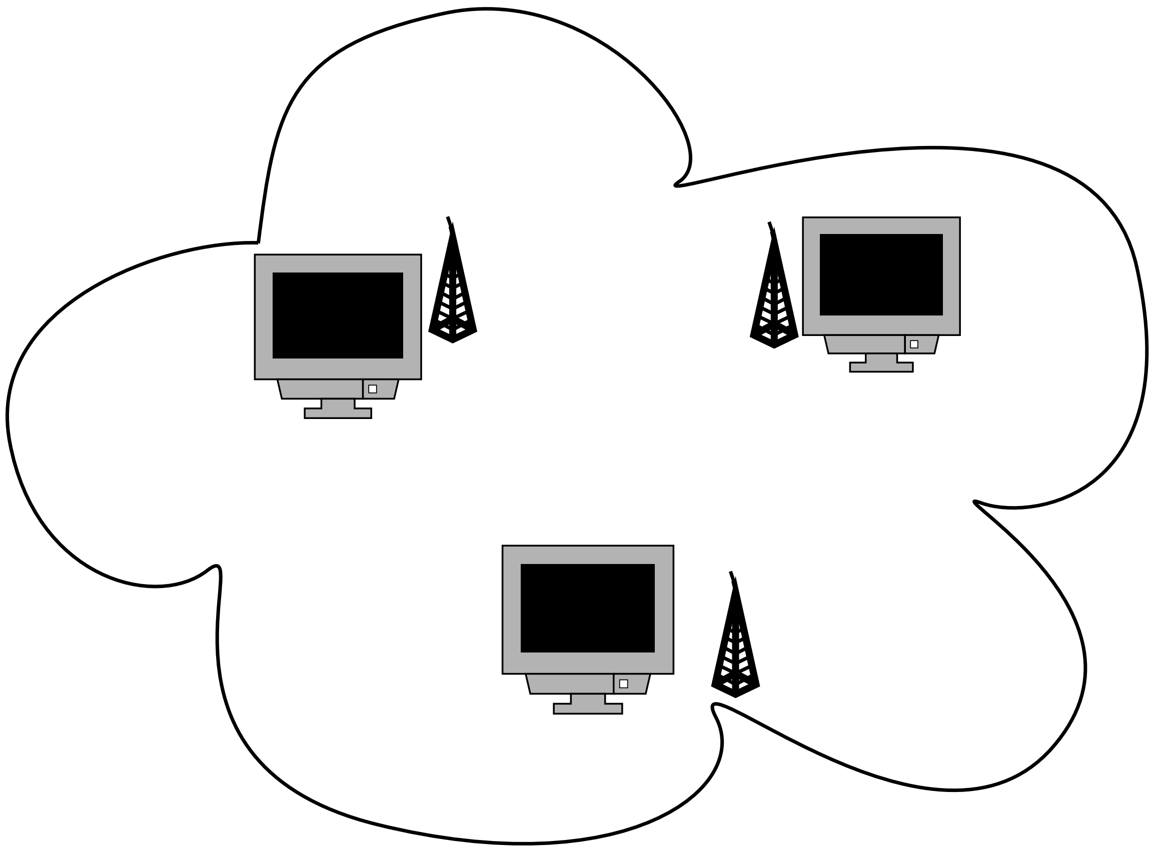
Final Report Fopra Energy Consumption in Ad-hoc Networks

Christian Gross

June 22, 2005



Contents

1 Introduction

The first goal of this work is to model and simulate the communication between stations according to IEEE 802.11 in adhoc modus using 'basic access method'. The second goal is to model the energy consumption and to introduce various energy economy modes to reduce energy consumption of each station. The implementation is conducted in the modelling and description language MoDeST, the MoDeST tool environment MoTor and the performance evaluation environment Möbius. Beside the energy consumption, the total channel usage in various situations with different number of senders and different loads is simulated.

1.1 Basic Access Method of IEEE 802.11

One of the basic classes of IEEE 802.11 is called Independent Service Set or ad hoc network. In this class, stations communicate over a shared wireless channel without a centralized medium access control protocol which would guarantee a collisionfree access to the channel. If there is no such medium access control present, it is mandatory to avoid collisions (simultaneous transmissions) as much as possible. A station is unable to listen to the channel for collisions while transmitting. The IEEE 802.11 standard defines a Distributed Coordination Function (DCF) to overcome this problem.

One of the situations in which collisions can occur is the following: before a transmission is started a station has to monitor the channel until it is considered to be free. After this the station can start the transmission at the actual time t. Other stations can detect this transmission only after a delay D at time t'. This delay consists of the time that the sending station needs to switch from receiving mode to transmission mode, the time needed to assess the channel and the airpropagation time. If another station decides to start a transmission during this delay D, there will occur a collision. The delay D is called the `vulnerable period`.

The Basic Access mechanism is as follows:
If a station wants to transmit a new packet, it has to monitor the channel as long as it is free for a given duration `DIFS` (DCF interframe space), which is at least as long as the delay D mentioned above. After sending the packet the station listens to the channel. If another station is transmitting at this time, the station decides that there has been a collision. If not it waits for an acknowledgement. This will be sent after time `SIFS` (short interframe space). If a station detects a collision it starts a backoff procedure, which is an important feature of DCF. After the channel is free for at least the dura-

tion `DIFS` a backoff value is chosen randomly from the interval [0,aCWmin]. This value indicates the number of channel-free time periods (slottimes) after which the station may retransmit. The backoff value is decremented by 1 each time period the channel is considered to be free. If the backoff value is 0, the station starts retransmitting. If the channel is considered to be busy in a time period, the backoff value will be kept and after the channel is free for `DIFS`, it will be decremented again each time period the channel is considered to be free. If a retransmission is not successful the station runs the backoff procedure once more, in which case the interval of the backoff value is set to be nearly double. If the interval is equal to [0,aCWmax] and the retransmission fails then the message is dropped. If a retransmission is successful the next backoff value will be chosen from the interval [0,aCWmin].

Nevertheless, it is important to note that there are two situations, called hidden terminal problem and exposed terminal problem, in which this basic access procedure can fail. In the hidden terminal problem there are at least two stations that are not in the same communication range of each other and both send at the same time to other stations that are in the communication range of each. The following example describes such a situation (see also Figure 1). Assume there are three stations A, B and C. A and C cannot communicate because the distance between them is too large. B can communicate to both. C does not have any ways to know that A is transmitting packets to B and can decide to transmit packets to B at the same time as A, which results in a collision. The situation in the exposed terminal problem is similar. There are at least two stations that are not in the same communication range of each other and both send messages to stations that are not in the communication range of the other sending station. These receiver stations are in the same communication range and if they send acknowledgements there occurs a collision. The following example describes such a situation. As in the previous example there are three stations A, B and C. Additionally there is a station D which can only communicate with C. If A sends to B and at the same time D sends to C, then after receiving the messages B and C have to send acknowledgements to their communication partners, which causes a collision (see also Figure 2).

IEEE 802.11 defines another access method, called 'rts/cts' to solve these problems. In this access method the sender transmits first a 'ready-to-send (rts)' message, the receiver answers with a 'clear-to-send (cts)' message and after this handshake the sender starts to transmit the data.

In this work it is assumed that every station is in the communication range of each others and hence this work focus on the basic access procedure.

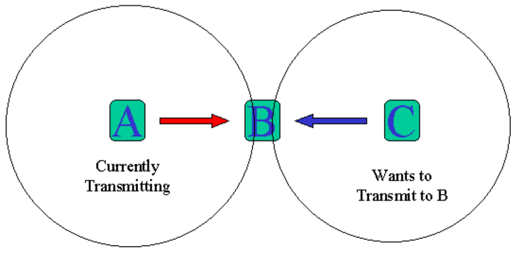


Figure 1: The Hidden Terminal Problem.
Source: http://pcl.cs.ucla.edu/slides/workshop99/Ken-pw99/sld008.htm

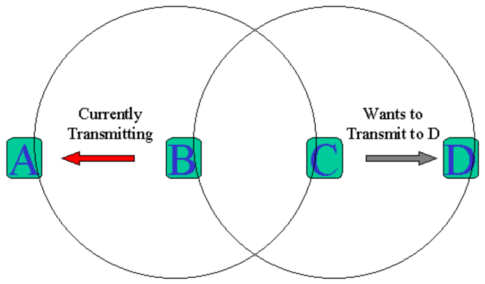


Figure 2: The Exposed Terminal Problem.
Source: http://pcl.cs.ucla.edu/slides/workshop99/Ken-pw99/sld009.htm

1.2 MoDeST, MoTor and Möbius

MoDeST is a modelling and description language for stochastic timed systems with a syntax similar to C and Promela. The most important features in MoDeST are:

1. non-deterministic choice
2. probabilistic branching
3. clocks

4. delay nondeterminism

MoDeST has a rigid formal basis such that formal reasoning is possible. Because of the syntax similar to C and some constructs like simple data types, modularisation and atomic statements, it is relatively easier to use than some other rigorous formalisms based on stochastic process algebras.

MoTor is a tool to facilitate the analysis of MoDeST models. With MoDeST a wide range of timed, probabilistic, nondeterministic, and stochastic models can be covered. The spectrum of this covered models includes ordinary labeled transition systems, discrete and continuous time Markov chains and timed and probabilistic timed automata. Because of the enormous expressiveness there is no generic analysis algorithm at hand. The philosopy behind MoTor is to connect MoDeST to existing tools and not to reimplement existing analysis algorithms anew.

Möbius is a performance evaluation tool environment developed at the University of Illinois at Urbana-Champaign, USA and supports several input formalisms. Atomic models are specified in one of these input formalisms, for example MoDeST. Furthermore the user can specify a reward model. In the case of MoDeST, the user appoints in the reward model the global variables that are monitored in a simulation. Möbius supports the definition of experiment series, called studies. In these studies the user can define different values for some parameters (in the case of MoDeST: values for global constants that are marked as extern in the atomic model). The combinations of all values defined for the different parameters define such series. One combination is called an experiment. In the section solver, the experiments are executed. The execution of one experiment is repeated until the values of the monitored variables converge in a defined interval and the minimum number of executions is reached or the maximum number of executions is reached. The user can specify the minimum and maximum number of executions. A nice feature of Möbius is that one or more experiments can be executed on some computers in parallel. The role of MoTor in the context of Möbius is that the integration of MoDeST into Möbius is done by means of MoTor.

2 The Model

In the first step, the basic access method described at the beginning of this report and the backoff procedure which is executed by a sending station when a collision occurs, are modelled. In the second step the energy consumption, three energy economy modes and a monitor process are modelled. It is assumed that the clocks of all stations are synchronous all the time and for this reason it is not necessary to model a distributed algorithm to synchronize the clocks. Initially all stations that take part in the simulation try to start sending a message and will collide. The values of the parameters that are used in this work are summarized in Table 1. They are explained when they occur for the first time.

Parameter	Value
DIFS (DCF interframe space)	128μs
SIFS (short interframe space)	28μs
ASLOTTIME	50μs
vulnerable period	48μs
Trans_time_min	224μs
Trans_time_max	15717μs
ACK_TO	300μs
aCWmin	15
aCWmax	1024
transmission_factor	1.625
sensing_factor	1.475
idle_factor	0.08

Table 1: Important Parameters of IEEE 802.11 Used

There are three kinds of processes to model the communication:

1. Sender process
2. Receiver process
3. Channel process

These processes are executed in parallel by using the 'par' construct of MoDeST.

2.1 The Sender Process

The sender process models the behaviour of a station that sends messages according to IEEE 802.11. The sender process has three parameters:

1. `id:` a positive integer. This is the "address" of a sender which a receiver uses to answer to the sender.

2. `load:` a float in interval (0,1], which defines the frequency with which a sender attempts to send new messages.

3. `active:` integer value. If it is greater or equal to `id`, then the sender is active during simulation. If not the sender is not active.

At the beginning, a sender process verifies if it should be active by the parameter `active`. If `active` is greater or equal to the parameter `id` the process starts to send messages. Otherwise the process will stop (lines 3 and 5). An 'active' sender inspects the channel.

```
1   process sender(int id, float load,int active) {
2   alt{
3      ::when(active<id) tau;
4        when(global_time==simulation_duration) tau
5      ::when(active>=id)
6        do{
7        ::when(ready_to_send!=true && initialsend!=true && y==633)
8          random_choice=Uniform(0,1);
9          alt{
10            ::when (load>=random_choice) ready_to_send=true
11            ::when (load<random_choice)
12              do_nothing{= y=0,backoffswitch=true,
13                           msg_to_send[id]=0 =}
14            }
15       ::when(ready_to_send || initialsend)
16         reset_clock_and_counter{=x=0,y=0,backoff_finished=!true,
17                                 continue_backoff=!true,
18                                 wait_DIFS=true,initialsend=!true,
19                                 msg_to_send[id]=1,ready_to_send=!true=};
```

Figure 3: Sender Model in MoDeST (1)

If the channel is free for `DIFS`, the sender enters the `vulnerable period` and starts sending for a random duration d (lines 55 and 61). The value of d is greater or equal to the parameter `Trans_time_min` and less or equal to `Trans_time_max` (line 57). If in the `vulnerable period` another sender decides to start sending too, there will occur a collision. The channel process notices this and sets the variable `collision` to true. The sender which has finished sending first, inspects the channel and notices that another process is sending, decides there must be a collision and will retransmit after performing the backoff procedure. The second process inspects the channel too after sending, notices that the channel is free and waits for a receiver acknowledgement.

```
do{::urgent(c.free==0 && backoffswitch!=true &&
            continue_backoff!=true)
     when  (c.free==0 && backoffswitch!=true &&
            continue_backoff!=true)
           s1_enable_backoff {=backoffswitch=true=}
   ::when((backoffswitch || continue_backoff) &&
           backoff_value>=0) deals_with_backoff{=x=0=};
     alt{
       ::when(wait_DIFS)
          alt{
           ::when (x<DIFS && c.free==0 && wait_DIFS)
             reset_sense_in_backoff{=x=0=};
             when (c.free==1) tau
           ::when(x>DIFS && c.free==1)
             waited_DIFS_backoff {=wait_DIFS=!true=}
             }
       ::when (wait_DIFS!=true)
         alt{
           ::when(backoffswitch) choose_backoffvalue
             {=backoffswitch=!true,continue_backoff=true,
               x=0,backoff_value=Uniform(-1,CW)=}
           ::when(continue_backoff) continuebackoff
             {=backoffswitch=!true,continue_backoff=true,x=0=}
            };
         alt{
           ::when(backoff_value<0) tau
           ::when(x>=ASLOTTIME && c.free==1)
             count_down_backoffvalue{=backoff_value-=1=}
           ::when(x>=ASLOTTIME && c.free==0 ) freeze_backoff
             {=continue_backoff=true,wait_DIFS=true=};
             when (c.free==1) tau }  } }
  ::when(backoff_value < 0)
    exit_backoff{=backoffswitch=!true,continue_backoff=!true,
                  backoff_finished=true,backoff_value=0=}
```

Figure 4: Sender Model in MoDeST (2)

Because of the collision there will be no such acknowledgement and after `ACK_TO` time interval this sender starts retransmitting after performing the backoff procedure too. The backoff procedure is also executed if a sender notices a busy channel when it is waiting for the channel to be free for `DIFS` (lines 20 and 25). In the backoff procedure a sender waits first until the channel is free for `DIFS`. If so, a random float value in the interval [-1,CW] is chosen.

```
54 ::when((x>DIFS || backoff_finished) && backoffswitch!=true
55         && continue_backoff!=true) enter_vulnerable_period
56   {=vuln_reached[id]=1.0,x=0,
57     sending_duration[id]=Uniform(Trans_time_min,Trans_time_max)=};
58     when (x>= VULN)
59     start_send{=sending+=1,is_sending[id]=1,x=0,backoffswitch=true,
60     c.m_in.destination=-id,c.m_in.source=id,c.m_in.content=5=};
61 ::urgent(x>=sending_duration[id]) when (x>=sending_duration[id])
62   send{=c.m_out.source=c.m_in.source,c.m_out.content=c.m_in.content,
63         c.m_out.destination=c.m_in.destination,sending-=1,
64         is_sending[id]=0 =}; x=0;
65   alt{
66     ::when (x==0 && sending>0)
67       detects_channel_busy_after_sending_and_will_resend
68       {=initialsend=true,CW=(CW+1)*2-1,vuln_reached[id]=0 =};
69       break
70     ::when(x==0 && sending==0) start_wait_for_ack {=x=0=};
71       alt{
72         ::urgent(c.m_out.destination==id&&x>=SIFS+ACK_DURATION)
73           when  (c.m_out.destination==id&&x>=SIFS+ACK_DURATION)
74           ack_received_correct {=CW=aCWmin,
75                                   vuln_reached[id]=0=};break
76     ::when(x>ACK_TO && c.m_out.destination !=id)
77       alt{
78         ::when(CW< aCWmax)
79           will_resend_message{=initialsend=true,CW=(CW+1)*2-1,
80                                 vuln_reached[id]=0=};break
81         ::when(CW>=aCWmax) will_skip_message
82           {=vuln_reached[id]=0=};break
83          }
```

Figure 5: Sender Model in MoDeST (3)

If this value is less than 0 the sender enters `vulnerable period` and starts sending. Otherwise it will be decreased by 1 everytime the channel is free for `ASLOTTIME` (lines 46 and 47). If the channel is busy the actual value will be kept and after the channel is again free for `DIFS` the value is decreased again everytime the channel is free for `ASLOTTIME` (lines 48 and 28). If a retransmission is necessary and the value of CW is less than `aCWmax` it will be increased according to formula: $CW = (CW + 1) * 2 - 1$. If a retransmission is necessary and CW is equal to `aCWmax`, the retransmission is dropped (line 81). After a successful retransmission the value of CW is reset to `aCWmin`.

The parameter `load` is a float value between zero and one that defines the frequency a sender tries to send new messages. A load of one means that a sender wants to send all the time and if the value approximates zero the sender tries to send less often. This means concretely that, if a message has been successfully sent, the process chooses a random value between zero and one. If this value is less than the parameter `load` the process will directly try to send another message. If not the process will wait for 633 μseconds and then repeat the random choice. The value 633 is chosen because it takes at least 633 μseconds to successfully send a message and receive an acknowledgement (128μs DIFS + 48μs vulnerable period + 224μs transmission + 28μs SIFS + 205μs acknowlegement).

2.2 The Receiver Process

This process models the behaviour of a station that receives messages and sends acknowledgements to the senders according to IEEE 802.11. A receiver process has a parameter `id`. If there is a message with the destination `id` and there is no collision the process receives the message and answers after `SIFS` with an ack message containing the `id` of the sender (lines 5 and 7).

```
process receiver(int id)
{
clock y;
do{
  ::when(c.m_out.destination==id && collision!=true)
    receive{=y=0,c.m_out.destination=0,source=c.m_out.source=};
    urgent(y>=SIFS) when(y>=SIFS) start_send_ack
    {=y=0,ack+=1,c.m_in.destination=source,c.m_in.source=id,
     receiver_is_sending=true,sending+=1=};
    urgent(y>=ACK_DURATION) when(y>=ACK_DURATION)
    send_ack{=c.m_out.source=c.m_in.source,
              c.m_out.destination=c.m_in.destination,
              receiver_is_sending=!true,
              sending-=1,time_receiver_send+=y=}
  }
 }
```

Figure 6: Receiver Model in MoDeST (3)

2.3 The Channel Process

The channel process monitors if there is a process sending and if so whether a collision occurs. If this is the case then the process deletes the content of the channel and sets the variable `collision` to true (lines 6 and 8) . The result is that the corresponding receiver process does not get the message.

```
process chan()
{
bool sending_enabled=!true;
do{
 ::when(sending>=1 &&  sending_enabled!=true)
   use_of_channel_detected{=sending_enabled=true,c.free=0=}

 ::when(sending>=2 && collision!=true)  collision_detected
   {=collision=true,c.free=0,number_of_collisions+=1,
     c.m_in.destination=0,c.m_in.content=0,c.m_in.source=0,
     c.m_out.destination=0,c.m_out.content=0,c.m_out.source=0=}

 ::when(sending==0 && sending_enabled )
   channel_is_free{=c.free=1,collision=!true,sending_enabled=!true=}
  }
}
```

Figure 7: Channel Model in MoDeST (3)

2.4 The Energy Model

The energy consumption is modelled through measuring the duration a station is sending, receiving or sleeping and multiplying this durations with appropriate factors. The energy consumption of a station depends on the current activity. There are four energy levels in this work. Transmitting a message costs more energy than receiving a message which costs more energy than being in sleeping respectively in standby mode. There is no energy consumption if the process is not active. The duration that a station is in one of these energy levels is measured using clocks and multiplied with different factors and added in energy consumption variables. The different factors are called `transmission_factor, sensing_factor` and `idle_factor`. The change from sleeping mode to receiving mode takes some time and costs some energy but it is assumed in the model that this change takes no time. The energy consumption is modelled by adding a constant value. The values for these factors have been chosen according to [6].

The following enumerates the energy consumption model in detail:

1. When a process (sender_s) decides to send a new message (based on the load), clock y is reset to zero (action: reset_clock_and_counter). When the process enters `vulnerable period` the value of y is multiplied with the `sensing_factor` and added to the variable `energy_sender[s]` (initial value zero, s$\in$ [1,number_of_senders]).
2. Then the clock x is reset to zero and after the transmission (action send) of the message the value of x is multiplied with the `transmission_factor` and added to `energy_sender[s]`.
3. The clock x is reset again to zero. When the acknowledgement arrives, the value of x is multiplied with the `sensing_factor` and added to `energy_sender[s]`.
4. If no acknowledgement arrives then the value of x is multiplied with the `sensing_factor`, added to `energy_sender[s]` if x equals `ACK_TO`.
5. If a sender waits to start to send a new message based on `load` then the time until the sender decides to send a new message is multiplied with `idle_factor` and added to `energy_sender[s]`.

2.5 The Energy Economy Modes

A nice feature of the following energy economy modes is that they are transparent to the standard which means that there is no difference in the communication behaviour, only the energy consumed by a process is lower. The realization is done by using some additional variables to store the consumed energy in the various economy modes. The advantage of this transparence is that all economy modes can be simulated at the same time.

Mode1: Idea: If the sender notices the channel to be busy in backoff procedure , it is so at least `Trans_time_min` $-$ `ASLOTTIME` and the sender will sleep for this time.

Realization: When the sender s notices the channel busy in the backoff procedure, `Trans_time_min` $-$ `ASLOTTIME` is added to `sleep_mode1[s]` and the value of the constant `turn_on_off_const` is added to `energy_economy_mode[s]`. When the sender s enters `vulnerable period`
(y $-$ `sleep_mode1[s]`)$*$`sensing_factor` $+$ `sleep_mode1[s]`$*$`idle_factor`

is added to energy_sender_economy_mode[s]. The further steps are analogous to above-mentioned energy consumption model 2-5 (energy_sender_economy_mode[s] instead of energy_sender[s]).

Mode2: Idea: The sender sleeps in backoff procedure almost the whole ASLOTTIME. At the start of a new ASLOTTIME the sender looks at the channel and sleeps again.

Realization: When the channel is free for ASLOTTIME in the backoff procedure, ASLOTTIME − 10 is added to sleep_mode2[s] and the value of the constant turn_on_off_const is added to energy_economy_mode2[s]. When the sender s enters vulnerable period (y − sleep_mode2[s])∗sensing_factor + sleep_mode2[s]∗idle_factor is added to energy_sender_economy_mode2[s]. The further steps are analogous to above-mentioned energy consumption model 2-5 (energy_sender_economy_mode2[s] instead of energy_sender[s]).

Mode3: Idea: Combine Mode1 and Mode2

Realization: When the sender notices the channel to be busy in the backoff procedure, Trans_time_min − ASLOTTIME is added to sleep_mode1[s] and the value of the constant turn_on_off_const is added to energy_economy_mode3[s]. When the channel is free for ASLOTTIME in the backoff procedure, ASLOTTIME − 10 is added to sleep_mode2[s] and the value of the constant turn_on_off_const is added to energy_economy_mode3[s]. When the sender s enters vulnerable period (y − sleep_mode1[s] − sleep_mode2[s]) ∗ sensing_factor + (sleep_mode1[s] + sleep_mode2[s])∗idle_factor is added to energy_sender_economy_mode3[s]. The further steps are analogous to above-mentioned energy consumption model 2-5 (energy_sender_economy_mode3[s] instead of energy_sender[s]).

2.6 The Monitor Process

The monitor process records the energy consumption of each sender, the time the channel is used respectively not used, the time the channel is free and the time there is a collision on the channel. This is not done directly in the particular processes (sender, channel), because it is more elegant to segregate the communication model and the energy model. There is a nice feature in MoDeST which can be used for this purpose. Actions can be declared and linked with some assignments. The recording of the values mentioned above is realized by "synchronizing" on actions. This means that the action names that appear in the sender respectively channel process are also present in

the monitor process. Every time an action linked with some assignments is executed in the sender/channel process the action with the same name in the monitor process is executed too and the variables which measure the time a sender transmits, store the energy consumption, channel usage and so on are updated. An example:
If a sender starts transmitting a message, then the monitor measures the time between the actions "start_send" and "send" , multiplies this with the `transmission_factor` and adds to the energy variable of this sender. If the simulation duration is reached the monitor updates the recorded values dependent on the current activity of the corresponding sender.

For further detail of the way the monitor process and the various energy models are implemented in MoDeST, please consult the appendix.

3 Simulation Results

The communication with up to six sender processes, various loads and various values for the parameter `Trans_time_max` has been simulated. The simulation duration is 300000 μseconds. This is long enough to see as the communication works and short enough to keep the time to execute the simulation low. The tool gnuplot has been used to plot the results. All simulations have been executed on a pentium 4 celeron D computer with 2.93 MHZ, 1024 MB RAM and Mandrake Linux 10.0 with kernel 2.4.23 as operating system. The time to execute the simulation series differs from few minutes to about 2 hours.

3.1 Assumptions

The following enumerates the assumptions in this work.

1. At the beginning all sender stations send at the same time.
2. The simulation duration is 300000μs.
3. The change between the energy levels takes no time.
4. A Sender process doesn't receive messages except for acknowledgements.
5. All stations are in the same communication range.

3.2 Energy Consumption Dependent on Maximum Transmission Time

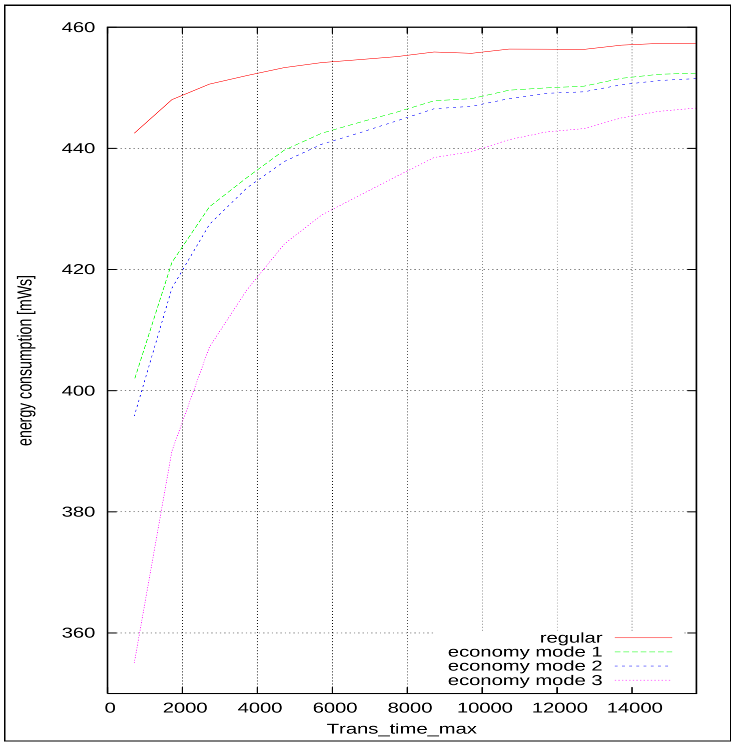


Figure 8: Energy Consumption in IEEE 802.11 Dependent on Max. Transmission Time

In the first experiment series three senders send messages to three receivers and the maximum transmission time is increased step by step from 717μs by 1000μs until it is equal to 15717μs. All senders have a `load` of one which means that they try to send another message directly after receiving an acknowledgement for the last message. Figure 8 shows the regular energy consumption and the energy consumption of the three economy modes for one sender. The energy consumption of the three economy modes is lower than the regular energy consumption independent of the maximum transmission time. The economy mode two consumes slightly less energy than economy mode one. Both consume clearly more energy than economy mode three. The reason is that economy mode three combines the economy modes one and two. The difference between the regular energy consumption and the energy consumption of the economy modes decreases if the maximum transmission time is increased. An explanation for this is that the average transmission time is higher if the maximum transmission time is increased, hence the backoff procedure is called less often and the economy modes can switch less often into sleep mode. The regular energy consumption increases if the maximum transmission time is increased. This can be explained by the fact that the percentage of channel usage increases if the maximum transmission time increases. If there is more activity at the channel, then the senders transmit more often and for a longer duration and the energy consumption increases.

3.3 Duration of Successful Transmissions Dependent on Maximum Transmission Time

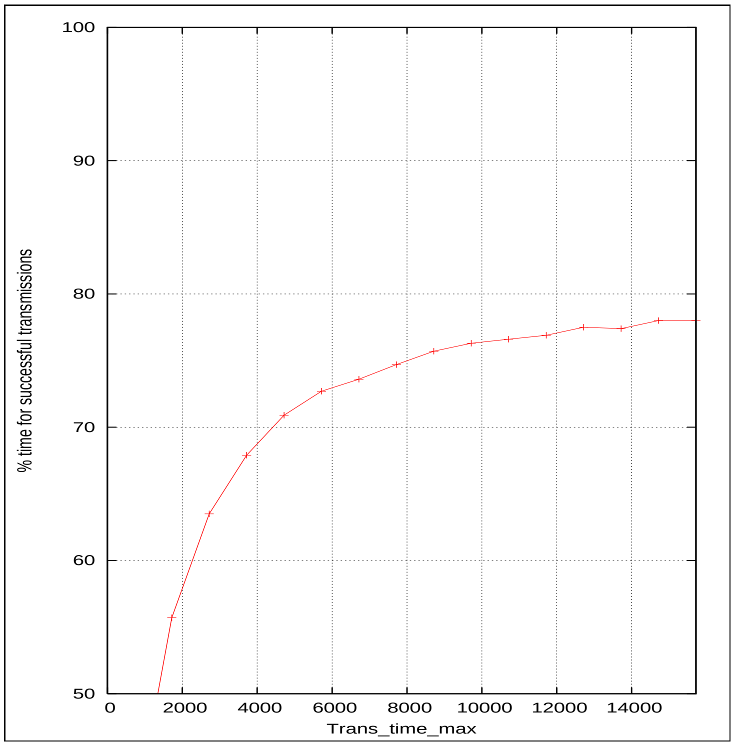


Figure 9: Successful Transmission Times Dependent on Max. Transmission Time

Figure 9 shows the above-mentioned percentage of time in which the senders transmit successfully messages dependent on the maximum transmission time. The situation is the same as in Figure 8. There are three senders with a load of one and three receivers. The percentage of time increases fast from about 50 per cent in the case of 717μs maximum transmission to about 5 per cent in the case of 8000μs maximum transmission time. When the maximum transmission time increases further to 15717μs the percentage of channel usage increases only about to 78 per cent.

3.4 Channel Usage Dependent on Load

In a further experiment series up to six senders send messages to up to six receivers. The percentage of channel usage dependent on the `load` of the senders is observed. Channel usage means the time where at least one sender or one receiver transmits a message and so includes the time when there is a collision. As shown in Figure 10 the percentage of channel usage increases a lot if the load of the senders increases from 0.01 to 0.11 while the percentage of channel usage decreases only moderately if the load increases from 0.11 to 1. If the load is equal to one the percentage of channel usage is nearly the same independent on the number of senders. If the load is equal to 0.01 the percentage of channel usage is less than 20 in the the case of one sender and more than 80 in the case of six senders.

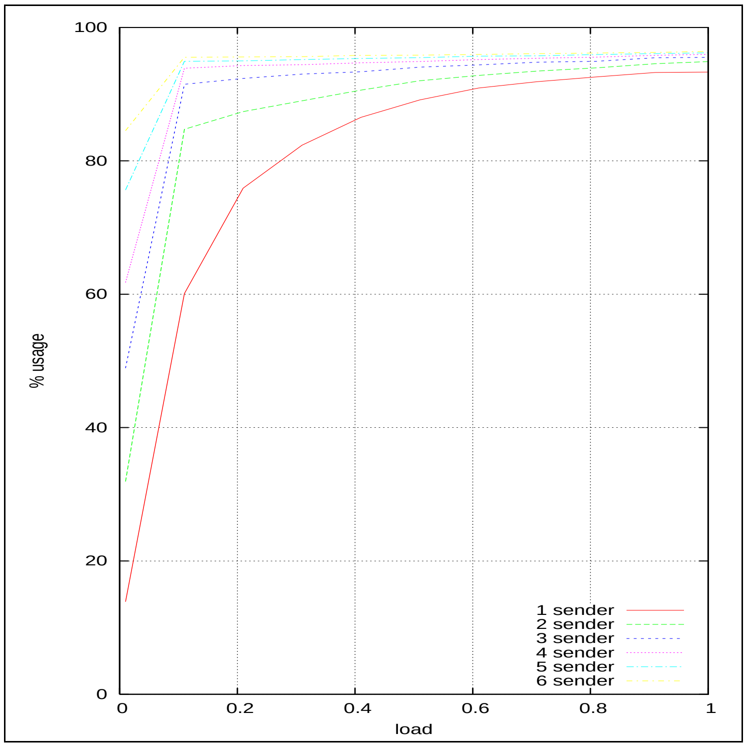


Figure 10: Channel Usage Percentage in IEEE 802.11

3.5 Effects on Transmission Times in Cheating Case

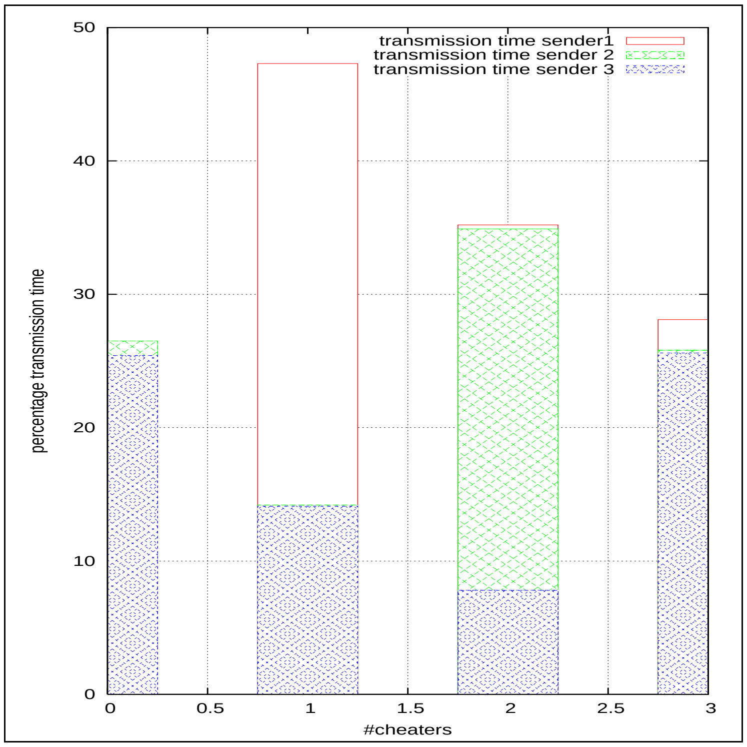


Figure 11: Transmission Times in IEEE 802.11 in Cheating Case (3 Senders)

In a further experiment series three senders send messages to three receivers. The senders have a `load` of one. The effects on the transmission time of each sender, that occur if one, two or all senders try to increase their transmission time, are observed. In the model the backoff procedure is executed if a sender wants to send again directly after it has successfully sent a message. However if a sender wants to send another message to a later moment, it doesn't perform the backoff procedure but if the channel is free for `DIFS` time units it starts transmitting. This is the background for a cheating idea in the case of high loads. If a sender wants to send more often, it waits after sending a message a short time. After this it tries to send without using the backoff procedure. This will be successful if the other senders are performing the backoff procedure and no sender will finish the backoff procedure within `DIFS` time units. If a sender finishes the backoff procedure within `DIFS` time units there are two possibilities. There is a collision because both transmit at the same time or the sender that has tried to get an advantage of not performing the backoff procedure has to perform this procedure because the channel is busy. Figure 11 shows the transmission times for three senders in the described situation. In case of i cheaters sender 1 until sender i cheat. If no sender cheats, all senders have nearly the same transmission times. This changes dramaticly if sender one cheats. The transmission time of this sender increases about 75 per cent while the transmission times of sender two and three decrease by about 50 per cent. If sender one and sender two cheat, the transmission times of both increase by about 37.5 per cent and the transmission time of sender three decreases by about 65 per cent in comparison to the situation in which no sender cheats. When all senders cheat, no sender gets an advantage and the transmission times are similar to the situation in which no sender cheats.

Figure 12 shows the transmission times of six senders with a `load` of one dependent on the number of senders that cheat by not performing the backoff procedure as described above. All senders have nearly the same transmission time if no sender cheats and if all senders cheat. As in the situation with 3 senders the transmission times of senders that cheat increase while the transmission times of the other senders decrease.

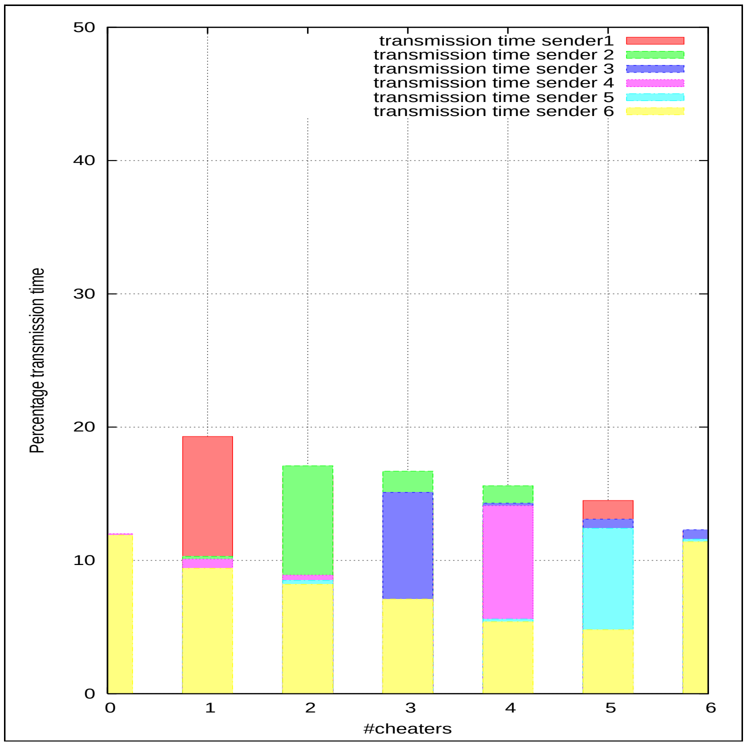


Figure 12: Transmission Times in IEEE 802.11 in Cheating Case (6 senders)

3.6 Collision Times Dependent on Load

Figure 13 shows the percentage of collision times dependent on the `load` of the senders. In this experiment series up to six senders send messages to up to six receivers. The collision time is defined as the time in which the senders cannot use the channel because of a collision. The `load` increases from 0.01 to one. If the number of senders increases, the collision times increase too. This is a behaviour which is expected. The behaviour that is not expected is that the collision times increase in the case of the loads are equal to 0.01 up to 0.11 and then decrease if the loads are equal to 0.21 to one. In the case of six senders and a load of 0.1 the percentage of collision times is greater is almost 50. If the load increases to one the percentage of collision times decreases to less than 25.

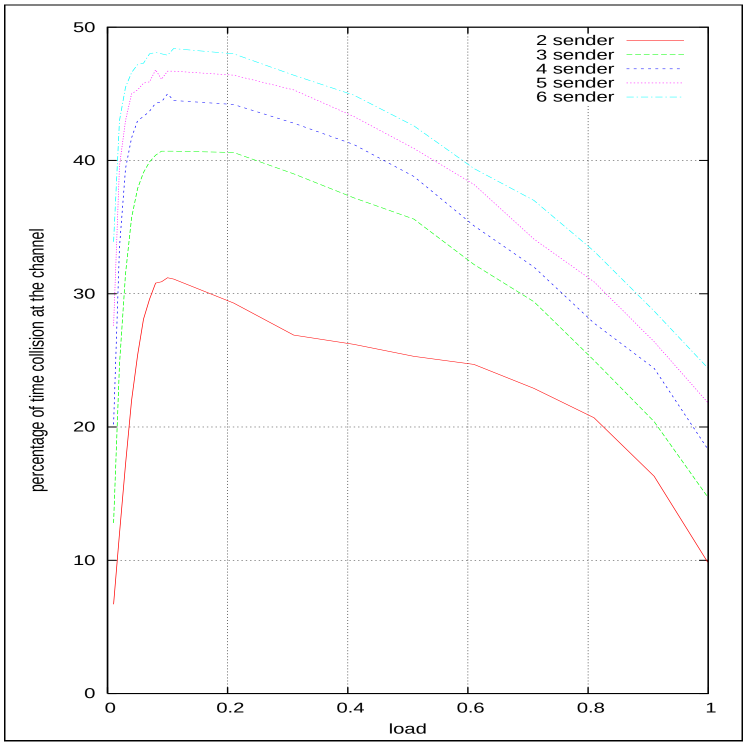


Figure 13: Collision Times in IEEE 802.11 (a)

The percentage of collision times as shown in Figure 13 raises the question why the collision times in the case of a load of 0.1 are higher than those in the case of a `load` of one. The mechanism that prevents or minimises collisions is the backoff procedure. In the model a sender does not perform this procedure if it wants to send another message which does not directly arrive after successfully sending a message. A further experiment series inspects if this behaviour causes the large collision times in the case of the low loads. For this purpose there are again up to six senders and six receivers that communicate. The `load` increases again from 0.01 to one. The difference between these series is that in the new series a sender performs the backoff procedure every time it wants to send a new message.

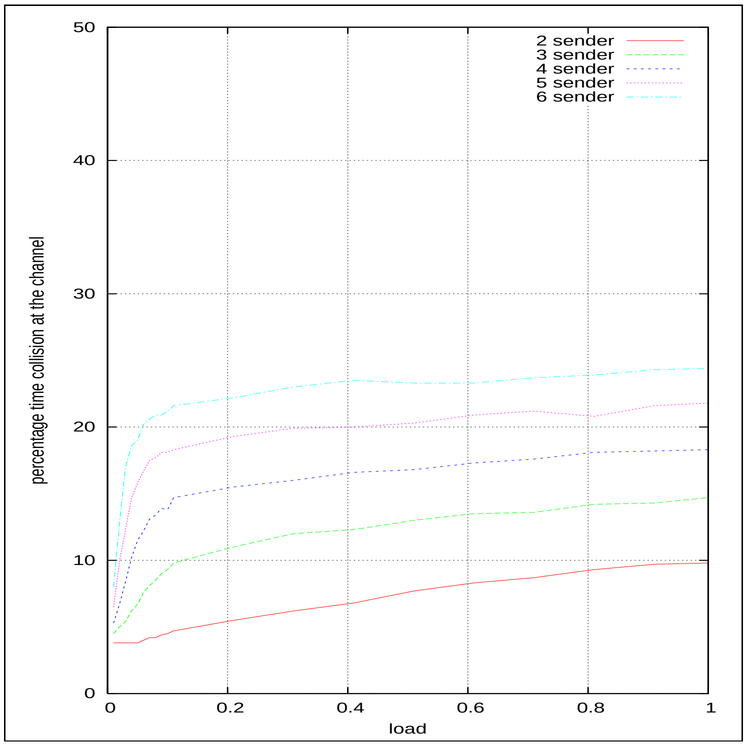


Figure 14: Collision Times in IEEE 802.11 (b)

Figure 14 shows the percentage of collision times in this situation. The collision times increase if there are more senders and if the `load` of these senders increases. In the case of six senders and a `load` of 0.1 the percentage of collision times is less than 25. The reduction of the collision times in this experiment series shows that it is useful to perform the backoff procedure every time when a new message has to be sent.

4 Conclusion

In this work the communication between stations according to 802.11 in ad hoc modus using the basic access procedure is modelled and simulated. Furthermore three energy economy modes are introduced and their performance dependent on various maximum transmission times is evaluated. Moreover the duration of collisions on the channel with various loads of some senders and various rules of calling the backoff procedure is simulated. The effects if one or more sender try to increase the throughput are investigated too.

During the development the following experiences with the language MoDeST and the tools MoTor and Möbius has been collected:
The syntax of MoDeST is similar to C and Promela such that it looks quite familiar. Although the language knows the keyword true, the counterpart false is not known. Further it is not possible to declare an array of clocks, which could be useful in a monitor process as described in this work. The command Uniform() only returns float values. An option for integer values would be useful. The palt construction only accepts constant integer values as parameters. Experiment series with various probabilities using this construct are not possible. If the relabeling feature for actions is used, there is no relabeling in the debugging mode 1. This means that it is not possible to see which instance of a process has taken this action.

References

[1] Joost-Pieter Katoen, Henrik Bohnenkamp, Ric Klaren and Holger Hermanns. Embedded Software Analysis with MoTor. Lecture Notes in Computer Science 3185, Springer-Verlag, pp. 268-293, 2004.

[2] Ric Klaren. MoDeST Language Manual. Formal Methods and Tools Group, University of Twente. http://fmt.cs.utwente.nl/tools/motor/

[3] ANSI/IEEE Std 802.11, 1999 Edition (R2003).
http://standards.ieee.org/getieee802/download/802.11-1999.pdf

[4] Marta Kwiatkowska, Gethin Norman and Jeremy Sproston. Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. Lecture Notes in Computer Science 2399, Springer-Verlag, pp. 169, 2002.

[5] Pedro R. D'Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST - A Modelling and Description Language for Stochas-

tic Timed Systems. Lecture Notes in Computer Science 2165, Springer-Verlag, pp. 87, 2001.

[6] Paul J.M. Havinga, Gerard J.M. Smit. Energy-Efficient Wireless Networking for Multimedia Applications. Wireless Communications and Mobile Computing, Vol.1, Wiley, pp. 165-184, 2001.

[7] Möbius: User Manual. PERFORM Performability Engineering Research Group, University of Illinois at Urbana-Champaign, 2005. http://www.mobius.uiuc.edu/

5 Appendix

5.1 Source Code

```
action  enable_backoff , start_wait_for_ack , reset_clock_and_counter ,
        choose_backoffvalue , continuebackoff , count_down_backoffvalue ,
        freeze_backoff , start_send , send , exit_backoff ,
        deals_with_backoff , reset_sense_in_backoff , do_nothing , do_nothing1 ,
        do_nothing2 , do_nothing3 , do_nothing4 , do_nothing5 , do_nothing6 ,
        will_resend_message , ack_received_correct , waited_DIFS_backoff ,
        enter_vulnerable_period , will_skip_message ,
        detects_channel_busy_after_sending_and_will_resend ,

        s1_enable_backoff , s1_start_wait_for_ack , s1_reset_clock_and_counter ,
        s1_choose_backoffvalue , s1_continue_backoff , s1_count_down_backoffvalue ,
        s1_freeze_backoff , s1_start_send , s1_send , s1_exit_backoff ,
        s1_deals_with_backoff , s1_reset_sense_in_backoff , s1_do_nothing ,
        s1_will_resend_message , s1_ack_received_correct , s1_waited_DIFS_backoff ,
        s1_enter_vulnerable_period , s1_will_skip_message ,
        s1_detects_channel_busy_after_sending_and_will_resend ,

        s2_enable_backoff , s2_start_wait_for_ack , s2_reset_clock_and_counter ,
        s2_choose_backoffvalue , s2_continue_backoff , s2_count_down_backoffvalue ,
        s2_freeze_backoff , s2_start_send , s2_send , s2_exit_backoff ,
        s2_deals_with_backoff , s2_reset_sense_in_backoff , s2_do_nothing ,
        s2_will_resend_message , s2_ack_received_correct , s2_waited_DIFS_backoff ,
        s2_enter_vulnerable_period , s2_will_skip_message ,
        s2_detects_channel_busy_after_sending_and_will_resend ,

        s3_enable_backoff , s3_start_wait_for_ack , s3_reset_clock_and_counter ,
        s3_choose_backoffvalue , s3_continue_backoff , s3_count_down_backoffvalue ,
        s3_freeze_backoff , s3_start_send , s3_send , s3_exit_backoff ,
        s3_deals_with_backoff , s3_reset_sense_in_backoff , s3_do_nothing ,
        s3_will_resend_message , s3_ack_received_correct , s3_waited_DIFS_backoff ,
        s3_enter_vulnerable_period , s3_will_skip_message ,
        s3_detects_channel_busy_after_sending_and_will_resend ,

        s4_enable_backoff , s4_start_wait_for_ack , s4_reset_clock_and_counter ,
        s4_choose_backoffvalue , s4_continue_backoff , s4_count_down_backoffvalue ,
        s4_freeze_backoff , s4_start_send , s4_send , s4_exit_backoff ,
        s4_deals_with_backoff , s4_reset_sense_in_backoff , s4_do_nothing ,
        s4_will_resend_message , s4_ack_received_correct , s4_waited_DIFS_backoff ,
        s4_enter_vulnerable_period , s4_will_skip_message ,
        s4_detects_channel_busy_after_sending_and_will_resend ,

        s5_enable_backoff , s5_start_wait_for_ack , s5_reset_clock_and_counter ,
        s5_choose_backoffvalue , s5_continue_backoff , s5_count_down_backoffvalue ,
        s5_freeze_backoff , s5_start_send , s5_send , s5_exit_backoff ,
        s5_deals_with_backoff , s5_reset_sense_in_backoff , s5_do_nothing ,
        s5_will_resend_message , s5_ack_received_correct , s5_waited_DIFS_backoff ,
        s5_enter_vulnerable_period , s5_will_skip_message ,
        s5_detects_channel_busy_after_sending_and_will_resend ,

        s6_enable_backoff , s6_start_wait_for_ack , s6_reset_clock_and_counter ,
        s6_choose_backoffvalue , s6_continue_backoff , s6_count_down_backoffvalue ,
        s6_freeze_backoff , s6_start_send , s6_send , s6_exit_backoff ,
        s6_deals_with_backoff , s6_reset_sense_in_backoff , s6_do_nothing ,
        s6_will_resend_message,s6_ack_received_correct,s6_waited_DIFS_backoff,
        s6_enter_vulnerable_period,s6_will_skip_message,
        s6_detects_channel_busy_after_sending_and_will_resend,

        receive,start_send_ack,send_ack,
        r1_receive,r1_start_send_ack,r1_send_ack,
        r2_receive,r2_start_send_ack,r2_send_ack,
        r3_receive,r3_start_send_ack,r3_send_ack,
        r4_receive,r4_start_send_ack,r4_send_ack,
        r5_receive,r5_start_send_ack,r5_send_ack,
        r6_receive,r6_start_send_ack,r6_send_ack,

        collision_detected,channel_is_free,
        use_of_channel_detected,channel_corrupted,
        channel_ready_after_corrupted;

const int SIFS = 28;
const int DIFS = 128;
const int aCWmin = 15;
const int aCWmax = 1023;
const int ACK_TO = 300;
const int ACK_DURATION= 205;
const int ASLOTTIME=50;
const int VULN=48;
const int TRANS_TIME_MIN=224;
const int number_of_senders = 6;

extern const int TRANS_TIME_MAX;   //original value 15717
extern const float simulation_duration;
extern const float transmitting_factor;//=1.625;
extern const float sensing_factor;//=1.475;
extern const float idle_factor; //=0.08;
extern const float turn_on_off_const;
extern const int chan_corrupted_max;
extern const int chan_corrupted_min;
extern const int   t_intervall;
extern const float turn_on_off_const2;
extern const float load1;
extern const int active_senders;
extern const float probability_chan_corrupted;

clock global_time;
int messages_send[number_of_senders+1];
float sending_duration[number_of_senders+1];
float dur_chan_corrupted;
int ack;
int sending;
float is_sending[number_of_senders+1];
int number_of_collisions;
bool receiver_is_sending;
bool collision;
float coll=0;
float time_in_coll=0;
float energy_sender[number_of_senders+1];
float energy_sender_economy_mode[number_of_senders+1];
float energy_sender_economy_mode2[number_of_senders+1];
float energy_sender_economy_mode3[number_of_senders+1];
float sleep_mode1[number_of_senders+1];
float sleep_mode2[number_of_senders+1];
float time_sender_send[number_of_senders+1];
float time_sender_channel_used_not_successful[number_of_senders+1];
float time_receiver_send;
float time_channel_used;
float time_channel_not_used;
float time_channel_error;
float vuln_reached[number_of_senders+1];
float msg_to_send[number_of_senders+1];

typedef struct {          //represents a message
        int source;
        int destination;
        int content;
} MESSAGE;

typedef struct {         //represents the channel
float free;
MESSAGE m_in;
MESSAGE m_out;
} CHANNEL;

CHANNEL c;

process sender(int id,float load,int active)
```

```
{
clock x,y;
float backoff_value;
int CW=aCWmin;
bool wait_DIFS=true;
bool backoffswitch=!true;
bool continue_backoff=!true;
bool backoff_finished=!true;
bool initialsend=true;
bool ready_to_send=true;
float random_choice;
c.free=1;
alt{
   ::when(active<id)tau;when(global_time==simulation_duration) tau
   ::when(active>=id)
do{
  ::when (ready_to_send!=true && initialsend!=true && y==634)
    random_choice=Uniform(0,1);
    alt{
       ::when (load>=random_choice) ready_to_send=true
       ::when (load<random_choice)
         do_nothing{=y=0,backoffswitch=!true,msg_to_send[id]=0=}
       }
  ::when(ready_to_send || initialsend)
    reset_clock_and_counter{=x=0,y=0,backoff_finished=!true,
                              continue_backoff=!true,
                              wait_DIFS=true,initialsend=!true,
                              sleep_mode1[id]=0,
                              sleep_mode2[id]=0,msg_to_send[id]=1,
                              ready_to_send=!true=};
       do{
         ::urgent(c.free==0 && backoffswitch!=true &&
                  continue_backoff!=true)
           when  (c.free==0 && backoffswitch!=true &&
                  continue_backoff!=true)
                  s1_enable_backoff {= backoffswitch=true =}


                                 //  the "backoff-procedure"
  ::when((backoffswitch || continue_backoff) && backoff_value>=0)
    deals_with_backoff{= x=0 =};
    alt{
      ::when (wait_DIFS)
        alt{
          ::when(x<DIFS && c.free==0 && wait_DIFS)
            reset_sense_in_backoff{=x=0=};when(c.free==1) tau
          ::when(x>DIFS && c.free==1)
            waited_DIFS_backoff{=wait_DIFS=!true=}
           }
      ::when (wait_DIFS!=true)
        alt{
          ::when(backoffswitch)choose_backoffvalue
            {=backoffswitch=!true,continue_backoff=true,x=0,
              backoff_value=Uniform(-1,CW)=}
          ::when(continue_backoff)continuebackoff
            {=backoffswitch=!true,continue_backoff=true,x=0=}
           };

        alt{
          ::when(backoff_value<0) tau
          ::when(x>=ASLOTTIME && c.free==1)
            count_down_backoffvalue{=backoff_value-=1=}
          ::when(x>=ASLOTTIME && c.free==0)
            freeze_backoff{=continue_backoff=true,
                            wait_DIFS=true=};
            when (c.free==1) tau
           }
       }

  ::when(backoff_value < 0)
    exit_backoff{=backoffswitch=!true,continue_backoff=!true,
                  backoff_finished=true,backoff_value=0=}

     //enter vulnerable period and send
  ::when((x>DIFS || backoff_finished) && backoffswitch!=true
         && continue_backoff!=true)
    enter_vulnerable_period
    {=vuln_reached[id]=1.0,x=0,
      sending_duration[id]=Uniform(TRANS_TIME_MIN,TRANS_TIME_MAX)=};
    when(x>= VULN) start_send
    {=sending+=1,is_sending[id]=1,x=0,backoffswitch=true,
      c.m_in.destination=-id,c.m_in.source=id,
      c.m_in.content=5=};
     urgent (x>=sending_duration[id])
     when   (x>=sending_duration[id])
     send{=c.m_out.source=c.m_in.source,
            c.m_out.content=c.m_in.content,
            c.m_out.destination=c.m_in.destination,
            sending-=1,is_sending[id]=0=};
     x=0;

     alt{
        ::when(x==0 && sending>0)
          detects_channel_busy_after_sending_and_will_resend
          {= initialsend=true,CW=(CW+1)*2-1,vuln_reached[id]=0 =};break
        ::when(x==0 && sending==0) start_wait_for_ack {=x=0=};
          alt{
            ::urgent(c.m_out.destination==id && x>= SIFS+ ACK_DURATION)
              when  (c.m_out.destination==id && x>= SIFS+ ACK_DURATION)
              ack_received_correct{=CW=aCWmin,messages_send[id]+=1,
              vuln_reached[id]=0=};break
            ::when (x>ACK_TO && c.m_out.destination !=id)
              alt{
                ::when(CW< aCWmax)
                  will_resend_message
                  {=initialsend=true,CW=(CW+1)*2-1,vuln_reached[id]=0=};break
                ::when(CW>=aCWmax) will_skip_message
                  {= vuln_reached[id]=0 =};break
                }
             }
        }
  }
 }
}

}

process receiver(int id)
{
clock y;
int source=0;
do{
  ::when(c.m_out.destination==id && collision!=true)
    receive{=y=0,c.m_out.destination=0,source=c.m_out.source=};
    urgent(y>=SIFS) when(y>=SIFS) start_send_ack
    {=y=0,ack+=1,c.m_in.destination=source,c.m_in.source=id,
      receiver_is_sending=true,sending+=1=};
    urgent(y>=ACK_DURATION) when(y>=ACK_DURATION)
    send_ack{=c.m_out.source=c.m_in.source,
              c.m_out.destination=c.m_in.destination,
              receiver_is_sending=!true, sending-=1,
              time_receiver_send+=y=}
  }
}

process chan()
{
clock timer1,timer_coll,disturb;
float random_choice;

bool sending_enabled=!true;
do{
  ::when(sending>=1 && sending_enabled!=true)random_choice=Uniform(0,1);
    alt{
        ::when(random_choice>probability_chan_corrupted)
          use_of_channel_detected{=sending_enabled=true,c.free=0=}
        ::when(random_choice<=probability_chan_corrupted)
          channel_corrupted
          {=collision=true,c.free=0,sending_enabled=true,disturb=0,
            c.m_in.destination=0,c.m_in.content=0,c.m_in.source=0,
            c.m_out.destination=0,c.m_out.content=0,c.m_out.source=0,
            dur_chan_corrupted=Uniform(chan_corrupted_min,chan_corrupted_max),
            time_channel_error+=dur_chan_corrupted=};
          when(disturb==dur_chan_corrupted)channel_ready_after_corrupted
                                          {= collision=!true=}
                                                         }
  ::when(sending>=2 && collision!=true)  collision_detected
    {= collision=true,c.free=0,number_of_collisions+=1,
        c.m_in.destination=0,c.m_in.content=0,c.m_in.source=0,
        c.m_out.destination=0,c.m_out.content=0,c.m_out.source=0=}

  ::when(sending==0 && sending_enabled )
    channel_is_free{=c.free=1,collision=!true,sending_enabled=!true=}
  }
}
```

```
process monitor()
{
clock x,y,z,x2,y2,z2,x3,y3,z3,timer1,timer_coll,
      x4,y4,z4,x5,y5,z5,x6,y6,z6;
int on_off[number_of_senders+1];
do{
  ::s1_do_nothing{=energy_sender_economy_mode[1]+=turn_on_off_const*on_off[1],
    energy_sender_economy_mode2[1]+=turn_on_off_const*on_off[1],
    energy_sender[1]+=turn_on_off_const*on_off[1],
    energy_sender_economy_mode3[1]+=turn_on_off_const*on_off[1],on_off[1]=0=}
  ::s1_reset_clock_and_counter{=x=0,y=0,
    energy_sender[1]+=z*idle_factor+z*turn_on_off_const2/t_intervall,on_off[1]=1;
    energy_sender_economy_mode[1] += z*idle_factor+z*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode2[1] += z*idle_factor+z*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode3[1] += z*idle_factor+z*turn_on_off_const2/t_intervall=}
  ::s1_freeze_backoff{=sleep_mode1[1]+=TRANS_TIME_MIN−ASLOTTIME,
    energy_sender_economy_mode[1]+=turn_on_off_const,
    energy_sender_economy_mode3[1]+=turn_on_off_const=}
  ::s1_count_down_backoffvalue{=sleep_mode2[1]+=ASLOTTIME−10,
    energy_sender_economy_mode2[1]+=turn_on_off_const,
    energy_sender_economy_mode3[1]+=turn_on_off_const=}
  ::s1_enter_vulnerable_period{=energy_sender[1]+=y*sensing_factor,
    energy_sender_economy_mode[1]+=(y−sleep_mode1[1])*
    sensing_factor+sleep_mode1[1]*idle_factor,
    energy_sender_economy_mode2[1] += (y−sleep_mode2[1])*
    sensing_factor+sleep_mode2[1]*idle_factor,
    energy_sender_economy_mode3[1] += (y−sleep_mode1[1]−sleep_mode2[1])*
    sensing_factor+(sleep_mode1[1]+sleep_mode2[1])*idle_factor=}
  ::s1_start_send{= x=0 =}
  ::s1_send{= energy_sender[1]+=transmitting_factor*x,
    energy_sender_economy_mode[1]+=transmitting_factor*x,
    energy_sender_economy_mode2[1]+=transmitting_factor*x,
    energy_sender_economy_mode3[1]+=transmitting_factor*x, x=0=}
  ::s1_detects_channel_busy_after_sending_and_will_resend{= z=0,
    time_sender_channel_used_not_successful[1]+=sending_duration[1] =}
  ::s1_ack_received_correct{=energy_sender[1]+=sensing_factor*x,
    energy_sender_economy_mode[1]+=sensing_factor*x,z=0,
    time_sender_send[1]+=sending_duration[1],
    energy_sender_economy_mode2[1]+=sensing_factor*x,
    energy_sender_economy_mode3[1]+=sensing_factor*x=}
  ::s1_will_resend_message{=energy_sender[1]+=sensing_factor*x,
    energy_sender_economy_mode[1]+=sensing_factor*x,z=0,
    energy_sender_economy_mode2[1]+=sensing_factor*x,
    energy_sender_economy_mode3[1]+=sensing_factor*x,
    time_sender_channel_used_not_successful[1]+=sending_duration[1]=}
  ::s1_will_skip_message{=energy_sender[1]+=sensing_factor*x,
    energy_sender_economy_mode[1]+=sensing_factor*x,z=0,
    energy_sender_economy_mode2[1]+=sensing_factor*x,
    energy_sender_economy_mode3[1]+=sensing_factor*x,
    time_sender_channel_used_not_successful[1]+=sending_duration[1]=}

  ::s2_do_nothing{=energy_sender_economy_mode[2]+=turn_on_off_const*on_off[2],
    energy_sender_economy_mode2[2]+=turn_on_off_const*on_off[2],
    energy_sender[2]+=turn_on_off_const*on_off[2],
    energy_sender_economy_mode3[2]+=turn_on_off_const*on_off[2],on_off[2]=0=}
  ::s2_reset_clock_and_counter{=x2=0,y2=0,
    energy_sender[2]+=z2*idle_factor+z2*turn_on_off_const2/t_intervall,on_off[2]=1,
    energy_sender_economy_mode[2] += z2*idle_factor+z2*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode2[2] += z2*idle_factor+z2*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode3[2] += z2*idle_factor+z2*turn_on_off_const2/t_intervall=}
  ::s2_freeze_backoff{=sleep_mode1[2]+=TRANS_TIME_MIN−ASLOTTIME,
    energy_sender_economy_mode[2]+=turn_on_off_const,
    energy_sender_economy_mode3[2]+=turn_on_off_const=}
  ::s2_count_down_backoffvalue{=sleep_mode2[2]+=ASLOTTIME−10,
    energy_sender_economy_mode2[2]+=turn_on_off_const,
    energy_sender_economy_mode3[2]+=turn_on_off_const=}
  ::s2_enter_vulnerable_period{=energy_sender[2]+=y2*sensing_factor,
    energy_sender_economy_mode[2]+=(y2−sleep_mode1[2])*
    sensing_factor+sleep_mode1[2]*idle_factor,
    energy_sender_economy_mode2[2] += (y2−sleep_mode2[2])*
    sensing_factor+sleep_mode2[2]*idle_factor,
    energy_sender_economy_mode3[2] += (y2−sleep_mode1[2]−sleep_mode2[2])*
    sensing_factor+(sleep_mode1[2]+sleep_mode2[2])*idle_factor=}
  ::s2_start_send{= x2=0 =}
  ::s2_send{=energy_sender[2]+=transmitting_factor*x2,
    energy_sender_economy_mode[2]+=transmitting_factor*x2,
    energy_sender_economy_mode2[2]+=transmitting_factor*x2,
    energy_sender_economy_mode3[2]+=transmitting_factor*x2, x2=0 =}
  ::s2_detects_channel_busy_after_sending_and_will_resend{= z2=0,
    time_sender_channel_used_not_successful[2]+=sending_duration[2]=}
  ::s2_ack_received_correct{=energy_sender[2]+=sensing_factor*x2,
    energy_sender_economy_mode[2]+=sensing_factor*x2,z2=0,
    time_sender_send[2]+=sending_duration[2],
    energy_sender_economy_mode2[2]+=sensing_factor*x2,
    energy_sender_economy_mode3[2]+=sensing_factor*x2=}
  ::s2_will_resend_message{=energy_sender[2]+=sensing_factor*x2,
    energy_sender_economy_mode[2]+=sensing_factor*x2,z2=0,
    energy_sender_economy_mode2[2]+=sensing_factor*x2,
    energy_sender_economy_mode3[2]+=sensing_factor*x2,
    time_sender_channel_used_not_successful[2]+=sending_duration[2]=}
  ::s2_will_skip_message;energy_sender[2]+=sensing_factor*x2,
    energy_sender_economy_mode[2]+=sensing_factor*x2, z2=0,
    energy_sender_economy_mode2[2]+=sensing_factor*x2,
    energy_sender_economy_mode3[2]+=sensing_factor*x2,
    time_sender_channel_used_not_successful[2]+=sending_duration[2]=}

  ::s3_do_nothing{=energy_sender_econmy_mode[3]+=turn_on_off_const*on_off[3],
    energy_sender_economy_mode2[3]+=turn_on_off_const*on_off[3],
    energy_sender[3]+=turn_on_off_const*on_off[3],
    energy_sender_economy_mode3[3]+=turn_on_off_const*on_off[3],on_off[3]=0=}
  ::s3_reset_clock_and_counter{=x3=0,y3=0,
    energy_sender[3]+=z3*idle_factor+z3*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode[3] += z3*idle_factor+z3*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode2[3] += z3*idle_factor+z3*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode3[3] += z3*idle_factor+z3*turn_on_off_const2/t_intervall=}
  ::s3_freeze_backoff{=sleep_mode1[3]+=TRANS_TIME_MIN-ASLOTTIME,
    energy_sender_economy_mode[3]+=turn_on_off_const,
    energy_sender_economy_mode3[3]+=turn_on_off_const=}
  ::s3_count_down_backoffvalue{=sleep_mode2[3]+=ASLOTTIME-10,
    energy_sender_economy_mode2[3]+=turn_on_off_const,
    energy_sender_economy_mode3[3]+=turn_on_off_const=}
  ::s3_enter_vulnerable_period{=energy_sender[3]+=y3*sensing_factor,
    energy_sender_economy_mode[3]+=(y3-sleep_mode1[3])*
    sensing_factor+sleep_mode1[3]*idle_factor,
    energy_sender_economy_mode2[3] += (y3-sleep_mode2[3])*
    sensing_factor+sleep_mode2[3]*idle_factor,
    energy_sender_economy_mode3[3] += (y3-sleep_mode1[3]-sleep_mode2[3])*
    sensing_factor+(sleep_mode1[3]+sleep_mode2[3])*idle_factor =}
  ::s3_start_send {= x3=0 =}
  ::s3_send{= energy_sender[3]+=transmitting_factor*x3,
    energy_sender_economy_mode[3]+=transmitting_factor*x3,
    energy_sender_economy_mode2[3]+=transmitting_factor*x3,
    energy_sender_economy_mode3[3]+=transmitting_factor*x3,x3=0
  ::s3_detects_channel_busy_after_sending_and_will_resend ,z3=0,
    time_sender_channel_used_not_successful[3]+=sending_duration[3]=}
  ::s3_ack_received_correct{=energy_sender[3]+=sensing_factor*x3,
    energy_sender_economy_mode[3]+=sensing_factor*x3, z3=0,
    time_sender_send[3]+=sending_duration[3],
    energy_sender_economy_mode2[3]+=sensing_factor*x3,
    energy_sender_economy_mode3[3]+=sensing_factor*x3=}
  ::s3_will_resend_message{=energy_sender[3]+=sensing_factor*x3,
    energy_sender_economy_mode[3]+=sensing_factor*x3;z3=0,
    energy_sender_economy_mode2[3]+=sensing_factor*x3,
    energy_sender_economy_mode3[3]+=sensing_factor*x3,
    time_sender_channel_used_not_successful[3]+=sending_duration[3]=}
  ::s3_will_skip_message{=energy_sender[3]+=sensing_factor*x3,
    energy_sender_economy_mode[3]+=sensing_factor*x3, z3=0,
    energy_sender_economy_mode2[3]+=sensing_factor*x3,
    energy_sender_economy_mode3[3]+=sensing_factor*x3,
    time_sender_channel_used_not_successful[3]+=sending_duration[3]=}

  ::s4_do_nothing{=energy_sender_economy_mode[4]+=turn_on_off_const*on_off[4],
    energy_sender_economy_mode2[4]+=turn_on_off_const*on_off[4],
    energy_sender[4]+=turn_on_off_const*on_off[4],
    energy_sender_economy_mode3[4]+=turn_on_off_const*on_off[4],on_off[4]=0=}
  ::s4_reset_clock_and_counter{=x4=0,y4=0,
    energy_sender[4]+=z4*idle_factor+z4*turn_on_off_const2/t_intervall;on_off[4]=1,
    energy_sender_economy_mode[4] += z4*idle_factor+z4*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode2[4] += z4*idle_factor+z4*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode3[4] += z4*idle_factor+z4*turn_on_off_const2/t_intervall=}
  ::s4_freeze_backoff{=sleep_mode1[4]+=TRANS_TIME_MIN-ASLOTTIME,
    energy_sender_economy_mode[4]+=turn_on_off_const,
    energy_sender_economy_mode3[4]+=turn_on_off_const=}
  ::s4_count_down_backoffvalue{=sleep_mode2[4]+=ASLOTTIME-10,
    energy_sender_economy_mode2[4]+=turn_on_off_const,
    energy_sender_economy_mode3[4]+=turn_on_off_const=}
  ::s4_enter_vulnerable_period{=energy_sender[4]+=y4*sensing_factor,
    energy_sender_economy_mode[4]+=(y4-sleep_mode1[4])*sensing_factor+
    sleep_mode1[4]*idle_factor,
    energy_sender_economy_mode2[4] += (y4-sleep_mode2[4])*
    sensing_factor+sleep_mode2[4]*idle_factor,
    energy_sender_economy_mode3[4] += (y4-sleep_mode1[4]- sleep_mode2[4])*
    sensing_factor+(sleep_mode1[4]+sleep_mode2[4])*idle_factor =}
  ::s4_start_send{= x4=0 =}
  ::s4_send{= energy_sender[4]+=transmitting_factor*x4,
    energy_sender_economy_mode[4]+=transmitting_factor*x4,
    energy_sender_economy_mode2[4]+=transmitting_factor*x4,
    energy_sender_economy_mode3[4]+=transmitting_factor*x4, x4=0=}
  ::s4_detects_channel_busy_after_sending_and_will_resend{=z4=0,
    time_sender_channel_used_not_successful[4]+=sending_duration[4]=}
  ::s4_ack_received_correct{=energy_sender[4]+=sensing_factor*x4,
    energy_sender_economy_mode[4]+=sensing_factor*x4,z4=0,
    time_sender_send[4]+=sending_duration[4],
    energy_sender_economy_mode2[4]+=sensing_factor*x4,
    energy_sender_economy_mode3[4]+=sensing_factor*x4=}
  ::s4_will_resend_message{=energy_sender[4]+=sensing_factor*x4,
    energy_sender_economy_mode[4]+=sensing_factor*x4,z4=0,
    energy_sender_economy_mode2[4]+=sensing_factor*x4,
    energy_sender_economy_mode3[4]+=sensing_factor*x4,
    time_sender_channel_used_not_successful[4]+=sending_duration[4]=}
  ::s4_will_skip_message{=energy_sender[4]+=sensing_factor*x4,
    energy_sender_economy_mode[4]+=sensing_factor*x4,z4=0,
    energy_sender_economy_mode2[4]+=sensing_factor*x4,
    energy_sender_economy_mode3[4]+=sensing_factor*x4,
    time_sender_channel_used_not_successful[4]+=sending_duration[4]=}

  ::s5_do_nothing{=energy_sender_economy_mode[5]+=turn_on_off_const*on_off[5],
    energy_sender_economy_mode2[5]+=turn_on_off_const*on_off[5],
    energy_sender[5]+=turn_on_off_const*on_off[5],
    energy_sender_economy_mode3[5]+=turn_on_off_const*on_off[5],on_off[5]=0=}
  ::s5_reset_clock_and_counter{=x5=0,y5=0,
    energy_sender[5]+=z5*idle_factor+z5*turn_on_off_const2/t_intervall;on_off[5]=1,
    energy_sender_economy_mode[5] += z5*idle_factor+z5*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode2[5] += z5*idle_factor+z5*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode3[5] += z5*idle_factor+z5*turn_on_off_const2/t_intervall=}
  ::s5_freeze_backoff{=sleep_mode1[5]+=TRANS_TIME_MIN-ASLOTTIME,
    energy_sender_economy_mode[5]+=turn_on_off_const,
    energy_sender_economy_mode3[5]+=turn_on_off_const=}
  ::s5_count_down_backoffvalue{=sleep_mode2[5]+=ASLOTTIME-10,
    energy_sender_economy_mode2[5]+=turn_on_off_const,
    energy_sender_economy_mode3[5]+=turn_on_off_const=}
  ::s5_enter_vulnerable_period{=energy_sender[5]+=y5*sensing_factor,
    energy_sender_economy_mode[5]+=(y5-sleep_mode1[5])*sensing_factor+
    sleep_mode1[5]*idle_factor,
    energy_sender_economy_mode2[5] += (y5-sleep_mode2[5])*
    sensing_factor+sleep_mode2[5]*idle_factor,
    energy_sender_economy_mode3[5] += (y5-sleep_mode1[5]-sleep_mode2[5])*
    sensing_factor+(sleep_mode1[5]+sleep_mode2[5])*idle_factor=}
  ::s5_start_send{= x5=0=}
  ::s5_send{= energy_sender[5]+=transmitting_factor*x5,
    energy_sender_economy_mode[5]+=transmitting_factor*x5,
    energy_sender_economy_mode2[5]+=transmitting_factor*x5,
    energy_sender_economy_mode3[5]+=transmitting_factor*x5, x5=0=}
  ::s5_detects_channel_busy_after_sending_and_will_resend{=z5=0,
    time_sender_channel_used_not_successful[5]+=sending_duration[5]=}
  ::s5_ack_received_correct{=energy_sender[5]+=sensing_factor*x5,
    energy_sender_economy_mode[5]+=sensing_factor*x5;z5=0,
    time_sender_send[5]+=sending_duration[5],
    energy_sender_economy_mode2[5]+=sensing_factor*x5,
    energy_sender_economy_mode3[5]+=sensing_factor*x5=}
  ::s5_will_resend_message{=energy_sender[5]+=sensing_factor*x5,
    energy_sender_economy_mode[5]+=sensing_factor*x5;z5=0,
    energy_sender_economy_mode2[5]+=sensing_factor*x5,
    energy_sender_economy_mode3[5]+=sensing_factor*x5,
    time_sender_channel_used_not_successful[5]+=sending_duration[5]=}
  ::s5_will_skip_message{=energy_sender[5]+=sensing_factor*x5,
    energy_sender_economy_mode[5]+=sensing_factor*x5,
    z5=0;energy_sender_economy_mode2[5]+=sensing_factor*x5,
    energy_sender_economy_mode3[5]+=sensing_factor*x5,
    time_sender_channel_used_not_successful[5]+=sending_duration[5]=}

  ::s6_do_nothing{=energy_sender_economy_mode[6]+=turn_on_off_const*on_off[6],
    energy_sender_economy_mode2[6]+=turn_on_off_const*on_off[6],
    energy_sender[6]+=turn_on_off_const*on_off[6],
    energy_sender_economy_mode3[6]+=turn_on_off_const*on_off[6],on_off[6]=0=}
  ::s6_reset_clock_and_counter{=x6=0,y6=0,
    energy_sender[6]+=z6*idle_factor+z6*turn_on_off_const2/t_intervall;on_off[6]=1,
    energy_sender_economy_mode[6] += z6*idle_factor+z6*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode2[6] += z6*idle_factor+z6*turn_on_off_const2/t_intervall,
    energy_sender_economy_mode3[6] += z6*idle_factor+z6*turn_on_off_const2/t_intervall=}
  ::s6_freeze_backoff{=sleep_mode1[6]+=TRANS_TIME_MIN-ASLOTTIME,
    energy_sender_economy_mode[6]+=turn_on_off_const,
    energy_sender_economy_mode3[6]+=turn_on_off_const=}
  ::s6_count_down_backoffvalue{=sleep_mode2[6]+=ASLOTTIME-10,
    energy_sender_economy_mode2[6]+=turn_on_off_const,
    energy_sender_economy_mode3[6]+=turn_on_off_const=}
  ::s6_enter_vulnerable_period{=energy_sender[6]+=y6*sensing_factor,
    energy_sender_economy_mode[6]+=(y6-sleep_mode1[6])*sensing_factor+
    sleep_mode1[6]*idle_factor,
    energy_sender_economy_mode2[6] += (y6−sleep_mode2[6])*
    sensing_factor+sleep_mode2[6]*idle_factor,
    energy_sender_economy_mode3[6] += (y6−sleep_mode1[6]−sleep_mode2[6])*
    sensing_factor+(sleep_mode1[6]+sleep_mode2[6])*idle_factor=}
  ::s6_start_send{= x6=0 =}
  ::s6_send{= energy_sender[6]+=transmitting_factor*x6,
    energy_sender_economy_mode[6]+=transmitting_factor*x6,
    energy_sender_economy_mode2[6]+=transmitting_factor*x6,
    energy_sender_economy_mode3[6]+=transmitting_factor*x6,x6=0=}
  ::s6_detects_channel_busy_after_sending_and_will_resend{=z6=0,
    time_sender_channel_used_not_successful[6]+=sending_duration[6]
  ::s6_ack_received_correct{=energy_sender[6]+=sensing_factor*x6,
    energy_sender_economy_mode[6]+=sensing_factor*x6,z6=0,
    time_sender_send[6]+=sending_duration[6],
    energy_sender_economy_mode2[6]+=sensing_factor*x6,
    energy_sender_economy_mode3[6]+=sensing_factor*x6=}
  ::s6_will_resend_message{=energy_sender[6]+=sensing_factor*x6,
    energy_sender_economy_mode[6]+=sensing_factor*x6,
    z6=0;energy_sender_economy_mode2[6]+=sensing_factor*x6,
    energy_sender_economy_mode3[6]+=sensing_factor*x6,
    time_sender_channel_used_not_successful[6]+=sending_duration[6]=}
  ::s6_will_skip_message{=energy_sender[6]+=sensing_factor*x6,
    energy_sender_economy_mode[6]+=sensing_factor*x6,
    z6=0;energy_sender_economy_mode2[6]+=sensing_factor*x6,
    energy_sender_economy_mode3[6]+=sensing_factor*x6,
    time_sender_channel_used_not_successful[6]+=sending_duration[6]=}


  ::use_of_channel_detected{= time_channel_not_used+=timer1,timer1=0=}
  ::channel_corrupted{= time_channel_not_used+=timer1=}
  ::channel_ready_after_corrupted{=timer1=0=}
  ::collision_detected{=coll=1,timer_coll=0=}
  ::channel_is_free{=time_in_coll+=coll*timer_coll,coll=0,
    time_channel_used+=timer1,timer1=0=}
  ::when (global_time==simulation_duration)time_channel_used+=(1.0−c.free)*timer1;
          time_channel_not_used+=c.free*timer1;timer1=0;time_in_coll+=coll*timer_coll;
          energy_sender[1]+=is_sending[1]*transmitting_factor*x +
          (1−is_sending[1])*sensing_factor*(vuln_reached[1]*x+
          ((1−vuln_reached[1])*msg_to_send[1]*y))+(1−msg_to_send[1])*z*idle_factor;
          energy_sender_economy_mode[1]+=is_sending[1]*transmitting_factor*x +
          (1−is_sending[1])*sensing_factor*(vuln_reached[1]*x+
          ((1−vuln_reached[1])*msg_to_send[1]*(y−sleep_mode1[1]))) +
           (1−msg_to_send[1])*z*idle_factor;
          energy_sender_economy_mode2[1]+=is_sending[1]*transmitting_factor*x +
          (1−is_sending[1])*sensing_factor*(vuln_reached[1]*x+
          ((1−vuln_reached[1])*msg_to_send[1]*(y−sleep_mode2[1]))) +
          (1−msg_to_send[1])*z*idle_factor;
          energy_sender_economy_mode3[1]+=is_sending[1]*transmitting_factor*x +
          (1−is_sending[1])*sensing_factor*(vuln_reached[1]*x+
          ((1−vuln_reached[1])*msg_to_send[1]*(y−sleep_mode1[1]−sleep_mode2[1])))
          + (1−msg_to_send[1])*z*idle_factor;

          energy_sender[2]+=is_sending[2]*transmitting_factor*x2 +
          (1−is_sending[2])*sensing_factor*(vuln_reached[2]*x2+
          ((1−vuln_reached[2])*msg_to_send[2]*y2))+(1−msg_to_send[2])*z2*idle_factor;
          energy_sender_economy_mode[2]+=is_sending[2]*transmitting_factor*x2 +
          (1−is_sending[2])*sensing_factor*(vuln_reached[2]*x2+
          ((1−vuln_reached[2])*msg_to_send[2]*(y2−sleep_mode1[2])))
          + (1−msg_to_send[2])*z2*idle_factor;
          energy_sender_economy_mode2[2]+=is_sending[2]*transmitting_factor*x2 +
          (1−is_sending[2])*sensing_factor*(vuln_reached[2]*x2+
          ((1−vuln_reached[2])*msg_to_send[2]*(y2−sleep_mode2[2])))
          + (1−msg_to_send[2])*z2*idle_factor;
          energy_sender_economy_mode3[2]+=is_sending[2]*transmitting_factor*x2 +
          (1−is_sending[2])*sensing_factor*(vuln_reached[2]*x2+
          ((1−vuln_reached[2])*msg_to_send[2]*(y2−sleep_mode1[2]−sleep_mode2[2])))
          + (1−msg_to_send[2])*z2*idle_factor;

          energy_sender[3]+=is_sending[3]*transmitting_factor*x3 +
          (1−is_sending[3])*sensing_factor*(vuln_reached[3]*x3+
          ((1−vuln_reached[3])*msg_to_send[3]*y3))+(1−msg_to_send[3])*z3*idle_factor;
          energy_sender_economy_mode[3]+=is_sending[3]*transmitting_factor*x3 +
          (1−is_sending[3])*sensing_factor*(vuln_reached[3]*x3+
          ((1−vuln_reached[3])*msg_to_send[3]*(y3−sleep_mode1[3])))
          + (1−msg_to_send[3])*z3*idle_factor;
          energy_sender_economy_mode2[3]+=is_sending[3]*transmitting_factor*x3 +
          (1−is_sending[3])*sensing_factor*(vuln_reached[3]*x3+
          ((1−vuln_reached[3])*msg_to_send[3]*(y3−sleep_mode2[3])))
          + (1−msg_to_send[3])*z3*idle_factor;
          energy_sender_economy_mode3[3]+=is_sending[3]*transmitting_factor*x3 +
          (1−is_sending[3])*sensing_factor*(vuln_reached[3]*x3+
          ((1−vuln_reached[3])*msg_to_send[3]*(y3−sleep_mode1[3]−sleep_mode2[3])))
          + (1−msg_to_send[3])*z3*idle_factor;
```

```
            energy_sender[4]+=is_sending[4]*transmitting_factor*x4 +
            (1-is_sending[4])*sensing_factor*(vuln_reached[4]*x4+
            ((1-vuln_reached[4])*msg_to_send[4]*y4))+(1-msg_to_send[4])*z4*idle_factor;
            energy_sender_economy_mode[4]+=is_sending[4]*transmitting_factor*x4 +
            (1-is_sending[4])*sensing_factor*(vuln_reached[4]*x4+
            ((1-vuln_reached[4])*msg_to_send[4]*(y4-sleep_mode1[4]))) +
            (1-msg_to_send[4])*z4*idle_factor;
            energy_sender_economy_mode2[4]+=is_sending[4]*transmitting_factor*x4 +
            (1-is_sending[4])*sensing_factor*(vuln_reached[4]*x4+
            ((1-vuln_reached[4])*msg_to_send[4]*(y4-sleep_mode2[4])))
            + (1-msg_to_send[4])*z4*idle_factor;
            energy_sender_economy_mode3[4]+=is_sending[4]*transmitting_factor*x4 +
            (1-is_sending[4])*sensing_factor*(vuln_reached[4]*x4+
            ((1-vuln_reached[4])*msg_to_send[4]*(y4-sleep_mode1[4]-sleep_mode2[4])))
            + (1-msg_to_send[4])*z4*idle_factor;

            energy_sender[5]+=is_sending[5]*transmitting_factor*x5 +
            (1-is_sending[5])*sensing_factor*(vuln_reached[5]*x5+
            ((1-vuln_reached[5])*msg_to_send[5]*y5))+(1-msg_to_send[5])*z5*idle_factor;
            energy_sender_economy_mode[5]+=is_sending[5]*transmitting_factor*x5 +
            (1-is_sending[5])*sensing_factor*(vuln_reached[5]*x5+
            ((1-vuln_reached[5])*msg_to_send[5]*(y5-sleep_mode1[5])))
            + (1-msg_to_send[5])*z5*idle_factor;
            energy_sender_economy_mode2[5]+=is_sending[5]*transmitting_factor*x5 +
            (1-is_sending[5])*sensing_factor*(vuln_reached[5]*x5+
            ((1-vuln_reached[5])*msg_to_send[5]*(y5-sleep_mode2[5])))
            + (1-msg_to_send[5])*z5*idle_factor;
            energy_sender_economy_mode3[5]+=is_sending[5]*transmitting_factor*x5 +
            (1-is_sending[5])*sensing_factor*(vuln_reached[5]*x5+
            ((1-vuln_reached[5])*msg_to_send[5]*(y5-sleep_mode1[5]-sleep_mode2[5])))
            + (1-msg_to_send[5])*z5*idle_factor;

            energy_sender[6]+=is_sending[6]*transmitting_factor*x6 +
            (1-is_sending[6])*sensing_factor*(vuln_reached[6]*x6+
            ((1-vuln_reached[6])*msg_to_send[6]*y6))+(1-msg_to_send[6])*z6*idle_factor;
            energy_sender_economy_mode[6]+=is_sending[6]*transmitting_factor*x6 +
            (1-is_sending[6])*sensing_factor*(vuln_reached[6]*x6+
            ((1-vuln_reached[6])*msg_to_send[6]*(y6-sleep_mode1[6])))
            + (1-msg_to_send[6])*z6*idle_factor;
            energy_sender_economy_mode2[6]+=is_sending[6]*transmitting_factor*x6 +
            (1-is_sending[6])*sensing_factor*(vuln_reached[6]*x6+
            ((1-vuln_reached[6])*msg_to_send[6]*(y6-sleep_mode2[6])))
            + (1-msg_to_send[6])*z6*idle_factor;
            energy_sender_economy_mode3[6]+=is_sending[6]*transmitting_factor*x6 +
            (1-is_sending[6])*sensing_factor*(vuln_reached[6]*x6+
            ((1-vuln_reached[6])*msg_to_send[6]*(y6-sleep_mode1[6]-sleep_mode2[6])))
            + (1-msg_to_send[6])*z6*idle_factor;
            when (timer1==1) tau
    }

}

par{
    :: relabel{
       enable_backoff, start_wait_for_ack, reset_clock_and_counter,
       choose_backoffvalue, continuebackoff, count_down_backoffvalue,
       freeze_backoff, start_send, send, exit_backoff,
       deals_with_backoff, reset_sense_in_backoff, do_nothing,
       will_resend_message, ack_received_correct, waited_DIFS_backoff,
       enter_vulnerable_period, will_skip_message,
       detects_channel_busy_after_sending_and_will_resend }
       by
       {
       s1_enable_backoff, s1_start_wait_for_ack, s1_reset_clock_and_counter,
       s1_choose_backoffvalue, s1_continue_backoff, s1_count_down_backoffvalue,
       s1_freeze_backoff, s1_start_send, s1_send, s1_exit_backoff,
       s1_deals_with_backoff, s1_reset_sense_in_backoff, s1_do_nothing,
       s1_will_resend_message, s1_ack_received_correct, s1_waited_DIFS_backoff,
       s1_enter_vulnerable_period, s1_will_skip_message,
       s1_detects_channel_busy_after_sending_and_will_resend}
       sender(1,load1,active_senders)

    :: relabel{
       enable_backoff, start_wait_for_ack, reset_clock_and_counter,
       choose_backoffvalue, continuebackoff, count_down_backoffvalue,
       freeze_backoff, start_send, send, exit_backoff,
       deals_with_backoff, reset_sense_in_backoff, do_nothing,
       will_resend_message, ack_received_correct, waited_DIFS_backoff,
       enter_vulnerable_period, will_skip_message,
       detects_channel_busy_after_sending_and_will_resend }
       by
       {
```

```
    s2_enable_backoff ,s2_start_wait_for_ack ,s2_reset_clock_and_counter ,
    s2_choose_backoffvalue ,s2_continue_backoff ,s2_count_down_backoffvalue ,
    s2_freeze_backoff ,s2_start_send ,s2_send ,s2_exit_backoff ,
    s2_deals_with_backoff ,s2_reset_sense_in_backoff ,s2_do_nothing ,
    s2_will_resend_message ,s2_ack_received_correct ,s2_waited_DIFS_backoff ,
    s2_enter_vulnerable_period ,s2_will_skip_message ,
    s2_detects_channel_busy_after_sending_and_will_resend}
    sender(2,load1 ,active_senders)

:: relabel{
    enable_backoff ,start_wait_for_ack ,reset_clock_and_counter ,
    choose_backoffvalue ,continuebackoff ,count_down_backoffvalue ,
    freeze_backoff ,start_send ,send ,exit_backoff ,
    deals_with_backoff ,reset_sense_in_backoff ,do_nothing ,
    will_resend_message ,ack_received_correct ,waited_DIFS_backoff ,
    enter_vulnerable_period ,will_skip_message ,
    detects_channel_busy_after_sending_and_will_resend }
    by
    {
    s3_enable_backoff ,s3_start_wait_for_ack ,s3_reset_clock_and_counter ,
    s3_choose_backoffvalue ,s3_continue_backoff ,s3_count_down_backoffvalue ,
    s3_freeze_backoff ,s3_start_send ,s3_send ,s3_exit_backoff ,
    s3_deals_with_backoff ,s3_reset_sense_in_backoff ,s3_do_nothing ,
    s3_will_resend_message ,s3_ack_received_correct ,s3_waited_DIFS_backoff ,
    s3_enter_vulnerable_period ,s3_will_skip_message ,
    s3_detects_channel_busy_after_sending_and_will_resend}
    sender(3,load1 ,active_senders)

:: relabel{
    enable_backoff ,start_wait_for_ack ,reset_clock_and_counter ,
    choose_backoffvalue ,continuebackoff ,count_down_backoffvalue ,
    freeze_backoff ,start_send ,send ,exit_backoff ,
    deals_with_backoff ,reset_sense_in_backoff ,do_nothing ,
    will_resend_message ,ack_received_correct ,waited_DIFS_backoff ,
    enter_vulnerable_period ,will_skip_message ,
    detects_channel_busy_after_sending_and_will_resend }
    by
    {
    s4_enable_backoff ,s4_start_wait_for_ack ,s4_reset_clock_and_counter ,
    s4_choose_backoffvalue ,s4_continue_backoff ,s4_count_down_backoffvalue ,
    s4_freeze_backoff ,s4_start_send ,s4_send ,s4_exit_backoff ,
    s4_deals_with_backoff ,s4_reset_sense_in_backoff ,s4_do_nothing ,
    s4_will_resend_message ,s4_ack_received_correct ,s4_waited_DIFS_backoff ,
    s4_enter_vulnerable_period ,s4_will_skip_message ,
    s4_detects_channel_busy_after_sending_and_will_resend}
    sender(4,load1 ,active_senders)

:: relabel{
    enable_backoff ,start_wait_for_ack ,reset_clock_and_counter ,
    choose_backoffvalue ,continuebackoff ,count_down_backoffvalue ,
    freeze_backoff ,start_send ,send ,exit_backoff ,
    deals_with_backoff ,reset_sense_in_backoff ,do_nothing ,
    will_resend_message ,ack_received_correct ,waited_DIFS_backoff ,
    enter_vulnerable_period ,will_skip_message ,
    detects_channel_busy_after_sending_and_will_resend }
    by
    {
    s5_enable_backoff ,s5_start_wait_for_ack ,s5_reset_clock_and_counter ,
    s5_choose_backoffvalue ,s5_continue_backoff ,s5_count_down_backoffvalue ,
    s5_freeze_backoff ,s5_start_send ,s5_send ,s5_exit_backoff ,
    s5_deals_with_backoff ,s5_reset_sense_in_backoff ,s5_do_nothing ,
    s5_will_resend_message ,s5_ack_received_correct ,s5_waited_DIFS_backoff ,
    s5_enter_vulnerable_period ,s5_will_skip_message ,
    s5_detects_channel_busy_after_sending_and_will_resend}
    sender(5,load1 ,active_senders)

:: relabel{
    enable_backoff ,start_wait_for_ack ,reset_clock_and_counter ,
    choose_backoffvalue ,continuebackoff ,count_down_backoffvalue ,
    freeze_backoff ,start_send ,send ,exit_backoff ,
    deals_with_backoff ,reset_sense_in_backoff ,do_nothing ,
    will_resend_message ,ack_received_correct ,waited_DIFS_backoff ,
    enter_vulnerable_period ,will_skip_message ,
    detects_channel_busy_after_sending_and_will_resend }
    by
    {
    s6_enable_backoff ,s6_start_wait_for_ack ,s6_reset_clock_and_counter ,
    s6_choose_backoffvalue ,s6_continue_backoff ,s6_count_down_backoffvalue ,
    s6_freeze_backoff ,s6_start_send ,s6_send ,s6_exit_backoff ,
    s6_deals_with_backoff ,s6_reset_sense_in_backoff ,s6_do_nothing ,
    s6_will_resend_message ,s6_ack_received_correct ,s6_waited_DIFS_backoff ,
    s6_enter_vulnerable_period ,s6_will_skip_message ,
    s6_detects_channel_busy_after_sending_and_will_resend}
```

```
        sender(6,load1,active_senders)

    :: relabel{receive,start_send_ack,send_ack} by
        {r1_receive,r1_start_send_ack,r1_send_ack} receiver(-1)

    :: relabel{receive,start_send_ack,send_ack} by
        {r2_receive,r2_start_send_ack,r2_send_ack} receiver(-2)

    :: relabel{receive,start_send_ack,send_ack} by
        {r3_receive,r3_start_send_ack,r3_send_ack} receiver(-3)

    :: relabel{receive,start_send_ack,send_ack} by
        {r4_receive,r4_start_send_ack,r4_send_ack} receiver(-4)


    :: relabel{receive,start_send_ack,send_ack} by
        {r5_receive,r5_start_send_ack,r5_send_ack} receiver(-5)

    :: relabel{receive,start_send_ack,send_ack} by
        {r6_receive,r6_start_send_ack,r6_send_ack} receiver(-6)

    :: chan()
    :: monitor()
}
```