Simulation of Channel Access Schemes

1 Introduction

The foreseen increased use of Short Range Devices (SRDs) will boost the competition for radio channel access in license-exempt frequency bands. Therefore, it is important to find out the most efficient SRD configurations in terms of their performance in high contention scenarios. Given the complexity of radio access schemes and the great variety of parameters on which they are dependent upon, analytical analysis is either very limited due to a number of necessary simplifications or not feasible at all. Thus, an alternative is to analyze the performance of channel access schemes through a Monte Carlo simulation.

This document describes a simulator developed using the C programming language in order to model the behavior of a sensor network; moreover, the simulator enables to assess two channel access mechanisms, namely Duty-Cycling (DC) and Carrier Sense Multiple Access (CSMA), in an independent manner and in coexisting scenarios. This document firstly presents the simulation scenarios and some parameter values from the literature that were used to model the mentioned scenarios, followed by the description of the simulator's structure and implementation.

2 Channel Access Schemes

The channel access techniques adopted by SRDs directly impact the overall performance. Generally, channel access schemes either adopt a contention-based or schedule-based approach. In the former, SRDs compete for channel usage in an independent and random manner, which usually leads to failure of some communication due to interference and packet collision among nodes. Regarding schedule-based approaches, also known as collision-free schemes, the nodes have resource (time or frequency) slots assigned to them (by a central control unit) in order to avoid collisions within the SRD system (note that schedule-based approaches are still vulnerable to interference from external sources).

Even though contention-based approaches are not immune to transmission collisions among different

nodes within the same system, these schemes are simpler to implement. Therefore, SRDs adopt mostly contention-based techniques and, for this reason, only this approach is considered henceforth.

In the following, the channel access schemes implemented in the simulator, namely DC and CSMA, are described. They were chosen due to their popularity among contention-based techniques, especially because of the simplicity of the DC mechanism and the "politeness" of the CSMA protocol.

2.1 DC

Duty-cycling (DC) is the simplest random channel access technique: its name derives from the fact that SRDs are only active for a very short period of time (usually a few ms), periodically — they spend almost no energy during the remaining time, as nodes enter in a sleep mode state. A survey made by ECC [1] analyzed the use of the European SRD band (863-870 MHz) and concluded that DC is also the most widely used channel access scheme. The duty cycle value associated with this scheme (D_c) denotes the percentage of time that an SRD is transmitting within a cycle,

$$D_c = \frac{T_p}{T_{cycle}}, \tag{1}$$

where T_p stands for the packet transmission time and T_{cycle} corresponds to the packet generation period (at the application layer).

The DC scheme does not take into consideration whether or not the channel is occupied prior to a transmission. Moreover, since this protocol does not perform any retransmission if a collision takes place, this occurrence always leads to a packet loss. Therefore, the Packet Loss Rate (PLR) of a DC system, considering that all nodes N in the network can cause collisions on one another, can be computed as [2]

$$PLR = 1 - (1 - 2D_c)^{N-1}. \tag{2}$$

2.2 CSMA

Carrier Sense Multiple Access (CSMA) techniques adopt a Listen Before Talk (LBT) approach: prior to a transmission, the channel is sensed (throughout a certain listen period of time, T_L) and, if any activity is detected, the transmission is postponed. [3] The LBT approach is regarded as a polite access technique, since it only makes use of the channel when it is free. Thus, this type of approach enables a more efficient use of radio resources, namely by allowing additional devices to opportunistically access the channel while avoiding collisions with ongoing transmissions.

Nonetheless, the use of the CSMA protocol may still lead to situations where there is a collision event:

- An SRD identifies an occupied channel if it senses the carrier during at least the minimum response time, T_R; the SRD commutation from listen to transmit mode also requires a certain amount of time, known as "dead" time, T_D. Therefore, an SRD is unable to recognize another node transmission if it starts during the SRD "dead" time or even up to T_R prior to this "dead" period.
- If two SRDs want to send data to the same node, but are too distant to detect each other's transmissions, their packets might collide at the receiver — this is commonly known as the *hidden node* problem and significantly decreases the CSMA performance.

Another situation that leads to a degradation of the CSMA performance is the *exposed node* problem, where a transmission is prevented when it would otherwise have gone ahead without any problem, thus reducing the spectral efficiency — this situation occurs when an SRD senses remote transmissions and identifies the channel as occupied, but the SRD transmission would not suffer a collision from these remote transmissions.

There are different approaches regarding the decision that a node makes when performing the transmission retrial, resulting in different CSMA versions. [3] The most prevalent is the nonpersistent-CSMA (np-CSMA): if an SRD detects a busy channel, it defers a new sensing to a later time by a random time, T_{Rep}. A less "polite" version is the 1-persistent CSMA, in which a device that senses the channel as busy, waits until the channel is free and then immediately starts transmitting (this means that if two or more devices are waiting for the channel to be free, they will interfere inevitably). The CSMA scheme can also be improved by the use of an acknowledgement packet (ACK) regarding a successful transmission. After sending a data packet, the transmitter waits for an ACK during a certain amount of time, $T_{TimeoutACK}$, and if it does not receive any ACK, a retransmission is performed after a random period of time, $T_{TimeoutRep}$.

The np-CSMA technique plus the ACK feature are adopted herein.

3 The Monte Carlo Method

Given the random nature of some input and internal variables of the simulator – the sensors' positions and the retransmission time intervals, for instance – the simulation results are also random. Consequently, to obtain statistically relevant results, the Monte Carlo method is used, by running the simulation a given number of times to derive a final result that includes a mean value and a confidence interval [4]. Simply put, given the random nature of the model, each individual simulation generates a result with an unknown accuracy. However, if a large number of simulations is run, we obtain instead a set of solutions that can

provide a statistical representation of the real result.

According to this computational technique, a random process with random inputs (such as the placement of the network nodes) generates an output that is itself a random variable which can be characterized by analyzing a set of its samples.

The calculation of the result is based on the Central Limit Theorem [5]. Consider a random variable X with finite mean, μ, and variance, σ^2. The theorem states that the arithmetic mean of a sufficiently large set of samples of the random variable, $\mathbf{X} = (X_1, X_2, ...)$ of size s samples, defined by

$$\hat{X} = \frac{1}{s} \sum_{i=1}^{s} X_i, \tag{3}$$

approaches the random variable's mean as the number of samples, n, approaches infinity:

$$s \to \infty \Rightarrow \hat{X} \to \mu. \tag{4}$$

The sample mean is therefore the most probable solution, while the standard deviation defines a given confidence interval. The higher the number of samples (i.e. the higher the number of simulation runs), the smaller the confidence intervals, which translates into a greater accuracy of the solution. The minimum number of simulations, S_{min}, to achieve a given targeted accuracy can be determined by the following expression [6]:

$$S_{min} = \left(\frac{z_{\alpha/2}}{w} \frac{\sigma}{\mu} \right)^2, \tag{5}$$

where $z_{\alpha/2}$ is the value of the normal probability distribution function for the half distance $\alpha/2$ and w is the size of the confidence interval, normalized to the mean.

A typical choice is to consider a maximum confidence interval size of 10% of the mean ($w = 0.1$), with a probability of 95% (corresponding to $z_{\alpha/2} = 1.96$, as illustrated in Figure 1). This means that if the program is run S_{min} times, a 95% certainty that the real solution is in the interval $[\hat{x} - w\hat{x},\ \hat{x} + w\hat{x}]$ is obtained. This is somewhat a chicken-egg problem, since in order to obtain the number of simulations, one must already know μ and σ. However, this can be solved iteratively. Firstly, an arbitrary large number of Monte Carlo simulations is used for each value of the number of sensors in the network [1]. After the simulations are run, the resulting μ and σ values for each number of simulations, S, are used

[1] To minimize the simulation time, lower S values were used for larger numbers of sensors, since fewer simulations are effectively necessary

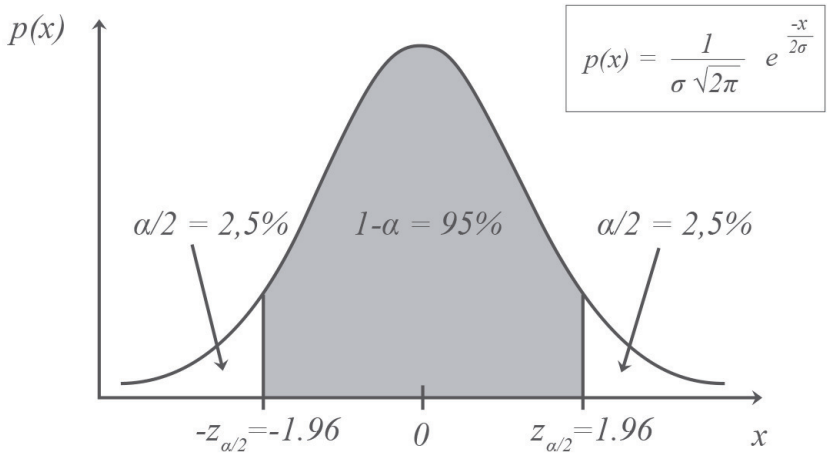


Figure 1: Illustration of 95% confidence interval in the normalized normal distribution.

to compute the minimum number of simulations necessary, S_{min}, using Equation (5). If S_{min} is larger than the number of simulations performed, the program is ran again – this time for S_{min} simulations; if not, the confidence interval requirements are met and no further simulations are needed. The S_{min} computations are performed for all output variables – and the confidence interval must be met by all of them. Therefore, if $S < S_{min}$ for at least one of the variables, the simulation needs to be re-run.

4 Simulation Scenarios

The simulation can be run in two different scenarios, named **reference** and **real** scenarios.

The reference scenario is an ideal one, where path loss is not considered, which means that all sensors are within range of one another and if two transmissions occur simultaneously, a collision always occurs. This model is implemented in order to enable the study of the access schemes theoretical behavior and to have a framework for comparing the results with the ones in the related literature.

The real scenario emulates a sensor network in a close-to-real indoor environment. In this way, it considers path loss due to signal propagation and noise. Due to the typical short distances between devices in sensor networks (up to some dozens of meters) however, the propagation delay is assumed to be zero. More specifically, this scenario models a building automation system in an open space office environment. In this type of network, sensors and actuators are used to monitor and operate, for instance, electrical and air conditioning systems, electric shutters and different types of alarms (e.g., fire/smoke and intrusion) [7].

When initializing the simulation, and when using the real scenario, network nodes are randomly placed in the environment and path loss and interference between nodes are computed, in order to be used throughout the simulation. Note that these informations are irrelevant for the case of the reference model,

and are therefore not computed.

4.1 Network Topology

For all the access schemes that can be simulated, a star (or one-hop) topology is considered, due to its simplicity and independence of routing algorithms. In a nutshell, all sensors are directly connected to a gateway[2], which is assumed to power plugged, and therefore not energy limited. In this way, gateways are constantly listening to channel activity, while sensors compete for access to the channel by using a given channel access scheme.

In the reference model case, all nodes are connected to the same gateway, which is conceptually the same as having different gateways, because interference from one node affects all nodes equally, resulting in packets collision whether they are being sent to different gateways or the same one.

4.2 Node Placement

At the start of each simulation, sensors are placed randomly within the defined building area. On the other hand, gateways are assumed to have fixed positions for all simulations (which emulates, for instance, a scenario where low-range gateways are incorporated into lamps placed on the ceiling). Considering that in real cases, gateway deployment is planned so as to guarantee coverage in the entire site, the building area was divided into sectors and within each one, a gateway is placed in the center. Sensors communicate with the gateway they are closer to. An example of node placement can be found in Figure 2, where the gateways' ranges are represented by circles centered on the gateway position. The range is computed from the path loss equation (cf. (14)).

4.3 Interference and Path Loss

Values for interference and path loss for all possible pair of nodes are computed once at initialization, in order to be used throughout the simulation (eliminating the need to calculate them multiple times).

Noise and interference are the main factors that affect the quality of communication or, more specifically, packet collisions. The greater the distance between devices, the more dominant the effect of thermal noise becomes. Also, the higher the sensor density, the higher the impact of interference. A message can only be correctly received and demodulated if the signal to interference plus noise ratio (SINR) at the receiver is above a given threshold:

[2]In sensor networks, the term gateway is used to designate the node that collects traffic from several nodes [8].

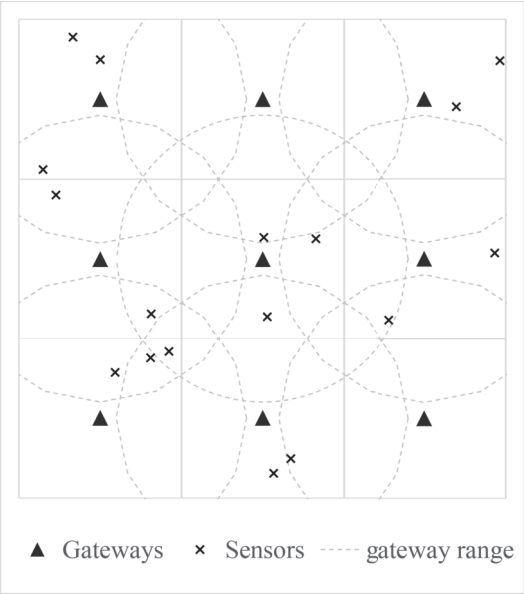


Figure 2: Network node placement example.

$$sinr > sinr_{min}. \tag{6}$$

The SINR is defined, in linear units, by the ratio between the useful signal power, p_r, and the sum of the interfering signal powers, $i_{r,k}$, and the noise power at the receiver, n:

$$sinr = \frac{p_r}{\sum\limits_k i_{r,k} + n}. \tag{7}$$

By manipulating inequality (6) and equation (7), one can obtain an expression for the maximum value of the total interfering power that allows for a correct reception of the signal:

$$\sum_k i_{r,k} < \frac{p_r}{sinr_{min}} - n. \tag{8}$$

Since we consider, for a given sensor, that p_r is constant and $sinr_{min}$ and n are characteristic of the receiver (and hence also constant), to evaluate if a packet collision occurred, one needs simply to evaluate at any given time if the current interference from all active transmitters is greater than this value – which will be denoted as interference limit, i_{lim}:

$$i_{lim} = \frac{p_r}{sinr_{min}} - n. \tag{9}$$

The signal power received by node i when node j is transmitting is calculated by dividing the power

emitted by node i, $p_{t,i}$, by the path loss between the nodes, pl_{ij} (note that this computation is valid for both useful and interfering signals):

$$p_{r,j} = \frac{p_{t,i}}{pl_{ij}}. \tag{10}$$

The path loss model for indoor environments is defined in ITU Recommendation [9], which estimates the total path loss PL, in dB, as:

$$PL_{dB} = 20\, log(f_{[MHz]}) + 10\, a\, log(d_{[m]}) + L_f(s) - 28, \tag{11}$$

where:

- f is the carrier frequency in MHz;
- a is the path loss exponent;
- d is the distance between the transmitter and the receiver in meters;
- $L_f(s)$ is the additional loss factor caused by the penetration of s floors or walls, in dB.

Both a and $L_f(s)$ values are provided in [9] and depend on the frequency and type of environment (residential, office or commercial). The office environment is adopted as test case in the simulations, with a path loss exponent of $a = 3.3$. Moreover, since a single floor is considered in the simulations, the floor penetration factor, L_f, equals zero.

One of the factors to consider when calculating a node's range is the receiver's sensitivity, P_{det}, which is the minimum signal power it can detect. Assuming a node i transmitting with power $P_{t,i}$ (dBW), at a distance d from the receiver j, the received power $P_{r,j}$ (dBW) is calculated from (10). In logarithmic units, equation (10) translates to

$$P_{r,j} = P_{t,i} - PL_{ij}. \tag{12}$$

To compute the nodes' range, r, the maximum path loss between nodes, PL_{max}, is calculated for a received power that equals the sensitivity, $P_r = P_{det}$, resulting in:

$$PL_{max} = P_t - P_{det}. \tag{13}$$

By solving equations (11) and (13) in d, the range can be obtained:

$$r = f^{-\frac{2}{a}}\ 10^{\frac{P_t + 28 - L_f - P_{det}}{10\ a}}. \tag{14}$$

Finally, noise power in the receiver is computed as the product of the input thermal noise power n_0 and the device noise factor, n_f:

$$n = n_0\ n_f. \tag{15}$$

In its turn, the thermal noise power, n_0, is dependent on the temperature and receiver's bandwidth. By multiplying the receiver's bandwidth B and the noise spectral density, $k\ T$, where k is the Boltzmann's constant and T the receiver's absolute temperature, the thermal noise power can be obtained:

$$n_0 = k\ T\ B. \tag{16}$$

Throughout the simulations, an ambient temperature of $T = 290\ K$, corresponding to normal temperature and pressure conditions (NPT), was assumed. A typical noise figure value for this type of devices of $N_f = 3$ dB was considered, corresponding to a noise factor of $n_f = 10^{N_f/10} \approx 2$ [10]. Finally, a bandwidth of $B = 200$ kHz, which is typical for SRD applications in this frequency band was used [11].

4.4 Additional Assumptions

Besides the already mentioned assumptions and scenarios, some additional and more general simplifications were considered:

- When two frames overlap in time and frequency, a packet collision occurs and both packets are completely destroyed. In reality, under some circumstances interfered frames can be recovered – totally or partially – at the receiver. However, this process is not considered.
- The only sources of signal degradation are interference and noise at the receiver.
- Signal processing and channel propagation delays are considered to be zero, since: 1) the amount of information to be processed is very small (packets are of the order of dozens of bytes); 2) the communication distances for SRDs are very small (as the term short range device itself indicates) [2].
- The nodes' buffer only keeps one packet in memory at a time, which means if a new packet arrives at the application layer and the previous one was still waiting to be transmitted, the older packet is

dropped and lost.

5 Simulator Implementation

The simulator was developed in the C programming language using an event-based approach: modeled actions, such as transmitting a data packet or probing channel activity, are represented by events that mark their start and end moments in time. Events are stored in a list, ordered by their occurrence time. Throughout the simulation, events are "pulled" from the list and, depending on the type of event, the correspondent actions are taken. Furthermore, to obtain statistically relevant results, the Monte Carlo method is adopted, which means that more than one simulation should be run to obtain results for any given scenario. All details concerning the Monte Carlo method, including the adequate number of simulations to be run and illustrative results, are presented in Section 3.

Apart from the event list, all other program's variables were grouped into four categories: **input parameters**, **status variables**, **counters** and **outputs**. Input parameters variables store all the information that was extracted from the input file during initialization; status variables have information regarding the current status of each node (including, for instance, the current received interference power and flags to indicate if the sensor is listening); counters store the current number of certain occurrences in each node (such as lost packets or received ACKs); finally, outputs store the results of the simulation and are computed using the counters' values, according to the definitions in Section 5.2. These categories comprise each several individual variables (and throughout the program's implementation, more were added as needed). Each category was implemented as a `struct` variable, in a "container-like" fashion, including all correspondent variables within the structure, which is illustrated in Figure 3, where some examples of variables can be found for each structure. By encapsulating variables in structures, extending the program's functionalities becomes easier, since variables can be added inside each structure without the need for changes in the program's `main` function (the complete listing of variables in each structure is detailed in Tables 2, 3, 4 and 6).

The choice of data types to be used was carried out considering the variable's function. Since the size of the event list varies constantly, a linked list structure is the most suitable choice in this case, since it allows for the dynamic allocation of memory for new events and the insertions of an event anywhere in the list with few instructions (if an array were to be used, the insertion of an element in the middle would imply the relocation of all following elements one step to the right). Nodes from the linked list have, as any standard linked list implementation, data fields and a pointer field with the address of the next node of the list. As summarized in Table 1, the data fields include: event time, event type (indicates, for instance if it is a new packet, the end of a packet or the start or end of the listening interval), receiver and

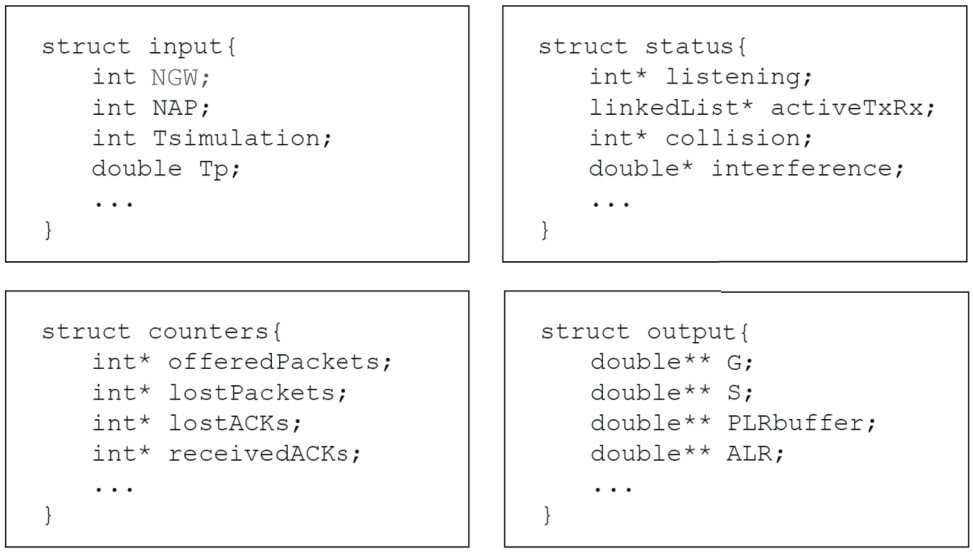


Figure 3: Structures used in simulator with examples of variables they contain.

Table 1: Fields of `eventList` structure.

`double`	`Te`	Time of event occurence
`int`	`T`	Transmitter ID
`int`	`R`	Receiver ID
`int`	`P`	Packet ID (binary)
`int`	`eventType`	Code for type of event
`eventList*`	`next`	Address of next node on list

transmitter identification (ID) and packet ID. In Figure 4, the `eventList` structure fields as well as an illustrative schematic of an event list can be found.

For the majority of other variables, when one value per sensor node is needed, one and two-dimensional arrays were used. In this way, for instance, information on whether the current packet has collided or not is stored as a one-dimension array of flags, each position of the array corresponding to a sensor node, as depicted in Figure 5, where an array of the current packet number for each node is represented. Path loss between any two nodes i and j, for instance, is stored in a two-dimensional array, where the line and column indexes correspond, in the same way as for the one-dimensional array, to the nodes' ID (as represented in Figure 6).

Counters and output variables are implemented with arrays. Status variables follow the same logic, with one exception. In the most simple approach, flags to indicate if a node is transmitting/receiving would be equally implemented with arrays. However, a different strategy was adopted: throughout the simulation, there is the recurrent and frequent need of checking all currently active transmitters/receivers. If this check were to be made on an array, all positions would have to be accessed, which represents a number of

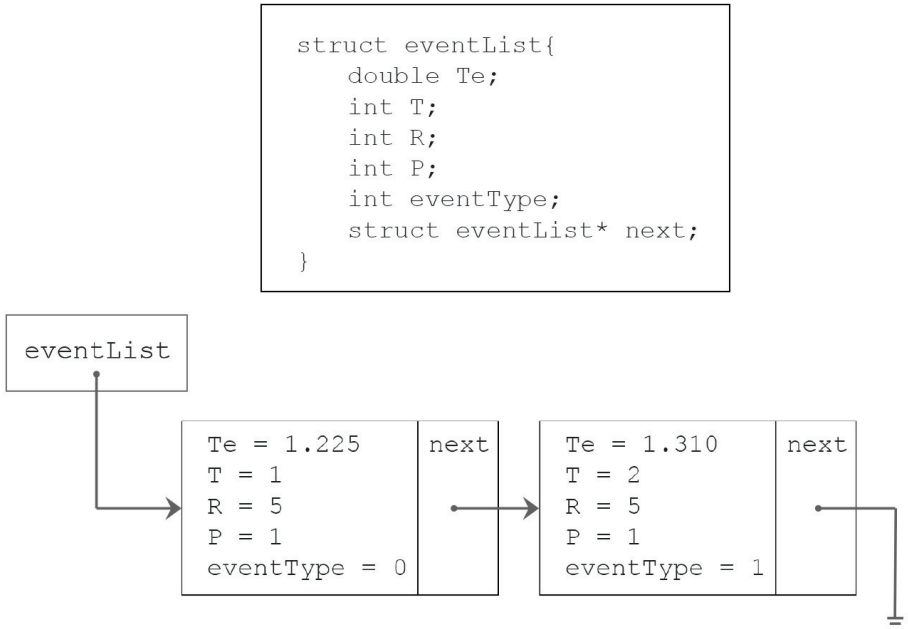


Figure 4: Event list structure and example.

Figure 5: Example of a one-dimensional array.

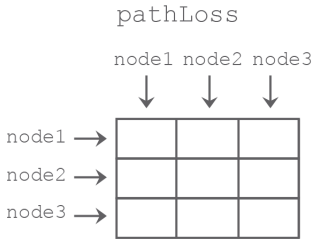


Figure 6: Example of a two-dimensional array.

operation proportional to the number of network nodes. However, given the high contention for accessing the channel that is characteristic of these networks, only very few nodes can actually be transmitting at the same time. Consequently, to make the process of finding active transmitters more efficient (and hence greatly decreasing simulation time), linked lists were used instead of arrays to store information about these nodes. Items in this list contain the transmitter and receiver IDs. At any time, the items on the list contain the active transmitter-receiver pairs. If the list is empty, no one is transmitting. One should note that this strategy, while maybe unnecessary for small networks, highly decreases simulation time for a greater number of network nodes.

All the status variables, as well as the data types that were used to implement them, are listed in Table 2.

As for the variables in the `counters` structure, they are all implemented with two-dimensional arrays, where each column corresponds to a node and each row corresponds to a different simulation, as represented in Figure 7, where the way the results are computed is also depicted.

5.1 Input Parameters

When the simulator starts, an input file containing the initialization values is read. The parameters initialized in this file are then stored in the `input` structure, as listed in Table 4. The number of sensor nodes, N_S, is the only variable that can take more than one value: if several values are inserted in the file, the simulation is run for all of them, the results for each value of N_S being stored in an array (as shown in Figure 7). This option was taken since the performance analysis is generally made by observing the behaviour of the network for different numbers of nodes.

While the network's behaviour was studied under the variation of some of the parameters in Table 4, some values were kept constant for all simulations. The values that were hence assumed (and some intervals) were taken from the literature and are summarized in Table 5. In terms of hardware, the Mica2 MpR400CB sensors [12] were used as a reference.

5.2 Output Parameters

The simulator produces a series of output parameters. Firstly, results are computed individually after each simulation, using the values from the counters. Finally, after all simulations have been run, the arithmetic mean and standard deviation of all the previous results are computed and stored in an output file for further treatment and visualization. This is performed for all values of N to be simulated, as shown in Figure 7. All the outputs produced by the simulator are listed in Table 6 and their computations defined as follows.

Table 2: Variables in `status` structure.

`int*`	`listening`	Flag array indicating if node is in listen mode
`linkedList*`	`activeTx`	Linked list containing current active transmitter-receiver pairs
`linkedList*`	`activeRx`	Linked list containing current active receivers
`linkedList*`	`activeLst`	Linked list containing current listening transmitters
`int*`	`collision`	Flag array indicating if packet being transmitted has collided (when in transmition mode) or if channel has been detected as busy (when in listening mode)
`int*`	`waitingACK`	Flag array indicating if node is waiting for an ACK after packet transmission
`int*`	`txLastPacket`	Flag array indicating if last application packet was correctly transmitted (i.e. if an ACK was received)
`int*`	`currentP`	Array with current application packet ID (binary)
`double*`	`channelBusySince`	Indicates time when activity was firstly detected on the channel
`double*`	`nextCycleStart`	Indicates time when next application packet will be available
`double**`	`sensorPositions`	Three-dimensional coordinates of sensors
`double**`	`GWpositions`	Three-dimensional coordinates of gateways
`double**`	`pathLoss`	Matrix with path loss between all pairs of nodes, computed from (11)
`double**`	`interference`	Matrix with interference power between all pairs of nodes, according to (10)
`double*`	`interferenceSum`	Current sum of all interfering signals power at the receiver
`double*`	`interferenceLimit`	Threshold for interfering power in each node above which there is a packet collision, as defined in (9)

Table 3: Variables in `counters` structure.

`int**`	`applicationPackets`	Number of packets generated by the application layer
`int**`	`offeredPackets`	Number of times the node tries to send a packet, either new or rescheduled (note that it does not mean the packet is actually sent, just listening to the channel is considered an attempt to send a packet)
`int**`	`transmittedPackets`	Number of packets that were effectively transmitted to the channel
`int**`	`lostPacketsCollision`	Number of packets that collided
`int**`	`lostPacketsBuffer`	Lost packets due to buffer overflow
`int**`	`receivedACKs`	Number of correctly received ACKs
`int**`	`transmittedACKs`	Number of ACKs that were transmitted
`int**`	`lostACKs`	Number of ACKs lost due to collision

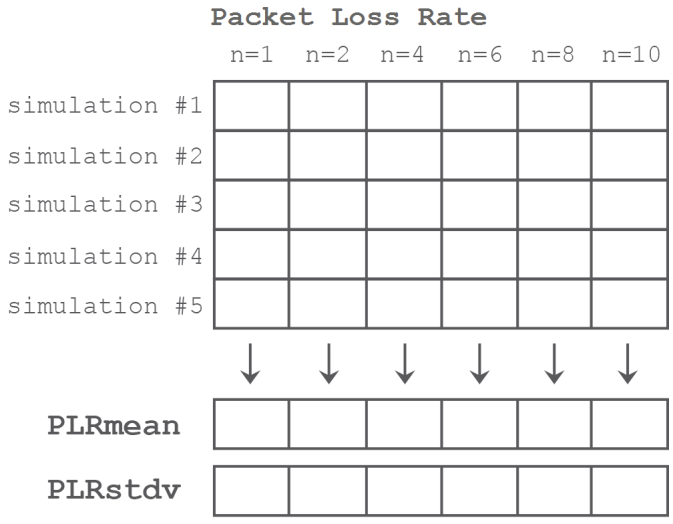


Figure 7: Example of computation of mean and standard deviation arrays from a results matrix.

Table 4: Variables initialized in simulator's input file.

Simulation Settings	
`S`	Number of Monte Carlo simulations
`Ncycles`	Duration of each simulation, in number of DC cycles
`scenario`	Scenario type (reference or real)
Network Parameters	
`NS`	Number of sensor nodes
`NGW`	Number of gateways
`DC`	Duty Cycle
`Tp`	Packet duration
`TrandomOffset`	Maximum random time offset to initiate listening after cycle start
ACK Parameters	
`TACK`	ACK duration
`TrespACK`	Time to receive an ACK after packet transmission
`TtimeoutACK`	Maximum time during which a receiver expects an ACK
`Ttimeoutrep`	Maximum time to retry transmission after not receiving an ACK
CSMA Parameters	
`TL`	Listen time
`TD`	Commutation time from listen to transmit mode
`TR`	Minimum time for channel activity detection
`Trep`	Maximum time interval for transmission retrial if channel is busy
Physical parameters	
`Pt`	Transmission Power
`Pdet`	Receiver Sensitivity
`SINRmin`	Minimum necessary SINR to correctly receive the packet
`N0`	Thermal noise power
`NF`	Receiver's noise figure
`f`	Carrier frequency
`a`	Propagation model path loss exponent
`lx`	Building's width
`ly`	Building's length

Table 5: Values used in all simulations.

Mica2 MpR400CB sensor [12]	
`f`	868 MHz
`Pt`	[-20, 5] dBm
`Pdet`	-98 dBm
`TL`	0.35 ms
`TD`	0.25 ms
`TR`	0.1 ms
`Tp` (1 byte)	0.416 ms
Typical parameters for a home/building automation application [11]	
packet size	small (up to 300 bit) to medium (up to 1 kByte)
data rate	small (up to 50 kbps)
duty cycle	low (up to 1%)
reliability (packet loss rate)	medium (up to 1%) to high (up to 0.1%)

To compute a final result after a simulation, first the individual results for each node are calculated and then all those values are averaged to produce the final result of the simulation.

The normalized offered load G, equals the percentage of the channel occupancy that would be necessary to send all offered packets (new and rescheduled) per gateway:

$$G = \#offeredpackets \, \frac{T_p}{T_{cycle} N_{GW}}. \tag{17}$$

The normalized throughput S, in its turn, is the ratio of the channel total capacity that is effectively used for successful transmissions per gateway, and hence can be calculated as:

$$S = \#receivedACKs \, \frac{T_p}{T_{cycle} N_{GW}}. \tag{18}$$

The packet collision rate (PCR) is simply the ratio of transmitted packets that were lost in collision:

$$PCR = \frac{\#collidedpackets}{\#transmittedpackets}. \tag{19}$$

The PLR indicates the ratio of packets that did not reach their destination. In case the system does not have an ACK mechanism (as in the case of DC, for instance), the transmitter has no way of knowing if the packet was lost in collision, so there are no retransmissions – the PLR in this case equals the PCR.

Table 6: Output parameters of the simulator.

`G`	Normalized offered load
`S`	Normalized throughput
`PLR`	Packet loss rate
`PCR`	Packet collision rate
`ALR`	ACK loss rate

On the other hand, if the receiver replies with ACKs, the transmitter can try to retransmit lost packets, and continues to do so until it either succeeds or a new packet arrives from the application layer. In the latter case – and assuming the buffer can only contain one packet – the packet from the last duty cycle is dropped due to buffer overflow and the PLR refers to the packets lost in these conditions, being calculated as:

$$PLR = 1 - \frac{\#receivedACKs}{\#transmittedpackets}. \tag{20}$$

5.3 Simulator Main Structure

The flowchart in Figure 9 illustrates the program's main function, which is common to all protocol implementations. The program starts by reading an input file with the simulation initialization parameters. Since most of the results are analyzed as a function of the number of sensors in the network, the option of running the simulator for several values of N_S was taken. All the N_S values to be simulated are stored in an array, `NSList`, and the simulations are run for each of them. The program finishes when the end of the array is reached. For each N_S, `S` simulations are run (the counter `nsimulated` registers how many simulations have been run to the moment) and for each simulation, the memory for all the variables is allocated upon the simulation initialization process and freed in the end. This initialization module also comprises the computation of the constant status variables: path loss, interference and interference limit values for all nodes (as detailed in Section 4.3) and the random placement of network nodes on the simulated scenario (as described in Section 4.2). After the initialization, the simulated time `Tsim` is reset. Every time an event is pulled from the event list, the simulation time is updated to the time of occurence of the event, `Te`. Afterwards, the function correspondent to the type of event is called. The mapping of the implemented events and their correspondent physical phenomena is depicted in Figure 8. It can be observed that each event corresponds to either a start or end of some action. The functions called for each event, although specific for the channel access schemes, share some common building blocks,

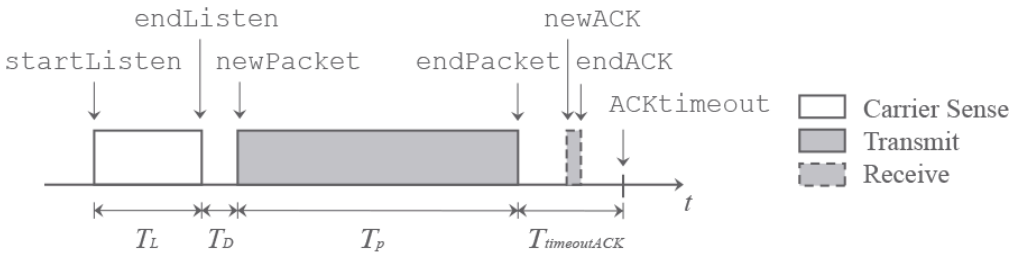


Figure 8: Simulated events and correspondent physical phenomena.

which are first detailed in the present section and represented in further flowcharts as a single block, for simplification and to avoid repetition. Note that some events are not applicable to all access schemes (DC, for instance, only has `newPacket` and `endPacket` events).

The simulation ends when a predefined time limit, `Tmax`, is exceeded. This time limit is given at input as a number of DC cycles (so that it can be independent of the cycle/packet duration) and is then translated to the correspondent value in time units (ms). To assure a good accuracy of the results while still guaranteeing a feasible simulation duration, a value of 10 000 simulated packet generation cycles was used.

Check New Cycle

In this section the modelling of the behaviour of dropping a packet when the buffer overflows is detailed.

The application layer generates packets periodically. After receiving an application packet, the transmitter waits for a random amount of time (with the maximum value of T_{offset}) before attempting to send it, to avoid cyclic conflicts between transmissions. At the beginning of the simulation, the "starting point" for each sensor is randomly placed within the first cycle. With these starting points as reference, the variable `nextCycleStart` contains, at any moment, the time when the next application packet will be received (or, in other words, the starting time of the next cycle). Whenever a new cycle is reached, if the previous packet had not been successfully transmitted, it is dropped – all further events related to the previous packet that might already be scheduled will not be considered and the correspondent `lostPacketsBuffer` counter is incremented.

Physically, deleting the previous packet would happen exactly upon the arrival of the new one. To emulate this behavior, at the beginning of every event the simulator checks if a new cycle has been reached (i.e. if there is a new application packet), by checking if the event time `Te` is larger than the current `nextCycleStart`. If so, it proceeds to updating the cycle-related information, as depicted in the flowchart of Figure 10 (and represented as a single block herin for better and simpler visualization of

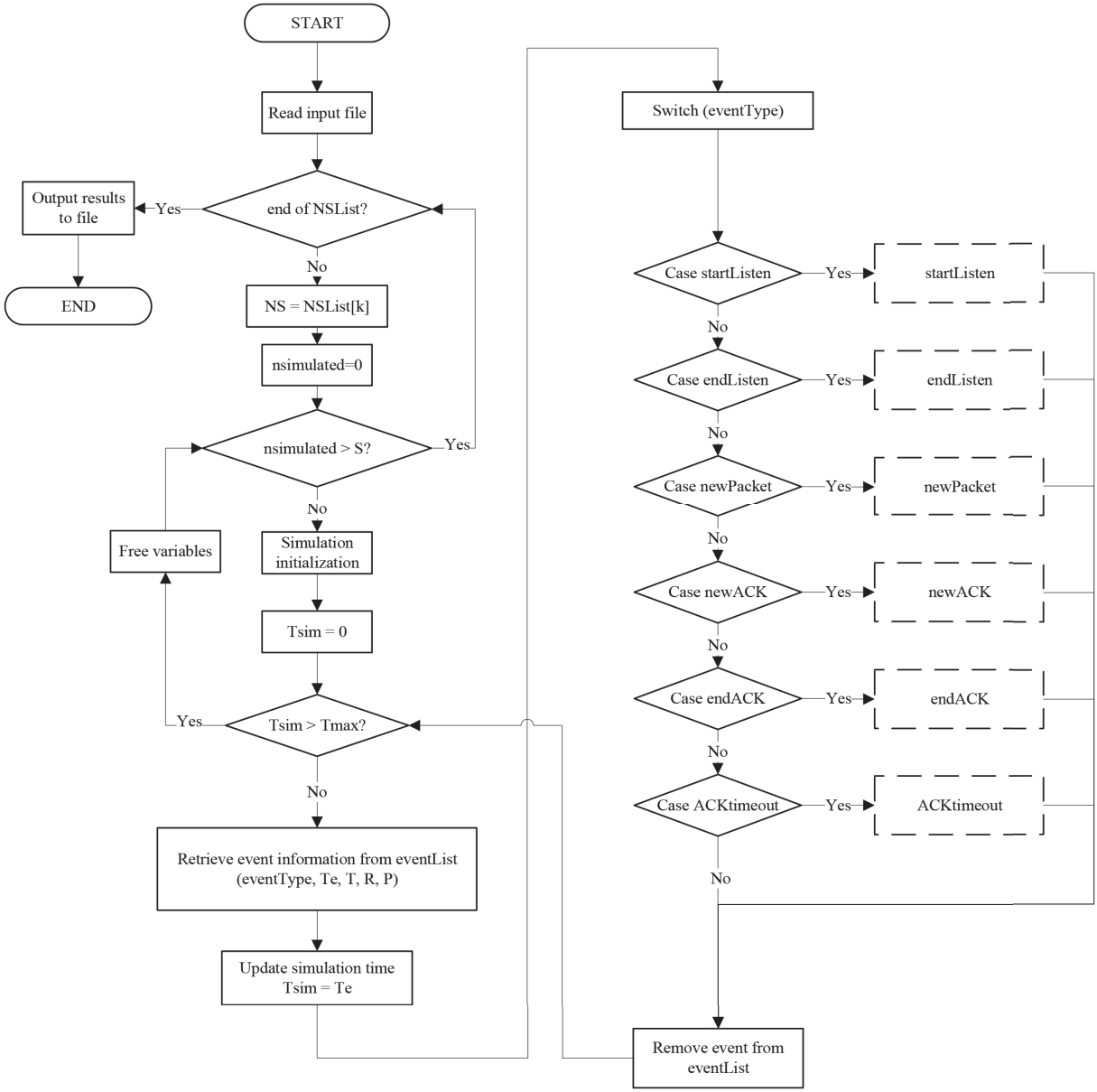
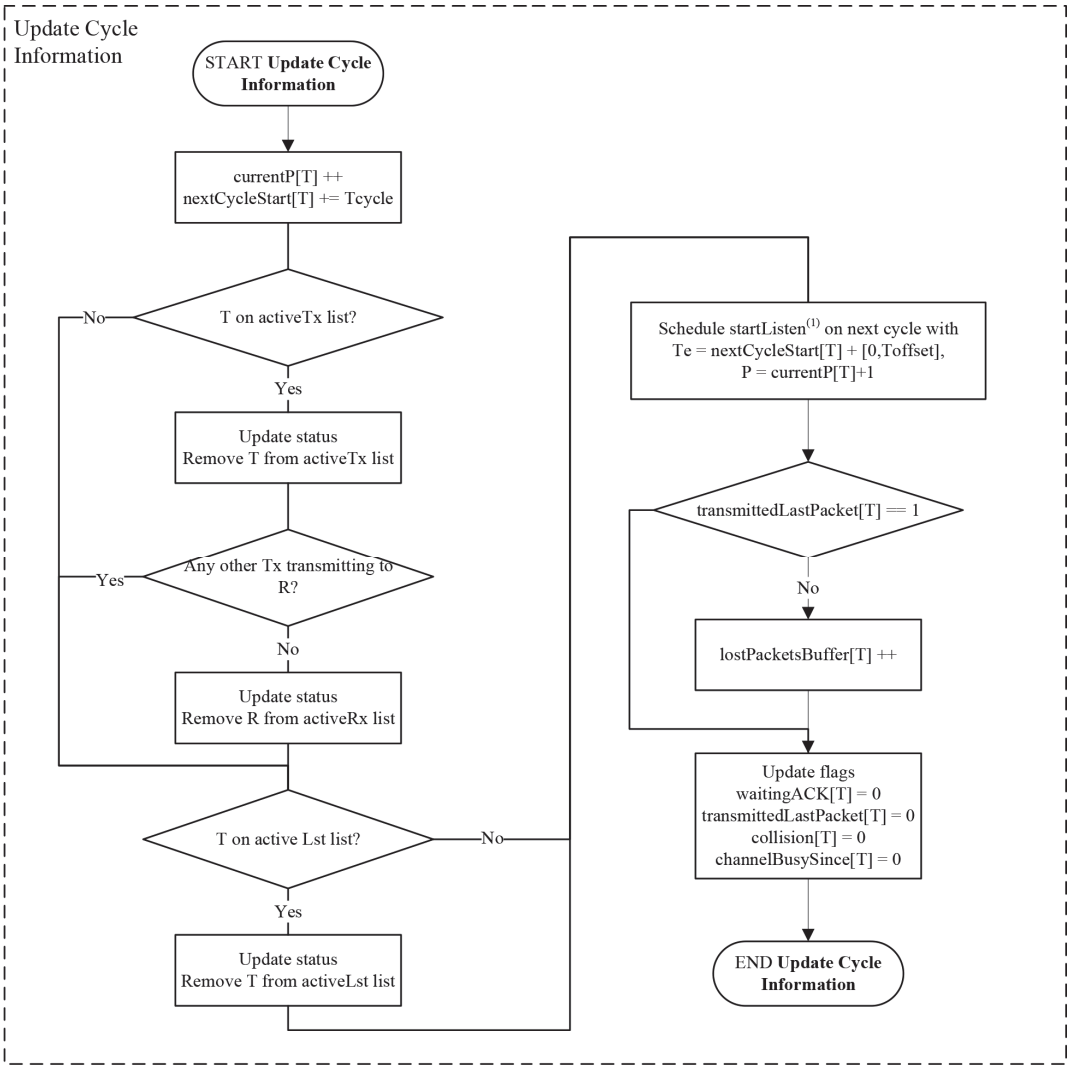


Figure 9: Simulator main flowchart.



(1) In the case of CSMA. For DC, schedule a newPacket evert instead.

Figure 10: Check new cycle flowchart.

the flowcharts). It starts by incrementing the current packet ID, `currentP`[3], followed by summing `Tcycle` to `nextCycleStart`. It then increments the `lostPacketsBuffer` counter if the previous packet has not been sent and finishes with the scheduling of the first `startListen` event of the next cycle.

Additionally to checking for a new cycle, the first verification to be made is if the current packet ID, `P`, equals the current application packet ID, `currentP` – if not (and the cycle has not changed), it means the packet was dropped and hence the event is outdated and, consequently, deleted without being processed.

To better illustrate the procedure, consider the example in Figure 11 for a CSMA system. The channel listening and packet transmission are shown, with the respective packet number. In the first cycle, an

[3]Only one bit is necessary for the packet number, and hence it alternates between 1 and 0 to differentiate the packets.

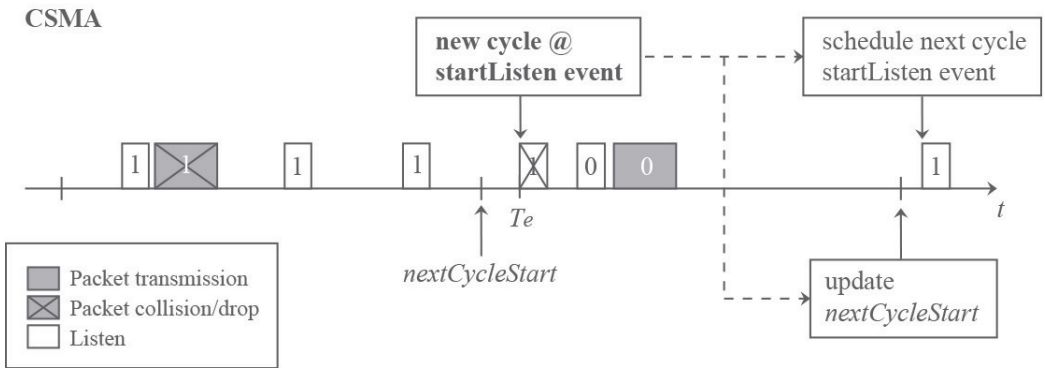


Figure 11: Check new cycle procedure example.

attempt of transmitting the packet resulted in a collision and subsequent retransmission attempts. The last attempt was made after the `nextCycleStart` time and hence the packet #1 was dropped.

All events are scheduled in a chain-like fashion: "start" events schedule their correspondent "end" events (e.g., `endListen` is scheduled at `startListen`) and the next event in the "chain" is scheduled depending on what happens (e.g., `endListen` schedules a `newPacket` if the channel is idle and a new `startListen` otherwise). The scheduling sequence only ends when the packet is transmitted successfully or if an event is "dropped" when a new cycle begins, as mentioned. In either case the node waits for the new cycle pre-scheduled event, when a new sequence starts.

Check Interference

In this section the process of managing the interference values in the simulation, as well as their impact on collisions, are presented. The described processes are both valid for packet and ACK collisions – ACKs are just as well considered as packets, where the sender is a gateway and the receiver is a sensor, but otherwise treated the same way.

Any active transmitter causes interference on all other nodes in the network. The most obvious and direct approach towards simulating this process would be to sum and subtract the interfering power on all other nodes every time a transmission starts and ends, respectively. For great number of nodes, however, this task becomes increasingly time-consuming. It is also redundant in most cases, since the interference that was added in other nodes will "stay" there for a very short time (only during the packet transmission), and the other node will most likely not even "notice" it. In other words, the interference value at any node only has any practical use if the node in question ever becomes active while the interference is there. Based on this observation, and since in general only a very small portion of the nodes are active at the same time (either transmitting or listening), a different strategy was adopted to determine the interference sum at the receivers: the interference power is only added/subtracted in active nodes (all

other nodes register null interference powers while being in sleep mode). Furthermore, the currently active nodes are registered in three different linked lists:

- All active transmitter-receiver pairs (`activeTx`);
- All active receivers (`activeRx`);
- All listening transmitters (`listening`).

While it might seem redundant to have a list for transmitters and one for receivers (when the pairs are unique), the receiver list serves the following purpose: since one receiver can have multiple transmitters sending information, the transmitter list can have more than one instance of that receiver, if they are transmitting simultaneously; when adding interference on all active receivers, it only needs to be done once per receiver, and hence the existence of a list of (unrepeated) active receivers facilitates this calculation.

It is worthy of reference that the option to store active node information in lists does not interfere on the overall process of checking interference (explained below), but only constitutes a lookup strategy.

The events that somehow change the state and/or impact of interference in the network are the start and end of transmissions and the start of a channel listening. Note that when a node stops listening, nothing happens in terms of interference – the only verification to be made is if activity was detected in the channel during listening.

Two main actions are always performed: firstly, the interference is added/subtracted and secondly, its effect is evaluated (did it cause a collision or was it detected by a listening node?).

Furthermore, there are three verifications to be made:

- interference of current transmitter on other active nodes:
  - receivers;
  - listening transmitters;
- interference of other active transmitters on current receiver.

Finally, these actions are crossed and mapped to form the following operations:

- in a `newPacket` event, all three actions are performed;
- in an `endPacket` event, there is no need to add the interference on the current transmitter, since it is changing to sleep mode, but note, however, that although the signal is no longer detected in listening transmitters, there is a need to check if the signal was present for long enough for the listener to decide if the channel is active;

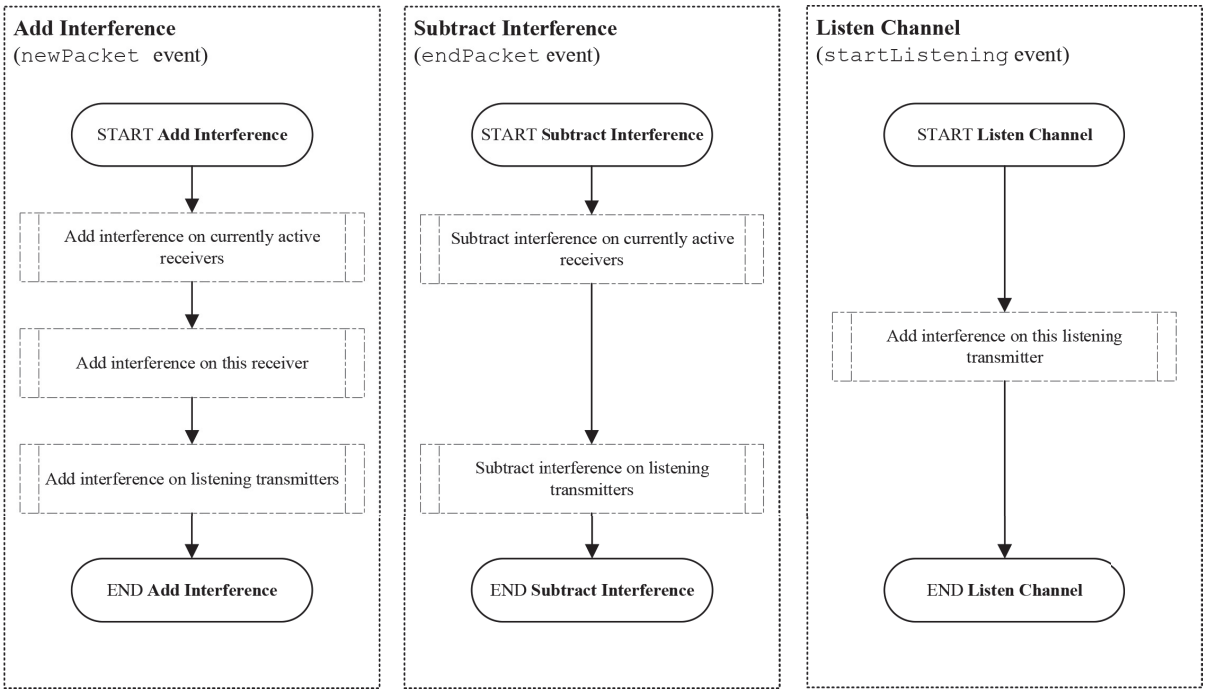


Figure 12: General flowcharts of interference-related functions in `newPacket`, `endPacket` and `startListen` events.

- in a `startListen` event, the node can only receive information, and hence the only verification to be made is that of detection of other signals.

In the flowchart of Figure 12 these verifications for each event are drawn and the distinction between the three types of verification is clearly visible. Each of the functions is further detailed in the flowcharts present in Annex A.

Furthermore, a relevant implementation detail resides in the fact that, in an ongoing transmission, interference is always checked at the receiver (which is where the interference makes damages to any ongoing transmissions) and the `interferenceSum` (which is the current total interference received power) is stored in the array's index correspondent to that very same receiver. However, if a collision occurs, that information, which is stored in the `collision` flag array, will be registered with the index of the transmitter, since, obviously, the transmission failure is an information that "belongs" to the transmitter. A packet collision occurs when the `interferenceSum` value is higher than the interference threshold `interferenceLim` defined for that transmitter-receiver pair. Note that the interference threshold is not characteristic of a receiver all alone, but of a transmitter-receiver pair, since it depends on the useful signal received power – with reference to (9). So the higher the received power, the higher the "allowed" interference.

In the case of channel listening, the interference is checked at the transmitter that is performing the listening – in this case the term "interference" is not really adequate, since the device is not sending anything

to be interfered with, but it is used for simplicity's sake and because the same implemented variables apply. In order for activity to be detected in the channel, the `interferenceSum` value must be higher than the detection threshold, `Pdet`. The detection method itself is different than that of packet collisions, since for the channel to be considered busy, the "interference" must be above the threshold for a minimum amount of time (`TR`), whereas a packet collision is instantaneous. The `channelBusySince` array serves the purpose of registering the time when activity first was detected in the channel, since it has been active. Once activity detection has effectively happened, the information is marked in the `collision` array – once again, "collision" is not an applicable term for this situation, but for simplicity the variable is used here. The following example serves as illustration: a signal was first detected at `Te = 1.00` and its time was registered; the signal stops at `Te = 1.01`; since the minimum detection time is `TR = 0.1`, the presence of the signal did not trigger the receiver, and so the time in `channelBusySince` is reset to zero; another signal is detected at `Te = 2.0` and fades at `3.0`; upon the transmission end, and knowing that it was present for enough time, the detection by the listening node is marked in the `collision` array.

5.4 DC

The implementation of DC features only the `newPacket` and `endPacket` events. Besides the new cycle and interference-related functions that were explained in Section 5.3, the implementation of DC follows a simple algorithm. In the `newPacket` event (Figure 13), the transmitted and offered packets counters are incremented (note that since there are no retransmissions, they are always equal), the transmitter and receiver are added to the active node lists and the `endPacket` is scheduled. Physically, this represents the fact that the node simply starts transmitting regardless of channel conditions.

During `endPacket` (refer to Figure 14), the transmitter and receiver information are first removed from the active node lists (note the receiver is only removed from `activeRx` if there it not currently receiving from any other node). Only after performing these actions (that emulate the turning off of the radio transmitter) does the function check for a new cycle, since they must always be executed. Finally, the `lostPacketsBuffer` is incremented in case of collision, the `collision` flag is reset and interference is removed. It should be observed that the addition and subtraction of interference are performed when the transmitter-receiver pair is not registered in the active node lists, which allows to simply calculate the interference on all the active nodes, without mistakenly adding/subtracting interference to the very pair that is causing it.

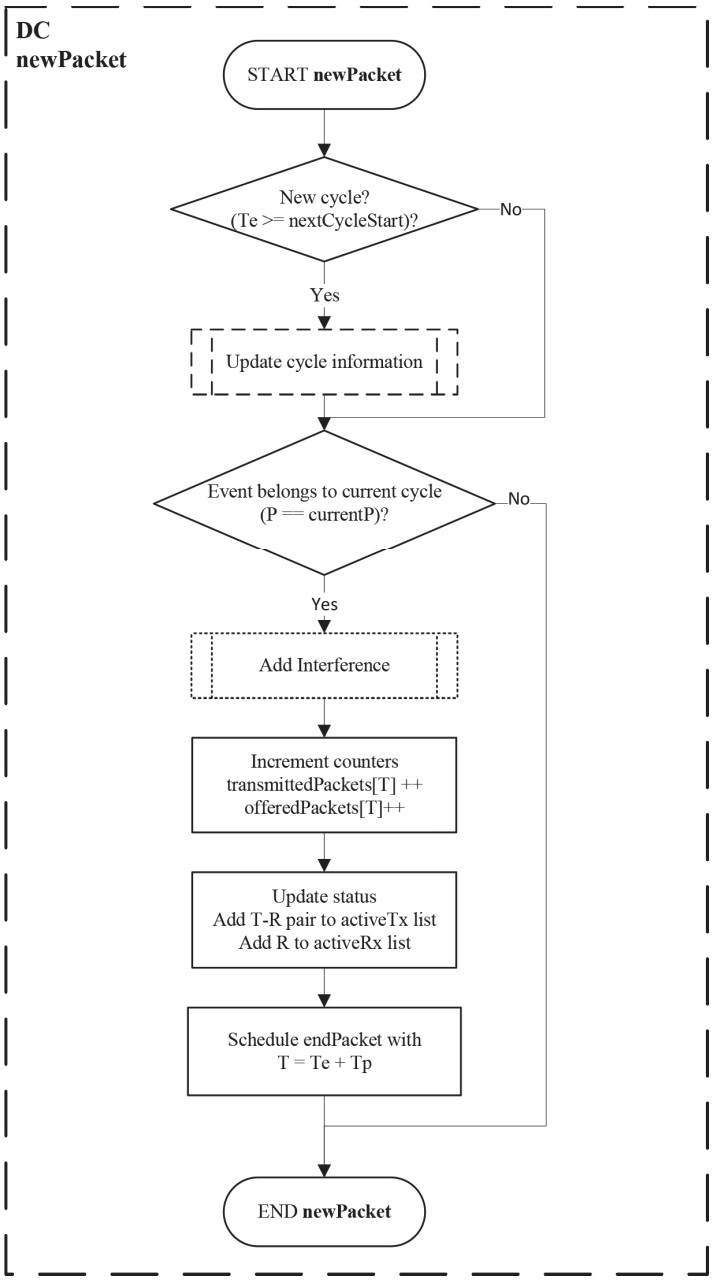


Figure 13: Implementation of DC `newPacket` event.

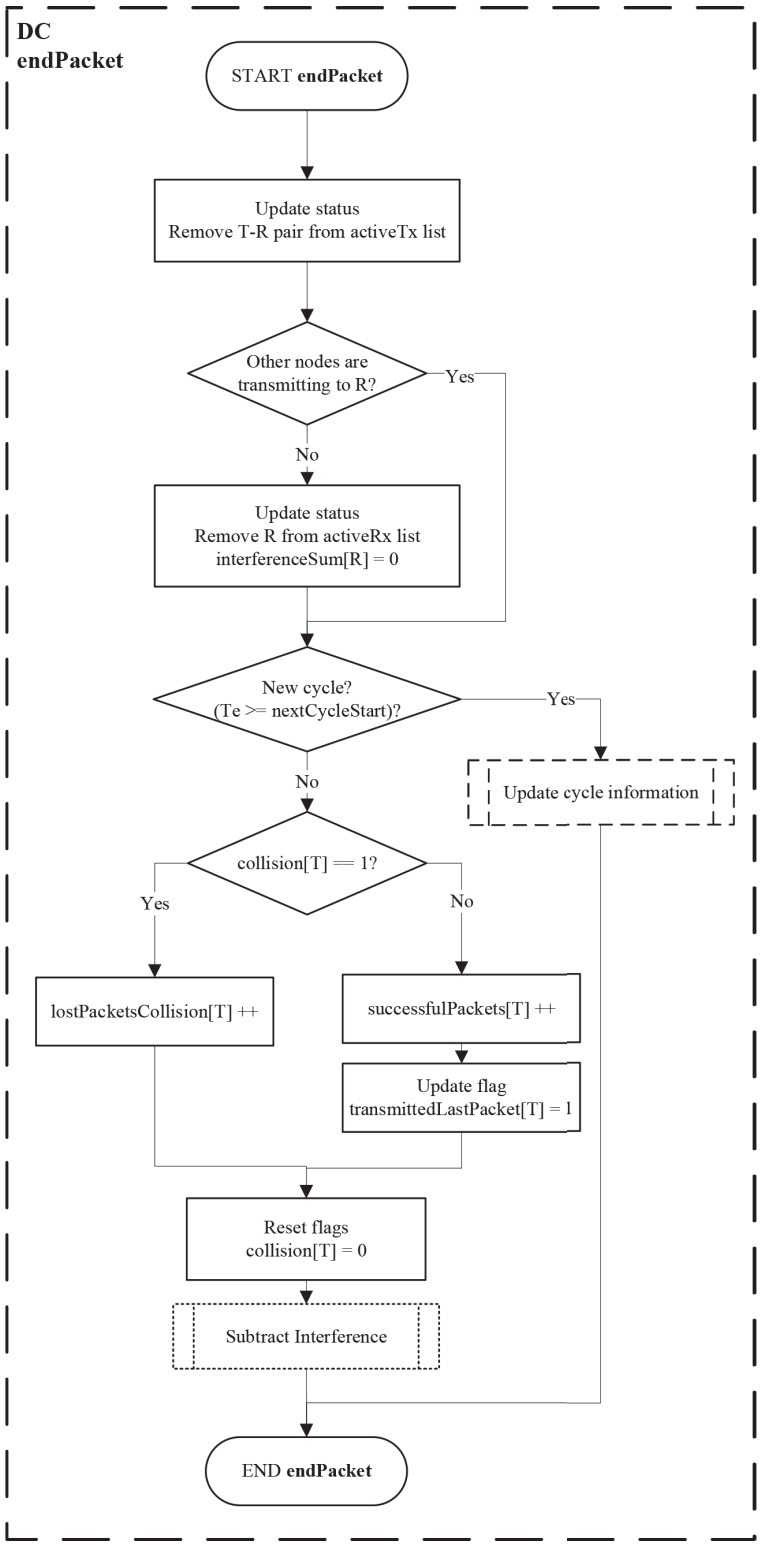


Figure 14: Implementation of DC `endPacket` event.

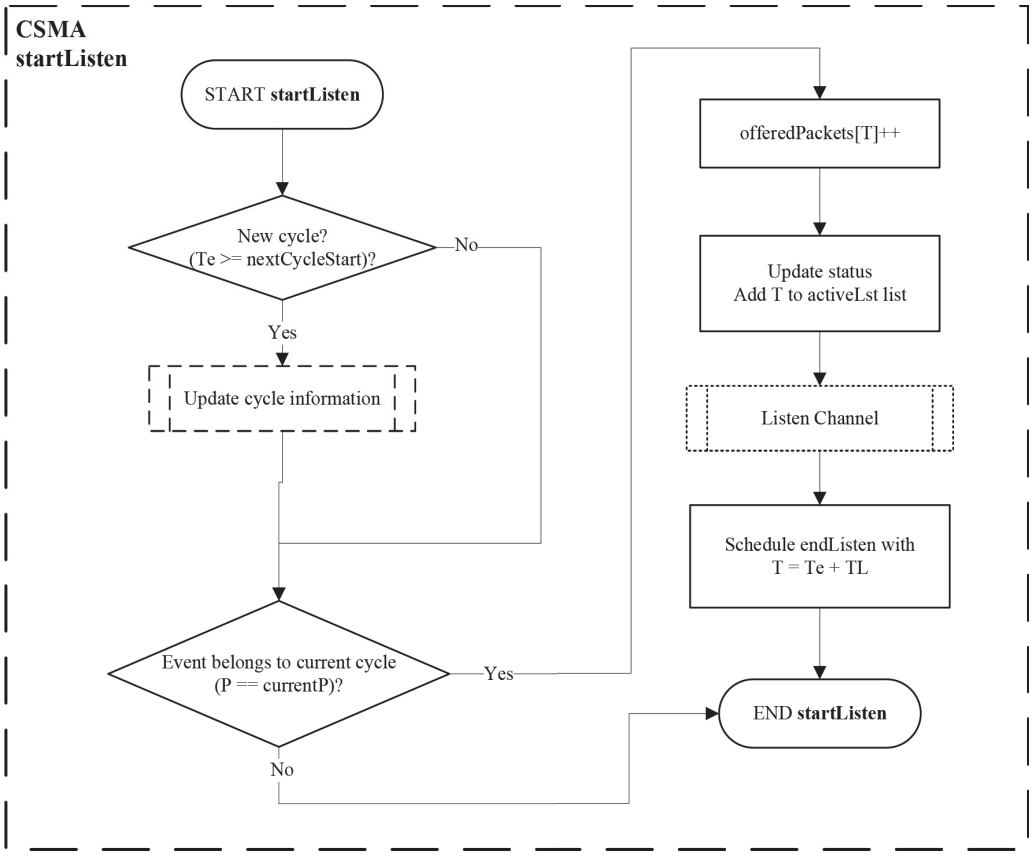


Figure 15: Implementation of CSMA `startListen` event.

5.5 CSMA

CSMA features the following events: `startListen`, `endListen`, `newPacket`, `endPacket`, `newACK`, `endACK` and `ACKtimeout`. The `newPacket` and `endPacket` functions are shared by Data and ACK packet events. Inside these funcions, the two packet types are distinguished, but some functions are common to both, since they both refer to the transmission of data.

Overall, the structure of CSMA events follows the same one as the previously detailed DC events. The arrival of a new application packet is checked right at the beginning for start events, and after updating the status information for end events. Likewise, Adding and Subtracting Interference functions are called when the node IDs are not on the active node lists. Besides these common blocks, the specific functions of each event are also located in the same place in the structure, as in the case of DC.

In the `startListen` event (in Figure 15), after checking for a new cycle and for an outdated packet, the function proceeds to incrementing the `offeredPackets` counter, updating the status, listening to the channel and scheduling the `endListen` event.

The `endListen` function starts by checking if the packet belongs to the current cycle (i.e., if it is not "outdated"). As mentioned previously for all end events, the status information is updated and only after

the new cycle is tested. Afterwards, the event-specific functions are run. Channel activity detection is assessed by checking if the `collision` flag has been set to 1, registering a detection that occurred previously, or if the channel is still being sensed busy, and has been so for at least a time of `TR`. In case no activity is detected, the `newPacket` is scheduled. In the opposite case, a `startListen` is rescheduled (and the respective counter incremented). Finally, the involved flags are reset.

Following the aforementioned structure for start events, the `newPacket` (represented in Figure 17) event checks for "outdated" packet and new cycle, adding interference next and updating status information. Finally, the event-specific functions are performed for either data or ACK packets. The respective counters are implemented and the respective end events scheduled. Data and ACK packets count as distinct events, but are executed in the same function due to having their nature in common (both emulate a radio transmission). An event ID serves as differentiator, not only in this case, but for all events.

Again, similarly to other end events, `endPacket` (Figure 18 and Figure 19) starts with the packet ID check, status update and new cycle verification, followed by the specific functions for Data and ACK packets. Only if a data packet transmission is successful, then the ACK is scheduled. The `ACKtimeout` event is scheduled and the `waitingACK` flag is set to one. When the ACK is successfully received, this flag is reset (meaning the node is no longer waiting for the confirmation). Finally, at the `ACKtimeout`, the `waitingACK` flag is evaluated: if it is 0, then the packet was successfully received and no further action is taken (the event "chain" ending here). Otherwise, the transmission is assumed to have failed, and hence a transmission retrial is scheduled.

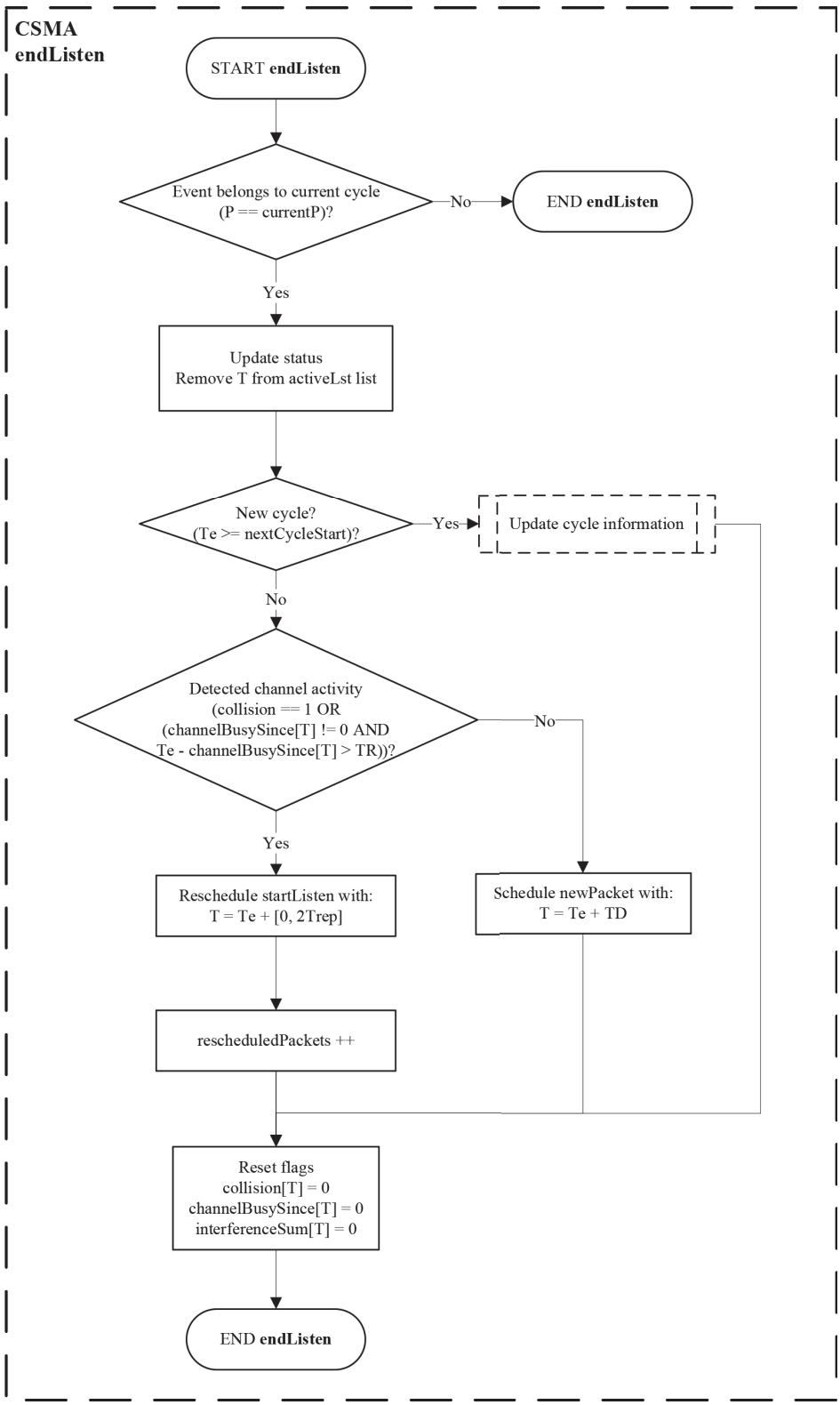


Figure 16: Implementation of CSMA `endListen` event.

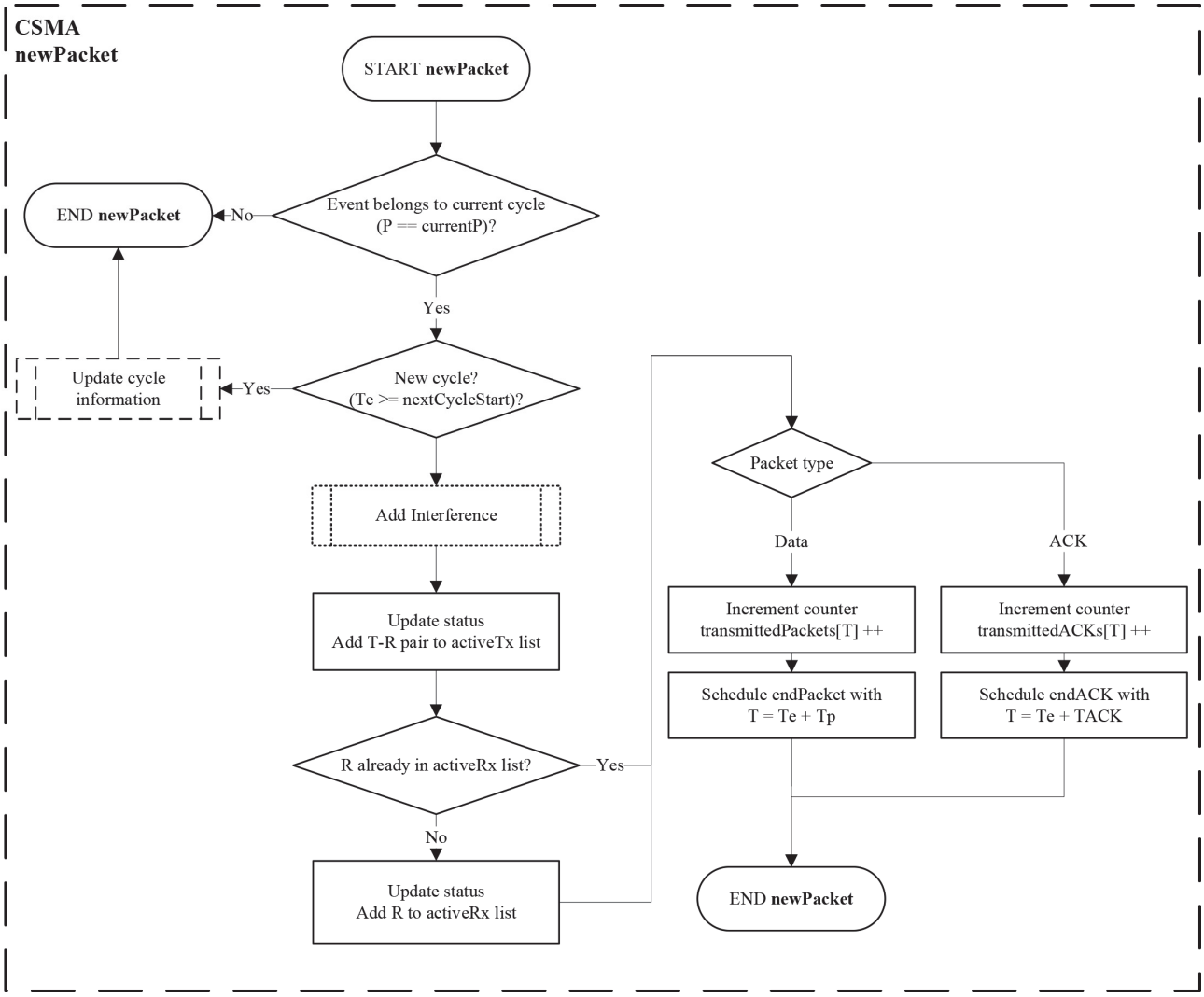


Figure 17: Implementation of CSMA `newPacket` event.

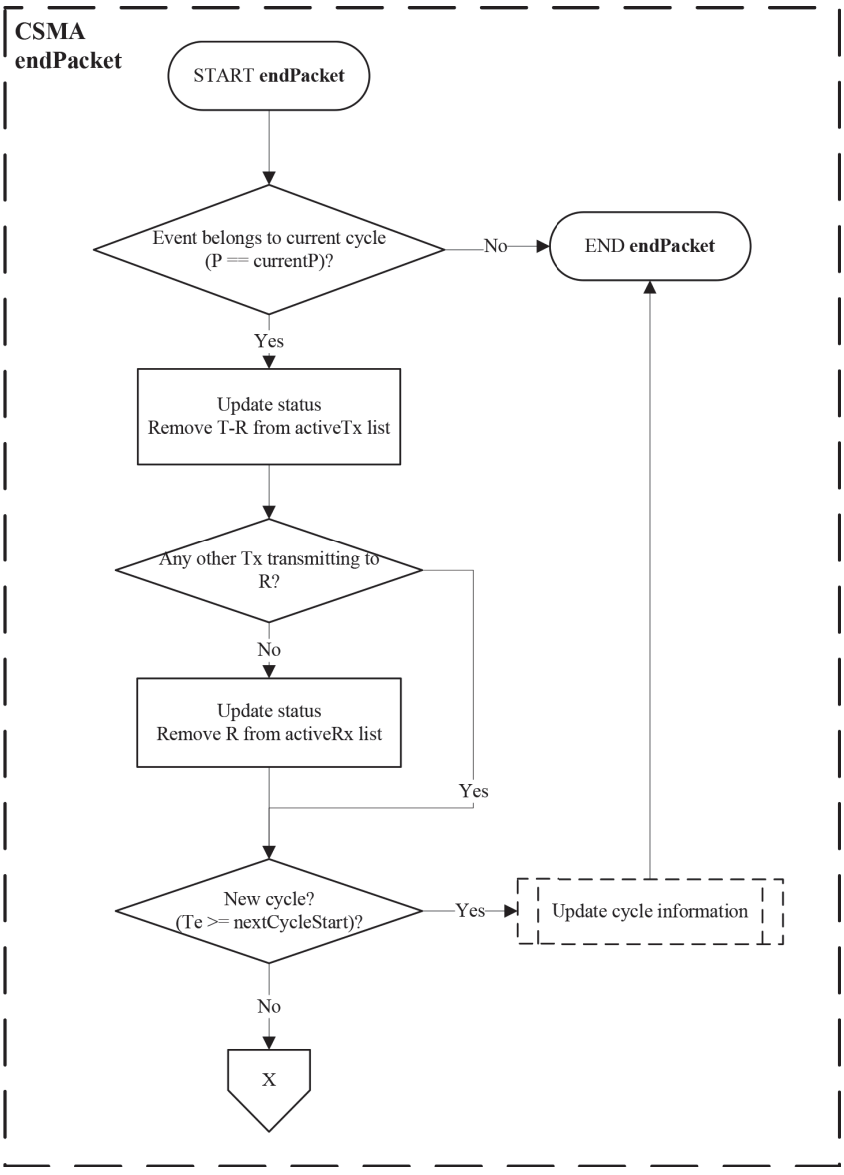


Figure 18: Implementation of CSMA `endPacket` event – part (1/2).

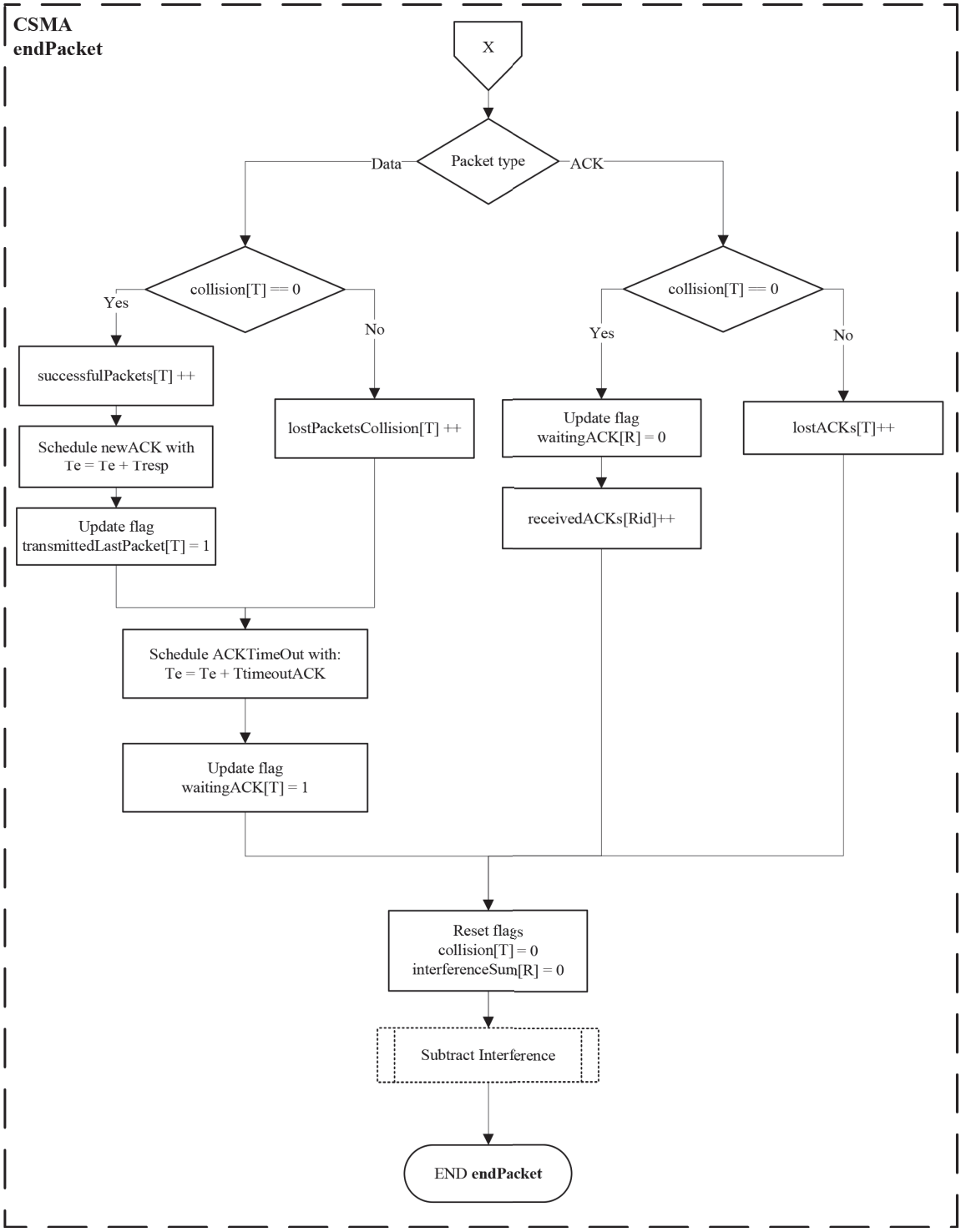


Figure 19: Implementation of CSMA `endPacket` event – part (2/2).

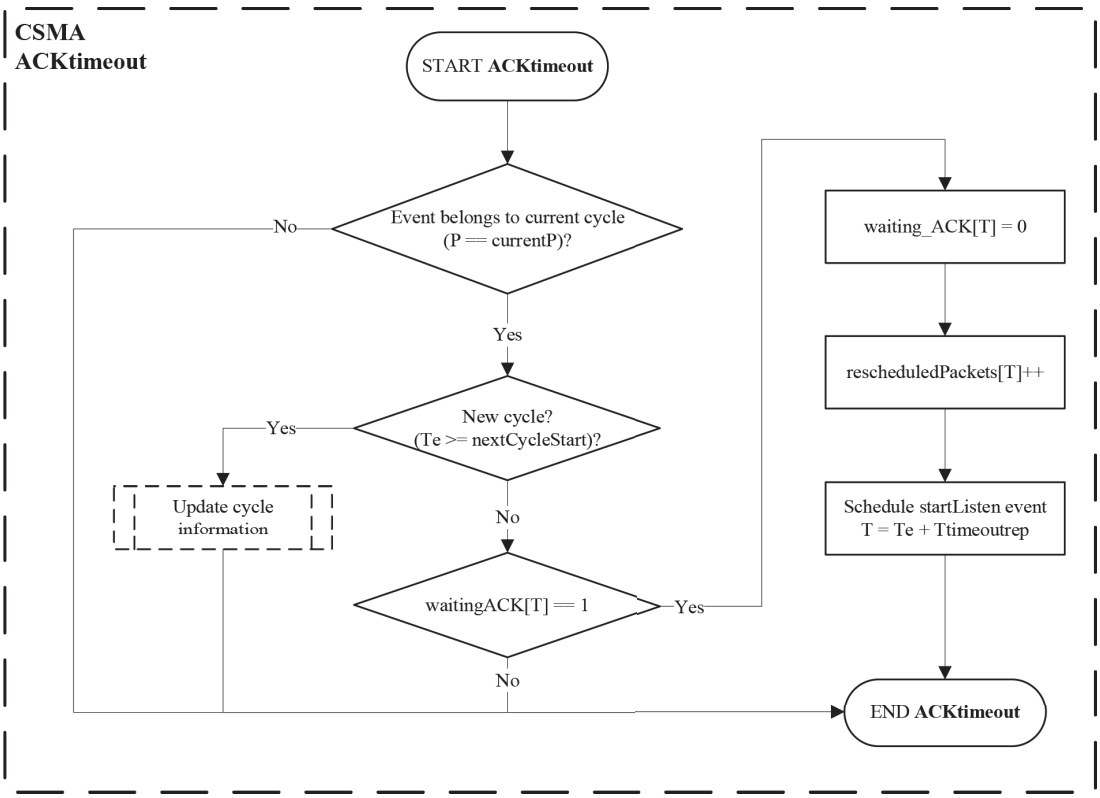


Figure 20: Implementation of CSMA `ACKtimeout` event.

Bibliography

[1] ECC. ERC Report 182: Survey about the use of the frequency band 863-870 MHz. Technical report, 2012.

[2] ECC. Report 181: Improving Spectrum Efficiency in the SRD Bands. Technical report, 2012.

[3] David J. Wetherall Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 5th edition, 1994.

[4] Introduction to Monte Carlo Methods. D. J. C. Mackay. `https://www2.stat.duke.edu/courses/Spring12/sta270/lec/mcmcnotes.pdf`, 2012. Accessed: 2016-09-01.

[5] UAH College of Science. The Central Limit Theorem. `http://www.math.uah.edu/stat/sample/CLT.html`. Accessed: 2016-09-01.

[6] Michael D Byrne. How many times should a stochastic model be run? An approach based on confidence intervals. In *Proceedings of the 12th International conference on cognitive modeling, Ottawa*, 2013.

[7] Martocci, Jerry and Vermeylen, Wouter and Riou, Nicolas and Mil, Pieter De. Building automation routing requirements in low power and lossy networks. Technical report, IETF, 2010.

[8] International Electrotechnical Commision (IEC). Internet of Things: Wireless Sensor Networks. Technical report, 2014.

[9] ITU. Recommendation ITU-R P.1238-8 (07/2015). Technical report, 2015.

[10] Ian Poole. What is noise figure. `http://www.radio-electronics.com/info/rf-technology-design/rf-noise-sensitivity/noise-figure-factor.php`. Accessed: 2017-04-12.

[11] IMST GmbH. Channel Access Rules for SRDs. Technical report, 2012.

[12] Crossbow Technology. Mica2 Datasheet. `https://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf`. Accessed: 2016-09-01.

Detailed Flowcharts of Interference-related Functions

The present Annex details the functions from Figure 12, including the respective complete flowcharts. The general approach towards managing interference-related values comprises two main parts: 1) updating the `interferenceSum` value and 2) checking the effects of the previous changes.

The algorithm in the Add Interference function (in Figure 21 and Figure 22), for the three blocks is explained as follows:

- Interference is first added to all active receivers; afterwards, all active transmitters' situation is evaluated – if their correspondent receiver registers an interference above the threshold, they are "marked" as having collided;
- On the second block, a similar process takes place, where the "target" is now the current transmitter-receiver pair – interference from all active transmitters is added and collision is checked;
- Finally, in the third block interference is added to each listening node, and the signal detection is evaluated – the node only registers the signal detection if 1) the signal power is above its sensitivity and 2) it has not yet started detecting any activity (if otherwise activity has been already detected, no changes are made).

Upon the end of a packet transmission the Subtract Interference function (shown in Figure 23 and Figure 24) is executed, where:

- Firstly, interference is removed from other currently active receivers; note that in this case there is no "checking the consequences" phase, since there are none (only when *adding* interference to ongoing transmissions might there be collisions);
- Secondly, the effect of ending the transmission on listening transmitters is evaluated by performing the tests: 1) there can only be any changes if the node was detecting channel activity, but then stopped when the transmission stopped (otherwise, if the node was either not detecting anything or if the channel activity remained, there are no changes) and 2) the listener actually stopped detecting

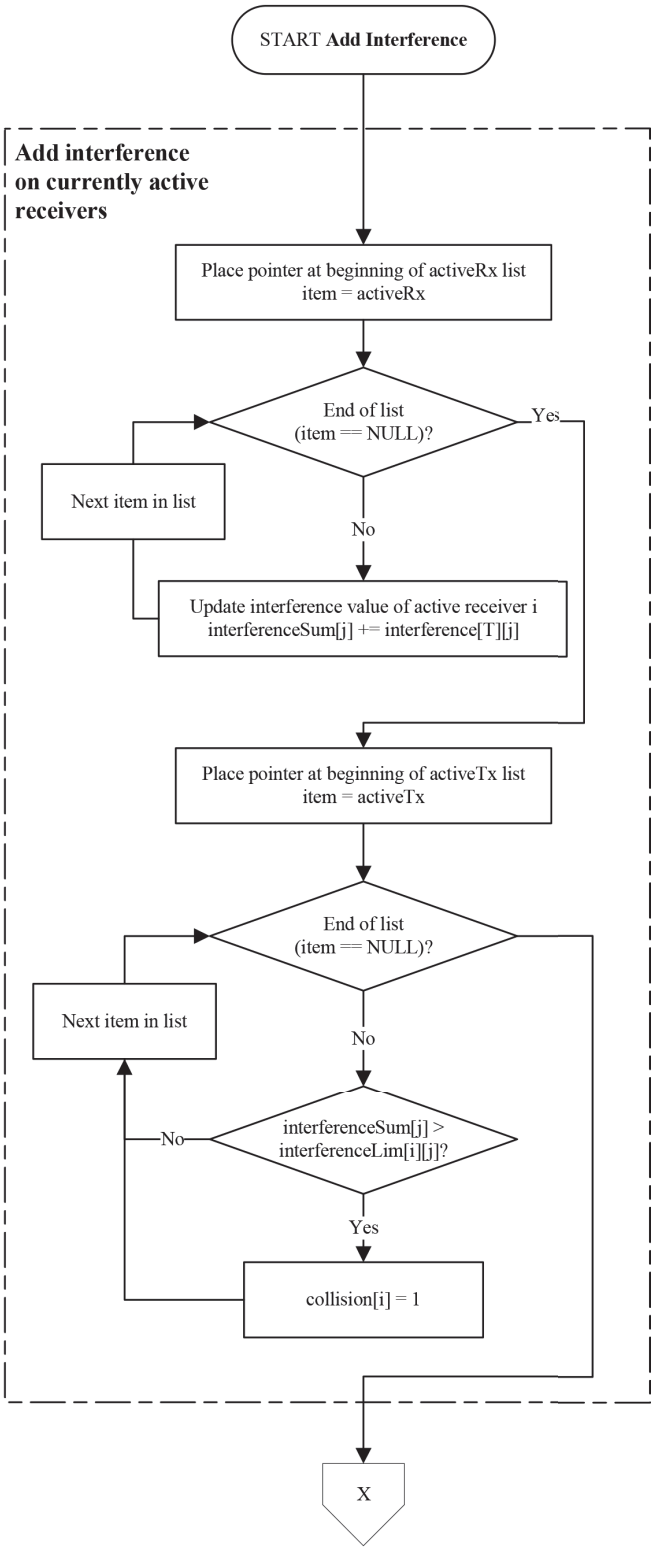


Figure 21: Flowcharts of **Add Interference** function in `newPacket` events (part 1/2).

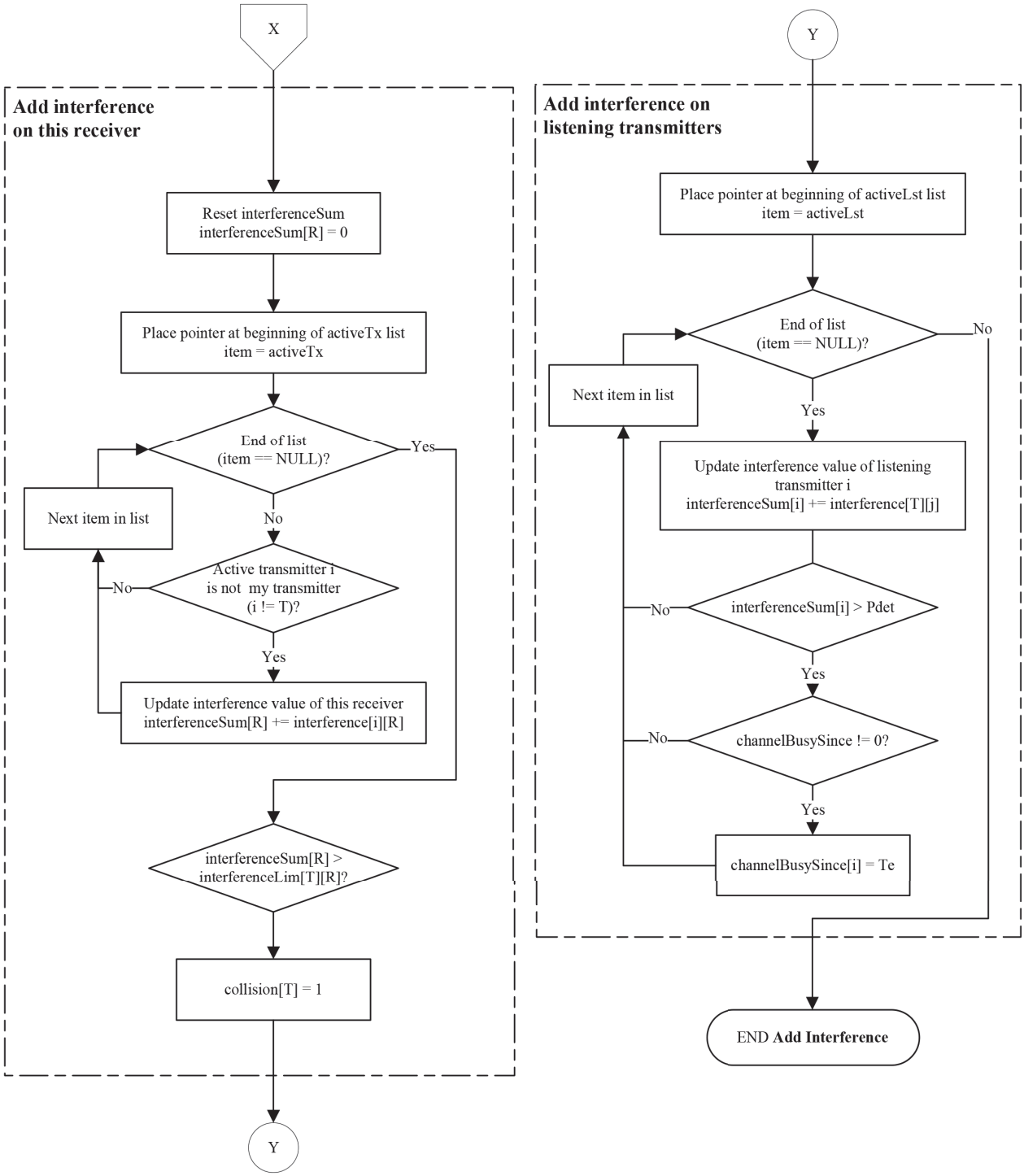


Figure 22: Flowcharts of **Add Interference** function in `newPacket` events (part 2/2).

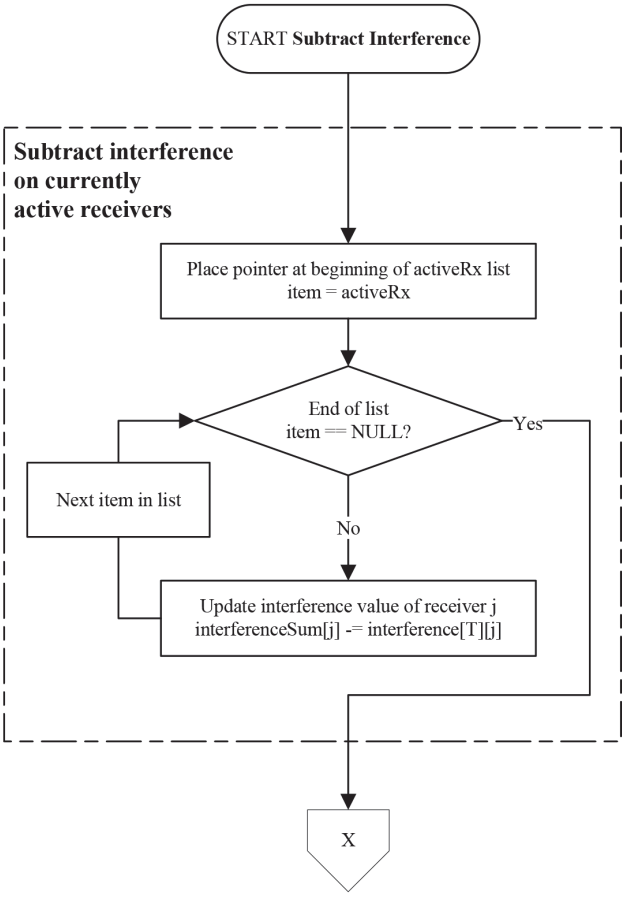


Figure 23: Flowcharts of **Subtract Interference** function in `endPacket` events (part 1/2).

activity upon the end of the transmission, it must be verified if that activity was present for long enough to be registered (i.e. if the time between the first detection and the current time is larger than the detection interval, `Te - channelBusySince[i] > TR`). If so, the `collision` value is set to one to register the the detection of the activity. If not, then the previous activity information is erased by resetting `channelBusySince`, since it did not result in an effective detection registration.

The Listen Channel function (in Figure 25) performs the sum of all the current signal power in the channel to determine if the current listening transmitter can detect the activity. If so, the time of detection is registered in `channelBusySince[T]`.

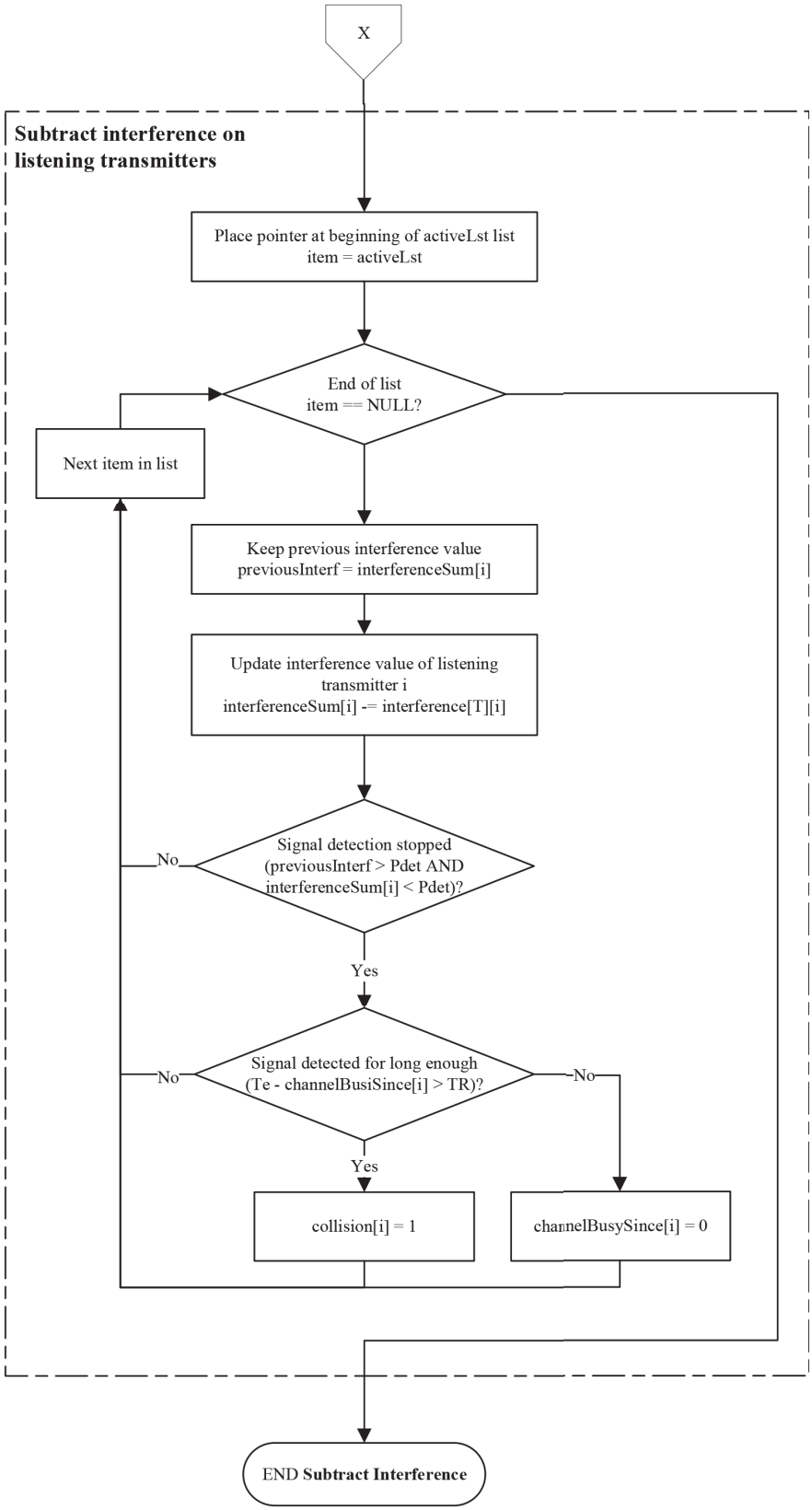


Figure 24: Flowcharts of **Subtract Interference** function in `endPacket` events (part 2/2).

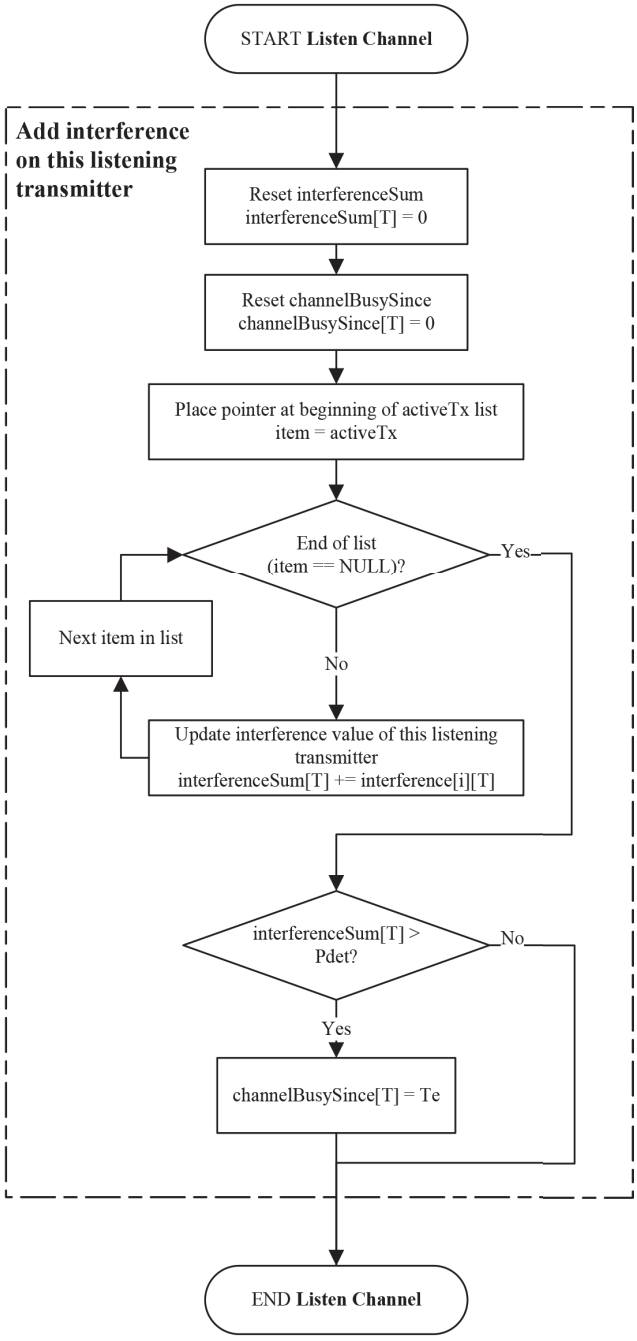


Figure 25: Flowcharts of **Listen Channel** function in `startListen` events.