It, however, does not allow for manipulating the value of CW in order to investigate how it affects network performance. It also does not return the probability of collision. The *Wi-Fi AirTime Calculator* takes a similar approach as the *Frame Duration Matlab Model*. It calculates the throughput and frame duration very accurately, but the CW value range is limited and the probability of collision is not calculated. Finally, in the *Matlab Coexistence Model* a lot of effort is put into simulating an environment where Wi-Fi, New Radio Unlicensed and License Assisted Access nodes coexist. For simple Wi-Fi network simulations a big part of its functionality is not utilized. The simulator, created for the purpose of this thesis, takes a yet another approach on the channel access. It is meant to be simple, so that it is easy to configure and fast to run. It is supposed to return both the probability of collision and the network throughput, which enables further analysis. The input parameters are intended to be fully configurable, including CW, number of stations, MAC payload, and data rate.

3. Implementation

This chapter describes the implementation of a basic IEEE 802.11 channel access simulator – the DCF-NumPy simulator – created for this thesis. The description consists of scenario considered in this work, tools used and solutions implemented to solve some theoretical assumptions. Additionally, information included in this chapter should allow to recreate a similar simulator.

3.1. Initial Assumptions

For the purpose of creating an IEEE 802.11 channel access simulation model, several assumptions were made. All presented results and conclusions should be considered with regard to to the following:

1. The channel access function is in accordance with IEEE 802.11a. This amendment to the IEEE 802.11 standard contains the basic version of channel access which has remained principally unchanged in subsequent amendments (up to the current 802.11ac).
2. For all simulation runs it is assumed that perfect radio channel conditions are available – we only focus on data link layer behaviour.
3. There is no Access Point – all stations contend for the channel access on the same terms.
4. There are no hidden stations, the hidden station problem is not taken into consideration – all stations are within range and performance of only one Wi-Fi network is examined. Therefore, the RTS/CTS mechanism is not implemented.
5. Only the DCF function is used and implemented, the EDCA function is left for future work – the focus is placed on investigating a basic, best-effort network, without packet prioritization.
6. Fragmentation and aggregation of frames are omitted from the analysis.
7. The transmission of a data frame and the subsequent acknowledgement frame are treated as a single, unbreakable transmission.
8. Only unicast transmissions are examined – there is no implementation of multicast or broadcast communication.
9. The examined network is in a saturated state – each station constantly has a frame to be sent.

3.2. NumPy Library

The implementation is based on the NumPy library [3], which is an open source Python library supporting numerical calculations and operations on arrays and matrices. Standard library lists in Python are arrays of references to other objects. This allows list elements to be of different types but slows down operations performed on them. The interpreter needs to look up every object at different location pointed to in the array. NumPy arrays, however, are stored as continuous, fixed blocks of memory. This requires all elements of a NumPy array to be of the same type, but makes operations much faster. NumPy in its functions uses strides – number of bytes between consecutive values in an array. Therefore, iterating over an array and performing operations on it is more time efficient, then using standard library functions. One of the biggest assets of the NumPy library is its implementation of different mathematical operations on arrays. Among them are basic operations, element-wise or with scalars. There are also logical operations, transpositions, reductions (like sum or maxima finding) and statistic functions (like mean or standard deviation). The time efficient mathematical computations on arrays are really useful for performing the simulation.

3.3. Simulator Concept

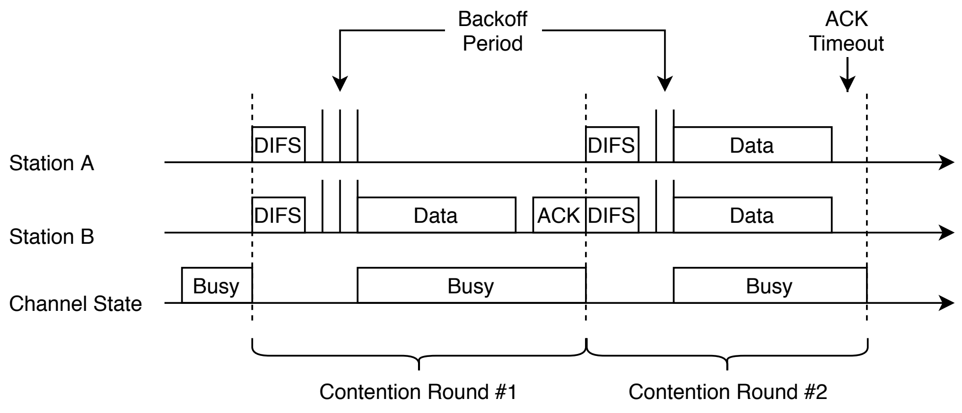

Figure 3.1: Example scenario

The NumPy-based implementation of DCF is based on the following observations. In case of a network in a saturated state, there are always frames to be sent, so transmission attempts occur one after another. As shown in Figure 3.1, access to the medium consists of consecutive contention rounds, each being a period of time in which stations contend for the channel access and transmit data afterwards. Contention rounds may have different durations, depending on the duration of the backoff period, whether a transmission was successful or not and the transmission duration.

The main part of the simulator is the simulation function – *dcf_simulation* – which performs the actual simulation of the channel multi-access procedure in the Wi-Fi network. It takes in arguments described in Table 3.1 and returns two values – mean aggregate network throughput and mean probability of collision. The function iterates through contention rounds. It is responsible for selecting random backoffs for stations and updating them between contention rounds. It also determines if a collision occurred or not. Finally, the function collects statistics from all contention rounds of a single simulation. It records successful and unsuccessful transmissions for every station. It also calculates the time of each round by summing up backoff period, data frame duration, acknowledgement frame duration, and Short Interframe Space – the time period between them as defined in IEEE 802.11. In case of an unsuccessful attempt of transmission, the contention round time is the sum of backoff period, data frame duration, and ACK timeout – the time a station waits for the acknowledgement frame before it assumes a data frame did not arrive at the receiver. The next section – Section 3.4 – describes in detail the simulation function.

3.4. Simulation Function Description

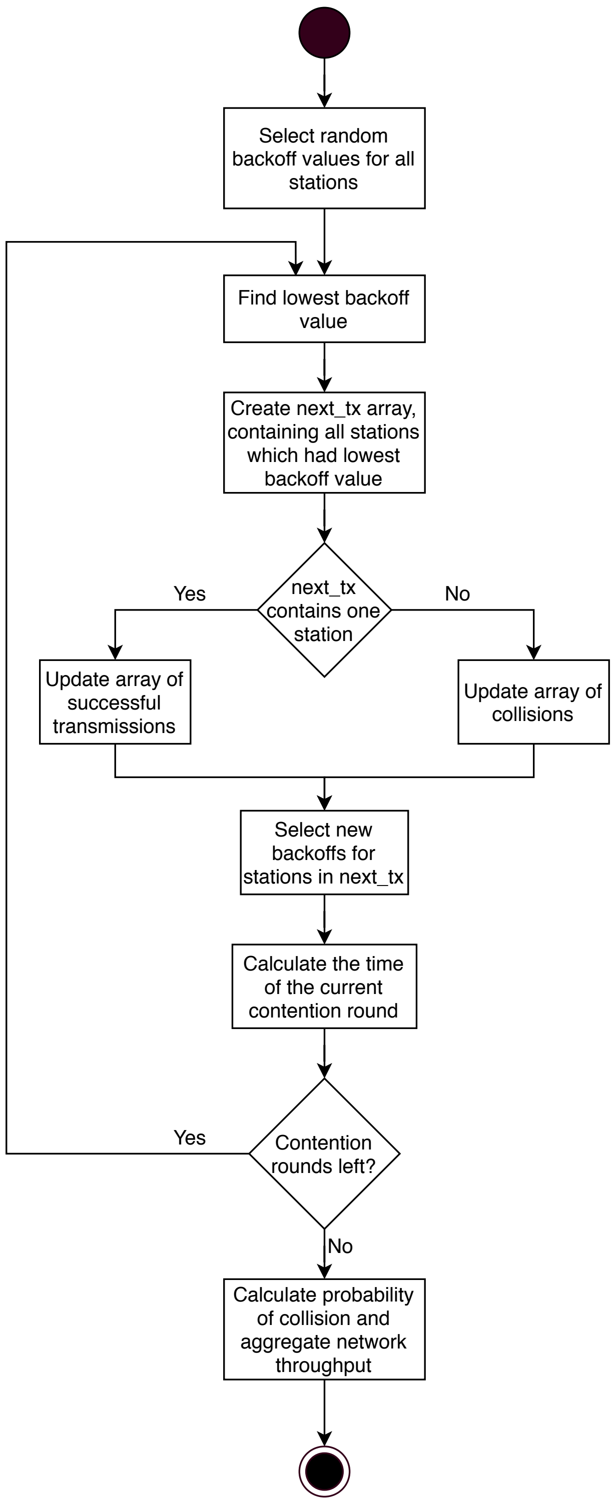
Select random backoff values for all stations
Find lowest backoff value
Create next_tx array, containing all stations which had lowest backoff value
next_tx contains one station
Yes
No
Update array of successful transmissions
Update array of collisions
Select new backoffs for stations in next_tx
Calculate the time of the current contention round
Contention rounds left?
Yes
No
Calculate probability of collision and aggregate network throughput

Figure 3.2: Simulation function block diagram

The *dcf_simulation* function, as mentioned before, is the main part of the simulator. Its block diagram is presented in Figure 3.2. The full code of the function can be found in Appendix A. The function consists of three main parts. The first one initializes NumPy arrays to store all simulation data. Among these are counters of per-station collisions, successful transmissions and retransmissions, value of current CW, and random backoff period for each station. All the operations on data, conducted later, are performed with the use of NumPy library functions.

The second, largest part of the simulation function, is the for loop which iterates through contention rounds as described in Section 3.3 and performs certain operations at each round. Shown in Listing 3.1, the NumPy *amin* function is used to find the lowest value in the backoff array, which is the shortest backoff period among all stations. Then, the NumPy *where* function returns an array of indexes of the lowest backoff values. This represents all the stations that have selected the shortest backoff in a given contention round. Then, the lowest backoff value is subtracted from all stations' backoff values stored in the *backoffs* array. This constitutes the time stations have already waited before an attempt to transmit. Additionally, 1 is subtracted, which constitutes for the current round. At the end of every round, a new backoff is selected only for the stations that reached 0 (i.e., selected the lowest backoff), other stations start the next round with their backoff times left from the previous one.

```
        min_backoff = np.amin(backoffs)  # find the minimal value of backoff
        next_tx = np.where(backoffs == min_backoff)[0]  # an array of index(es)
        # of station(s) with lowest backoff(s)
        backoffs = backoffs - min_backoff - 1  # subtract from all stations' backoffs,
        # time they've already waited
```

Listing 3.1: Backoff operations

In Listing 3.2 we can see the procedure of determining if a transmission was successful or not and assigning new *CW* values and counters accordingly. By examining the length of the *next_tx* array, that is the array of indexes of the lowest backoff values, we determine if the transmission was successful or if a collision occurred. If *next_tx* contains only one element, only one station had the lowest backoff value and its transmission was successful. The counter of successful transmissions for that station is increased. The CW value of the station is reset back to CW_{min} and a new backoff value is randomly selected. Otherwise, if the length of *next_tx* is greater than one, more than one station made an attempt to transmit and a collision occurred. In that case, counters of collisions for given stations are increased. Provided the counter of retransmissions for a station does not exceed the limit, the counter is increased and the CW value is increased accordingly to the BEB algorithm. If the station's counter of retransmissions exceeds the retry limit, it is reset back to 0 and the CW value is reset to CW_{min}. Finally, a new backoff value is randomly chosen for all stations that participated in the collision.

```
        if len(next_tx) == 1:  # only one station had smallest backoff - success
            successful[next_tx] += 1
            retransmissions[next_tx] = 0
            cw[next_tx] = cw_min+1
            # selecting new backoff for the stations
            backoffs[next_tx] = np.random.randint(low=0, high=cw[next_tx])
            # appending the newly selected backoff to the array of all backoffs
            all_backoffs = np.append(all_backoffs, backoffs[next_tx])
        else:  # more than one station with smallest backoff - collision
            collision = True  # collision - True, necesarry for transmission time calculation
            for tx in next_tx:
                collisions[tx] += 1
                if retransmissions[tx] <= retry_limit:
                    retransmissions[tx] += 1
                    cw[tx] = min(cw_max+1, cw[tx]*2)
                    # cw is always the upper limit (excluded), therefore we only need to
                    # multiply it by two to get the next value of cw limit
                else:  # if retry limit is met, cw and retransmissions couter are reset
                    retransmissions[tx] = 0
                    cw[tx] = cw_min + 1
                # new backoff chosen for all the station that collided
                backoffs[tx] = np.random.randint(low=0, high=cw[tx])
```

Listing 3.2: Determining if collision occurred

At the end of each contention round, its total time is derived. This is done with a function that sums up DIFS, the backoff period of a given round, the data frame transmission time, SIFS and acknowledgement frame transmission time (or ACK timeout). The durations of all rounds are stored in an array, and are later used to calculate throughput.

The third part of the *dcf_simulation* function calculates statistics of the whole simulation and returs them as simulation results. As shown in Listing 3.3 , the collision probability for every station is calculated by dividing *collisions* by the sum of *collisions* and *successful* arrays. Next, the mean value of the collision probability is obtained with *np.mean* and assigned to the results of the simulation.

```
# calculation of collision probabilty per station and mean value of it
collision_probability = collisions/(successful + collisions)
simulation_results.mean_collision_probability = np.mean(
    collision_probability)
```

Listing 3.3: Collision probability calculation

Listing 3.4 depicts throughput calculation. All elements of *tx_time* are summed up with the help of *np.sum* and the total time of the whole simulation is obtained. Throughput per station is calculated, by dividing payload of all successful transmissions of the station by the total time of the simulation, described above. The result of this operation is an array called *throughput*. Aggregate network throughput is simply obtained by summing up all elements of the *throughput* array with *np.sum*. Aggregate throughput is assigned to final simulation results.

```
# calculation of throughput per station in b/s and aggregate throughput of whole network
simulation_time = np.sum(tx_time) * slot_time
throughput = successful * mac_payload * 8 / (simulation_time)
simulation_results.network_throughput = np.sum(throughput)
```

Listing 3.4: Throughput Calculation

3.5. Simulation Parameters

Table 3.1: Input parameters

Parameter	Description	Value
N	Number of contending stations	1-10
CW_{min}	Minimal value of contention window	15
CW_{max}	Maximal value of contention window	1023
seed	Seed for the NumPy random number generator	1-10
data_rate	Data rate of the data link layer packet [Mb/s]	54
control_rate	Control rate in the data link layer [Mb/s]	24
payload	Payload of the data link layer packet [B]	1500

The simulation function allows specifying various input parameters to test different scenarios. Table 3.1 depicts the parameters and their default values, which were used in this simulation. However, these parameters may be adjusted accordingly to a specific use case, as long as they meet the requirements. There are also various parameters defined in the body of the simulation function. These values are fixed and we cannot modify them. Table 3.2 displays them and their values.

Table 3.2: Internal function parameters

Parameter	**Description**	**Value**
retry_limit	Retry limit, as specified in 802.11a	7
simulation_rounds	Number of rounds performed in simulation	10^5
slot_time	Duration of a single slot, as defined in 802.11a [s]	$9 \cdot 10^{-6}$
SIFS	Short Interframe Space, as defined in 802.11a [μs]	16
DIFS	DCF Interframe Space, as defined in 802.11a [μs]	34

3.6. Simulation Workflow

The *dcf_function*, described in detail in Section 3.4, takes in arguments shown in Table 3.1 and returns the average network throughput and average probability of frame collision in the network. In this work a Python script *run_simulation.py* is used to run the function with different argument values. The script iterates through sets of arguments and calls the *dcf_simulation* with given values. For every set of N, CW_{min} and CW_{max} it runs the simulation 10 times in order to achieve reliable results. With the help of the Pandas library, the results with the same values of N, CW_{min} and CW_{max} are grouped and the mean value of collision probability and throughput is calculated. The results are saved as a csv file. Later, a Python script *validation.py*, using the Pandas library, adds results from the validation models to the simulation results. The final, combined results are saved as an another csv file. Finally, *draw_graphs.py* plots the data from the final results csv file with the help of the Matlpotlib library. The workflow process is depicted in the form of a chart in Figure 3.3.

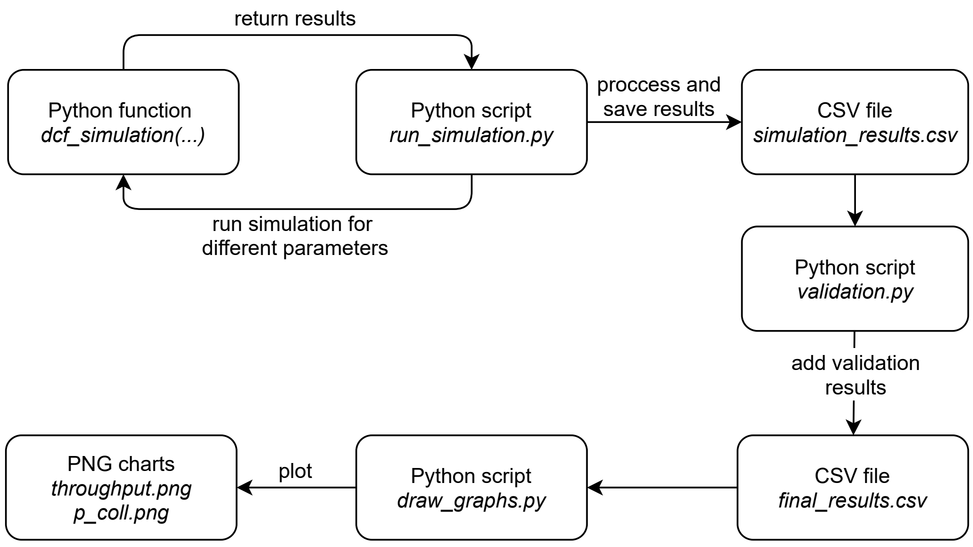

Figure 3.3: Workflow chart

4. Validation and Example Performance Analysis

In order to validate the DCF-NumPy implementation, a simulation campaign was performed to compare its results with various IEEE 802.11 channel access models, described in Section 2.4. This chapter presents results of the validation and conclusions arising from them. Three different parameters were compared during the validation: frame duration, probability of collision, and aggregate network throughput. Probability of collision and network throughput are simple yet very accurate measures characterising the multi-access procedure in a WiFi network. Additionally, the backoff values distribution was explored and confronted with the theoretical assumptions. Finally, the impact of packet size and MCS (Modulation and Coding Scheme) on throughput was examined and compared between two simulators: ns-3 and DCF-NumPy. At the end of this chapter a simple performance analysis of the created simulator is presented. As all of the validation scenarios require running the DCF-NumPy simulator with different input parameters, the beginning of this chapter briefly describes how to run a simulation in DCF-NumPy. All graphs presented in this chapter depict data with 95% confidence intervals.

4.1. Running a DCF-NumPy simulation

To perform a simulation in DCF-NumPy, it is enough to run the *dcf_simulation* function with desired parameters. The list of all possible input parameters, their meaning and default values can be found in Table 3.1. Listing 4.1 depicts the Python code used to run a simulation for 10 contending stations, CW_{min} equal to 15, CW_{max} equal to 32, the seed for the NumPy random number generator equal to 1, and the rest of the parameters equal to their defaults. It also shows how the the returned values – collision probability and network throughput – can be accessed.

```
sim_results = dcf_simulation(N=10, cw_min=15, cw_max=32, seed=1)

throughput = sim_results.network_throughput
p_coll = sim_results.mean_collision_probability
```

Listing 4.1: Running a DCF-NumPy simulation

4.2. Frame Duration

The simplest validation was performed by comparing a single frame duration, between different models and simulations. The results used in this comparison come from the Frame Duration Matlab Model, the WiFi AirTime Calculator spreadsheet, the ns-3 simulator and the DCF-NumPy simulation. The frame times, presented in Figure 4.1, were obtained for a data rate equal to 54 Mb/s and the MAC layer payload equal to 1500 B. This validation checks the correctness of the *transmission_time* function of DCF-NumPy. The *transmission_time* function is used to calculate times of consecutive contention rounds, which later are used to derive the network throughput. As we can see, times from DCF-NumPy and other models are almost identical, which leads to the conclusion, that the *transmission_time* function of the DCF-NumPy is precise enough.

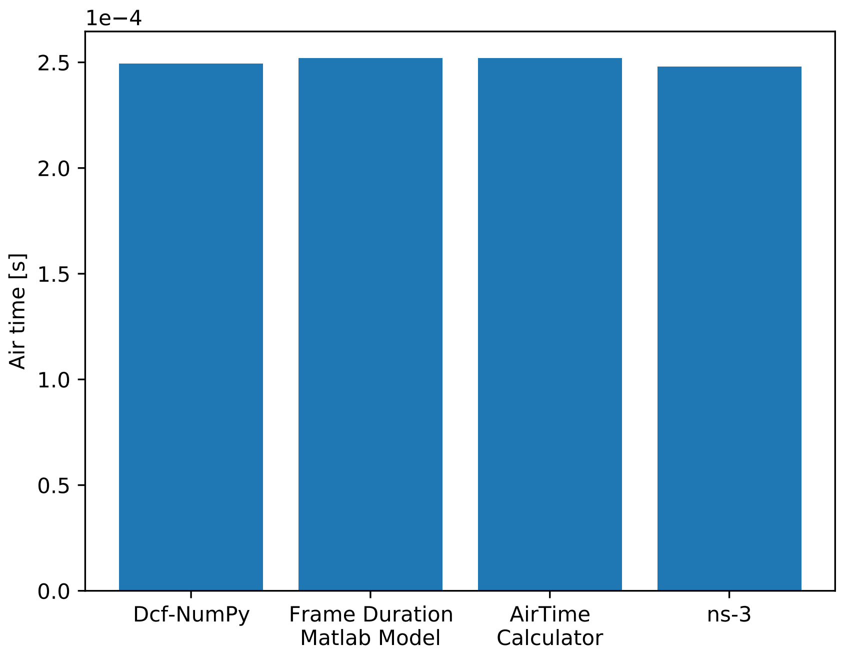

Figure 4.1: Frame duration comparison

4.3. Collision Probability

In terms of collision probability, results from the ns-3 simulator, Bianchi's Analytical Model and Matlab Coexistance Model were compared with the DCF-NumPy simulation results. As shown in Figure 4.2, the outcomes of all validation models are close to the ones of the DCF-NumPy simulation. The only noticeable deviation is between ns-3 and all the other models for 8 contending stations. This however, may be treated as an anomaly in the ns-3 simulation, as it is the only result lying outside the curve. The mean squared errors (MSE), which measure the average squared difference between two values, are around the order of 10^{-4} – 10^{-5}. This is an acceptable value for a simulation with the level of complexity such as in DCF-NumPy. Additionally, for larger amounts of contending stations, the probability of collision derived from ns-3 is slightly lower then the ones from other models. This, most likely, is a result of marginal differences in implementation of the channel access procedure. The ns-3 is a complex simulator, designed to accurately depict the real life scenarios. Therefore, in case of this validation, the consistency of the DCF-NumPy and Bianchi's Analytical Model results is the most important indicator of accuracy.

The collision probability in a network depends mostly on the medium access procedure. In terms of IEEE 802.11 networks, it is the BEB algorithm and consequently backoff values, selected throughout the simulation, that have the biggest impact on it. Figure 4.3 presents a histogram of all backoffs selected during one DCF-NumPy simulation round, for $CW_{min} = 15$, $CW_{max} = 1023$ and number of contending stations equal to 10.

As we can see, lower values are selected more frequently than the bigger ones. This concurs with the logical explanation that lower backoff values can be also selected when bigger limits are set. Additionally, the biggest CW values, that limit the backoff value, are used only after several consecutive collisions for a given station occur.

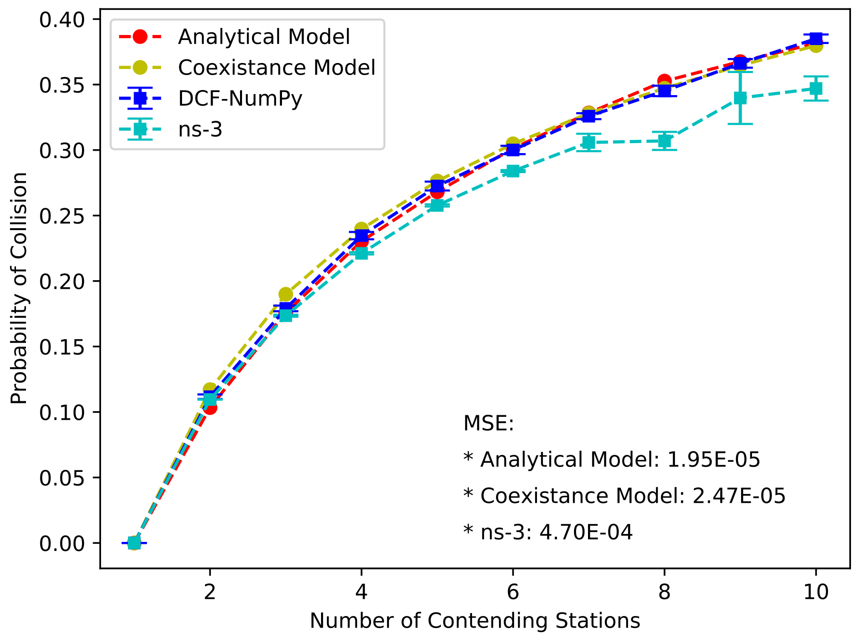

Figure 4.2: Collision probability for CW_{min} = 15 and CW_{max} = 1023

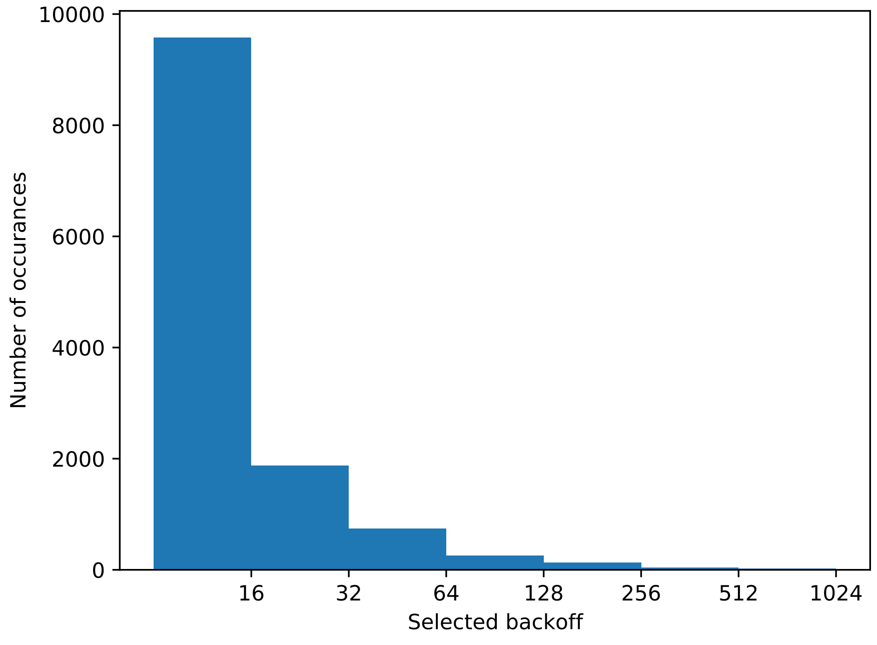

Figure 4.3: Histogram of backoff values

After each successful transmission the CW of a station is set back to its default value, equal to 15. This means that the CW of a station is more often set to the lower values and rarely reaches the highest limitations.

4.4. Throughput

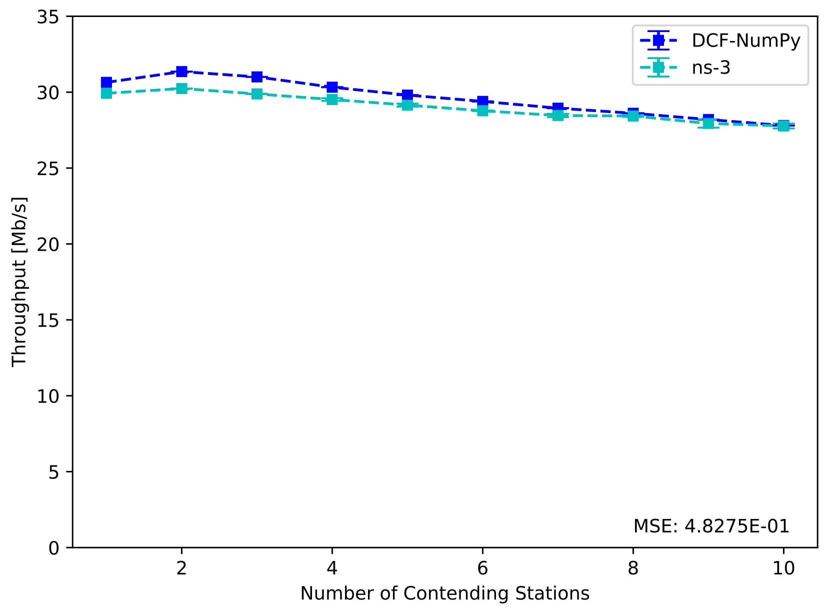

Figure 4.4: Aggregate throughput for CW_{min} = 15 and CW_{max} = 1023

Another parameter used to validate the simulation results was the aggregate network throughput. The DCF-NumPy simulation was compared with the ns-3 simulation. The outcomes of this comparison are presented in Figure 4.4. As we can see, the results from ns-3 are a little lower, especially for small amounts of contending stations. This is a consequence of ns-3 being a more precise and realistic simulator than DCF-NumPy. As the simulator implemented in this work is a very basic one, it does not take into account many factors that are present in a real network. Additionally, the figure shows that both simulators properly reflect decrease of the aggregate network throughput for a growing number of contending stations. In this scenario the highest possible rates were used, i.e., 24 Mb/s for the control rate and 54 Mb/s for the data rate. The MAC layer payload for both simulators was set to 1500 B.

4.5. Impact of Packet Size

Next, the impact of packet size on the network throughput was examined and compared between ns-3 and DCF-NumPy. For this purpose, ns-3 and DCF-NumPy simulations were performed with different MAC payloads, varying from 100 B to 2200 B. The upper limit was set by the fragmentation threshold of the ns-3 simulator. Both simulations were run for 10 contending stations, data rate equal to 54 Mb/s and control rate equal to 24 Mb/s. The outcomes of the performed simulations are presented in Figure 4.5. As can be seen, results from ns-3 and DCF-NumPy lay close to each other. Moreover, the graphs coincide with the expected results that enlarging the MAC payload (to a certain threshold) increases the network throughput. This is because the overhead sent in headers stays the same for different payload sizes. Second reason is that for smaller payload sizes the contention rounds

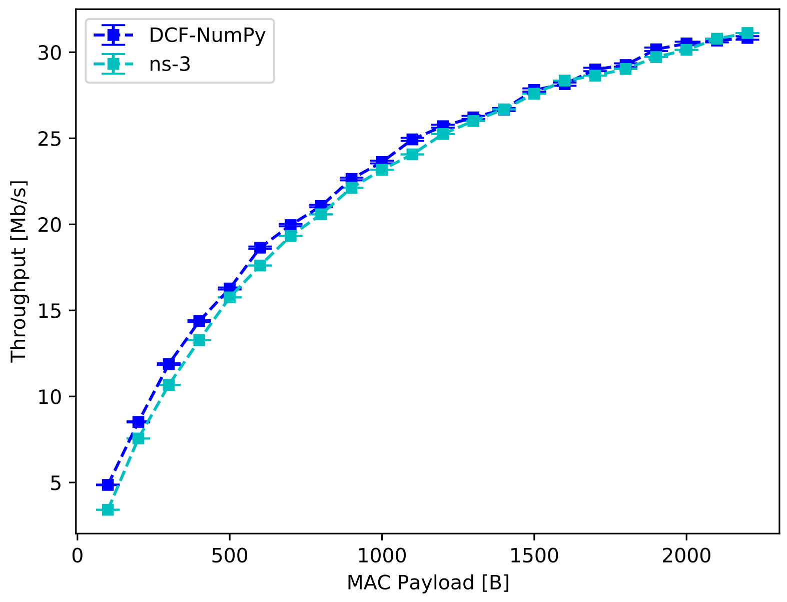

Figure 4.5: Network throughput vs packet size

are shorter (as the frame duration is shorter) and consequently there are more of them. This results in more channel access procedures, during which no user data is sent.

4.6. Impact of MCS

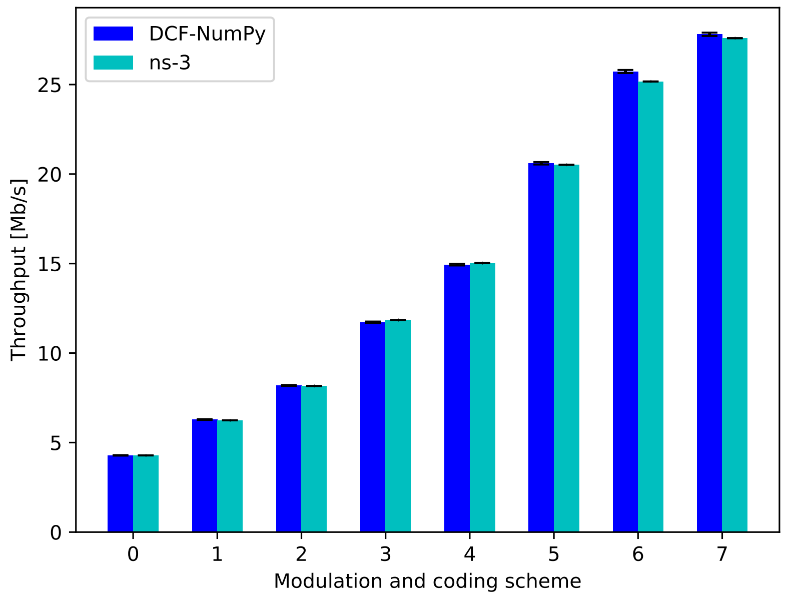

Figure 4.6: Network throughput vs MCS

This section explores the impact of Modulation and Coding Scheme – MCS – on the aggregate network throughput. Similarly as in Section 4.5, the throughput for different MCS values from ns-3 and DCF-NumPy simulators was compared. The results are presented in Figure 4.6. There are 8 different MCS values (from 0 to 7) defined in the IEEE 802.11a standard. Each of the values determines modulation and coding used in the transmission, which affect the rate at which data is transferred. In this scenario, all 8 MSC values were tested for the data rate, while control rate was constant and equal to 24 Mb/s (MCS = 4). It can be seen that results from ns-3 and DCF-NumPy are very close to each other and, as expected, the throughput is bigger for the higher MCS values. It is important to note that we assume perfect channel conditions, therefore it was possible to achieve high throughput for higher modulation schemes.

4.7. Performance Analysis

After validating the DCF-NumPy simulator and checking that it is working correctly, a basic performance analysis was conducted. It consisted of a simple scenario, which presents how the simulator can be easily used in some studies or research. Second part of the performance analysis was comparing the simulation execution time between the ns-3 simulator and the DCF-NumPy. These two analyses are to ensure that the objectives of DCF-NumPy, for it to be relatively easy and fast to run, were achieved.

4.7.1. Example Usage

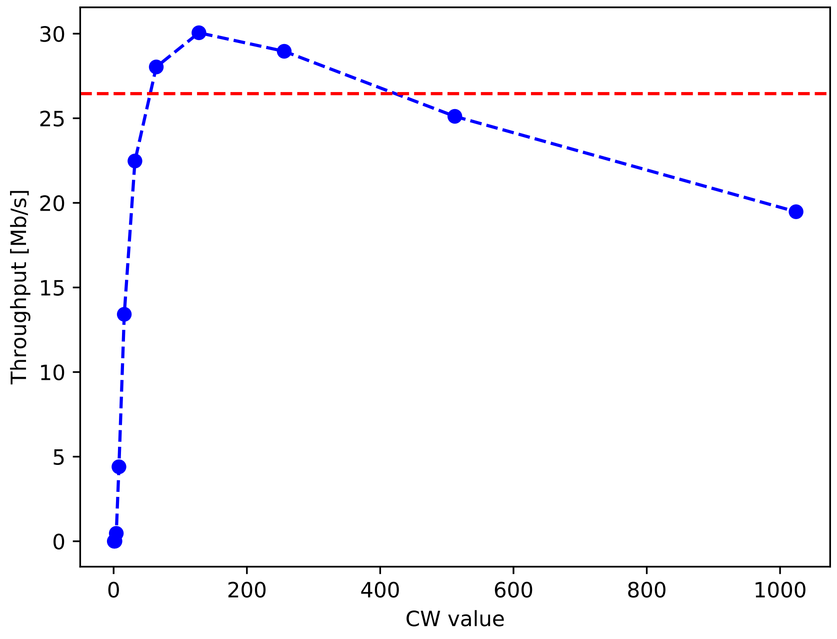

Figure 4.7: Throughput vs CW value for 16 contending stations

The goal of the performed example was to find an optimal value of CW for 16 contending stations. A series of DCF-NumPy simulations were performed for CW values ranging from 0 to 1024. In this scenario $CW_{min} = CW_{max} = CW$. For the consecutive values of CW the aggregate network throughput was recorded and compared to find the most optimal value of CW. Th results of this study are presented in Figure 4.7. The red horizontal line is a baseline throughput value obtained for 16 contending stations with the default CW_{min} and CW_{max} values, i.e., 15 and 1023. It can be seen that for this specific number of contending stations the optimal CW value is around

127 ($2^7 - 1$). For this value of CW the average network throughput is around 30 Mb/s. This, in comparison with 26.46 Mb/s, which is the throughput achieved with default CW values, is an improvement by around 12.9%.

4.7.2. Simulation Performance

In order to prove the time efficiency of the created simulator, a simple simulation was performed in ns-3 and in DCF-NumPy. The execution times of simulations in both simulators were recorded using Linux's `time` utility and compared with each other. The parameters of the simulations used for this analysis were the same for both simulators and were equal to its default values (presented in Figure 3.1), except the number of contending stations, which was equal to 50. The time of a single simulation was set to 50 seconds and the simulation was run 10 times in each simulator. The mean execution times were then calculated to ensure more stable results. The results of the comparison are presented in Figure 4.8. It can be seen that the execution time of DCF-NumPy is significantly lower than the one of ns-3. The mean execution time of a simulation in DCF-NumPy is around 0.8 s, while in ns-3 around 914 s, which is over 1000 times more than in DCF-NumPy. For this reason the graph had to be presented in a logarithmic scale, so that it can clearly exhibit results from both simulators. This is because the DCF-NumPy simulator implements a much simpler, and as a consequence less precise, approach towards the DCF function. However, for simple use cases, like the one presented in Section 4.7.1, this is a big advantage of DCF-NumPy over the ns-3 simulator.

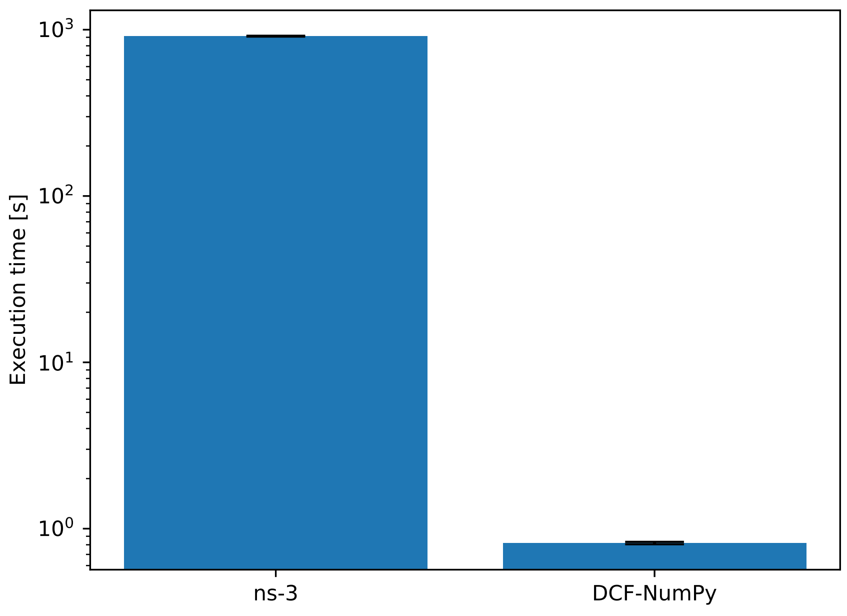

Figure 4.8: Simulation execution time comparison in a logarithmic scale

5. Summary

The main objective of this thesis was to implement a channel access simulator of the IEEE 802.11 network, taking into consideration some initial assumptions stated at the beginning of the work. The goal was achieved and a basic simulator was created with the help of the Python NumPy library. The second part of the work consisted of comparing the results and statistics obtained from the created simulator with the results from other simulators and models. For that part, among others, the ns-3 simulator was utilized.

Chapter 3, describing the implementation of the simulator, shows how the functionality of the DCF function and BEB algorithm can be relatively easily realised with the help of the NumPy library, mainly the NumPy arrays and functions operating on them. Implementing the channel access mechanism also allows us to take a close look into consecutive steps taken in the procedure.

From the validation part of the work we can safely assume that the created simulator is valid and properly implements the IEEE 802.11 channel access procedure. Results obtained from different models and simulators, configured to fulfil the initial assumptions, overlap with the results from the DCF-NumPy simulator. However, it can be observed that there are still some minor differences between results from the ns-3 simulator and the DCF-NumPy simulator, created in this work. This shows the effect of small aspects, not implemented in the DCF-NumPy simulator to simplify its structure, on the network performance. Additionally, throughout the whole work, it was assumed that the perfect channel conditions were granted. This situation never happens in a real world scenario and therefore the results from this simulator can only serve as a guideline (upper bound) when comparing with the real network results. The chapter also presents a simple exemplary use case of the created simulator. It displays how the simulator can be utilized for some basic research or studies. At the end of the validation chapter the time efficiency of the DCF-NumPy simulator was proved by means of a simple comparison with ns-3.

As mentioned, some simplifications were made in the process of implementing the simulator. This was done to ensure that the creation of the simulator was an achievable goal for this thesis. Therefore there are many ways in which the discussed simulator could be furthered developed. Some of the possible ideas for future work in this topic are the following. The simulator could provide support for other IEEE 802.11 standards than 802.11a. Furthermore, the EDCA function could be examined and implemented in order to provide the prioritized channel access mechanism and support for QoS. Additionally, the RTS/CTS, aggregation and fragmentation mechanisms could be implemented to allow more various network configurations and different scenarios. Another aspect could be an implementation of a real channel conditions, which are never perfect in a real network as we assumed for the purpose of this work. This is a complicated task, as those conditions are dependent on many factors and are not easily determined. However, this would allow to compare the created simulator with a real working WiFi network, which would be a very interesting study.

References

[1] G. Bianchi. “Performance analysis of the IEEE 802.11 distributed coordination function”. In: *IEEE Journal on Selected Areas in Communications* 18.3 (2000), pp. 535–547.

[2] Matthew S Gast. *802.11 Wireless Networks: The Definitive Guide, Second Edition*. O’Reilly Media, Inc., 2005. ISBN: 0596100523.

[3] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585 (2020), pp. 357–362. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[4] “IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”. In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016), pp. 1–3534. DOI: `10.1109/IEEESTD.2016.7786995`.

[5] Katarzyna Kosek-Szott, Alice Lo Valvo, Szymon Szott, Ilenia Tinnirello, and Pierluigi Gallo. *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]`.

[6] *ns-3 about page*. URL: `https://www.nsnam.org/about/`. accessed: 20.10.2020.

[7] Gjermund Raaen. *The WiFi AirTime Calculator*. URL: `https://gjermundraaen.com/thewifiairtimecalculator/`. Accessed: 2020-10-20.

[8] Kamil Słowik. “Performance Evaluation of IEEE 802.11n/ac Networks with Frame Aggregation”. MSc thesis. Poland, Krakow: AGH University of Science and Technology, 2017.

A. DCF-NumPy Simulator Source Code

Code created as part of the thesis.

```
# simulator created as an engineering project
# author: Cecylia Borek <cecylia.borek@gmail.com>

import numpy as np
import math


def dcf_simulation(N, cw_min=15, cw_max=1023, seed=1, data_rate=54, control_rate=24,
    mac_payload=1500, debug=False, sim_time=100):
    """Simulates DCF function as method of multiple access in 802.11 network
    and returns mean probability of colliosion per station

    Arguments:
        N (int): number of stations contending for the wireless medium
        cw_min (int): value of CWmin of BEB algorithm
        cw_max (int): value of CWmax of BEB algorithm
        seed (int): seed for the numpy random generator, allowing to reproduce the simulation
        data_rate (int): rate at which data is transmitted in Mb/s (one of the defined in
    802.11a standard),
            default 54 Mb/s
        control_rate (int): rate at which control data is transmitted in Mb/s, default 6 Mb/s
        mac_payload (int): payload of MAC frame in B, maximally 2304B, default 1500B
        debug (bool): if set to true additional info returned (simulation time, number of
    successful and
            unsuccessful transmission attemptss), default to false
        sim_time (float): simulation duration in seconds

        Values cw_min and cw_max should be the powers of 2 minus 1, i.e. 15, 31...1023

    Returns:
        Results: class that has two fields - aggregate network throughput in b/s and mean per
    station
            probability of collision
        (all_backoffs): returned if debug set to true, array with all backoff values selected
    throughout simulation
        (debug_info): returned if debug set to true, tuple in the form of (successful,
    collisions, simulation time)
    """

    contention_rounds = 10000
    retry_limit = 7
    slot_time = 9e-6  # s
```

```
successful = np.zeros(N)  # successful transmissions per station
collisions = np.zeros(N)  # collisions per station
retransmissions = np.zeros(N)  # counter of retransmissions per station
cw = np.ones(N) * (cw_min + 1)
# table of current CW for each station, changed after collisions
# and reset back to cw_min on success, contains upper excluded
# limits, i.e. 16, 32,..., 1024

tx_time = np.zeros(contention_rounds)  # times of all contention rounds
throughput = np.zeros(N)  # throughput per station

np.random.seed(seed)  # setting the seed of PRN generator

# random backoff for each station
backoffs = np.random.randint(low=0, high=cw_min+1, size=N)

# initializing list of all backoff values throught the simulation
all_backoffs = backoffs

for round in range(contention_rounds):
    if(np.sum(tx_time) >= sim_time / slot_time):
        break
    collision = False  # variable determining if collision occured or not,
    # necessary for transmission time calculation
    min_backoff = np.amin(backoffs)  # find the minimal value of backoff
    next_tx = np.where(backoffs == min_backoff)[0]  # an array of index(es)
    # of station(s) with lowest backoff(s)
    backoffs = backoffs - min_backoff - 1  # subtract from all stations' backoffs,
    # time they've already waited
    if len(next_tx) == 1:  # only one station had smallest backoff - success
        successful[next_tx] += 1
        retransmissions[next_tx] = 0
        cw[next_tx] = cw_min+1
        # selecting new backoff for the stations
        backoffs[next_tx] = np.random.randint(low=0, high=cw[next_tx])
        # appending the newly selected backoff to the array of all backoffs
        all_backoffs = np.append(all_backoffs, backoffs[next_tx])
    else:  # more than one station with smallest backoff - collision
        collision = True  # collision - True, necesarry for transmission time calculation
        for tx in next_tx:
            collisions[tx] += 1
            if retransmissions[tx] <= retry_limit:
                retransmissions[tx] += 1
                cw[tx] = min(cw_max+1, cw[tx]*2)
                # cw is always the upper limit (excluded), therefore we only need to
                # multiply it by two to get the next value of cw limit
            else:  # if retry limit is met, cw and retransmissions couter are reset
                retransmissions[tx] = 0
                cw[tx] = cw_min + 1
            # new backoff chosen for all the station that collided
            backoffs[tx] = np.random.randint(low=0, high=cw[tx])
            # appending the newly selected backoff to the array of all backoffs
            all_backoffs = np.append(all_backoffs, backoffs[tx])
    # calculation of round time in slots
    tx_time[round] = transmission_time(
        min_backoff, data_rate, control_rate, mac_payload, collision)['tx_time']
    simulation_results = Results()

    # calculation of collision probabilty per station and mean value of it
    collision_probability = collisions/(successful + collisions)
    simulation_results.mean_collision_probability = np.mean(
        collision_probability)

    # calculation of throughput per station in b/s and aggregate throughput of whole network
    simulation_time = np.sum(tx_time) * slot_time
    throughput = successful * mac_payload * 8 / (simulation_time)
    simulation_results.network_throughput = np.sum(throughput)

    # debug info
    debug_info = (np.sum(successful), np.sum(collisions), simulation_time)

    if(debug):
        return simulation_results, debug_info, all_backoffs

    return simulation_results


def transmission_time(backoff_slots, data_rate, control_rate, mac_payload, collision):
    """Calculates the time of single round of contention in dcf

    Args:
        backoff_slots (int): number of slots of backoff period in given round
        data_rate (int): rate at which data is transmitted in Mb/s (one of the defined in
    802.11a standard)
        control_rate (int): rate at which control data is transmitted in Mb/s
        mac_payload (int): payload of MAC frame in B, maximally 2304B
        collision (Bool): boolean value indicating if collision, in given round, occured or
    not;
            True if there was a collision, False otherwise
    Returns:
        results (str:int): dictionary with duration of contention round and duration of mac
    frame
    """

    # dictionary: (data rate, bits per symbol)
    bits_per_symbol = dict([(6, 48), (9, 48), (12, 96),
                            (18, 96), (24, 192), (36, 192), (48, 288), (54, 288)])

    results = {}

    slot_duration = 9e-6  # s
    sifs = 16e-6  # s
    difs = sifs + 2 * slot_duration  # s

    ofdm_preamble = 16e-6  # s
    ofdm_signal = 24  # bits
    ofdm_signal_duration = ofdm_signal/(control_rate * 1e6)  # s
    service = 16  # bits
    tail = 6  # bits

    # ack frame
    ack = 14*8  # bits
    ack_duration = ofdm_preamble + ofdm_signal_duration + \
        (service + ack + tail)/(control_rate * 1e6)  # s

    # data frame
    mac_header = 36*8  # bits
    mac_tail = 4*8  # bits
    mac_frame = mac_header + mac_payload*8 + mac_tail  # bits
    padding = (math.ceil((service + mac_frame + tail) /
                         bits_per_symbol[data_rate]) * bits_per_symbol[data_rate]) - (service
    + mac_frame + tail)  # bits

    data_duration = ofdm_preamble + ofdm_signal_duration + \
        (service + mac_frame + tail + padding)/(data_rate * 1e6)  # s

    tx_time = difs + data_duration  # s
    # adding sifs and ack duration only if the transmission was successful
    if not collision:
        tx_time += sifs + ack_duration  # s
    # adding ack timeout (2*sifs) in case of collision
    else:
        tx_time += 2*sifs  # s

    tx_time_slots = math.ceil(tx_time/slot_duration) + backoff_slots  # slots

    results['tx_time'] = tx_time_slots
    results['frame_time'] = data_duration

    return results


class Results:

    def __init__(self):
        self.mean_collision_probability = 0
        self.network_throughput = 0
```