IoT Project – Hidden Terminal

Alberto Tavini 10578559 – Marco Petri 10569751

DESIGN CHOICES

We developed our project using TinyOS and Tossim simulator. We used the meyer-heavy file provided for the fourth challenge as noise file for the project.

We utilized a struct defined in node.h to define messages to be exchanged between the nodes, it has four fields:

- **type**: is an integer to which constants RTS, CTS and CNT are associated (we use it to distinguish between different message types).
- **counter**: is the payload to be delivered, if any.
- **RTS_sender**: is the sender of the RTS message or the sender of the payload, if any.
- **CTS_authorized**: is the node allowed from the base station to transmit its payload, if any.

We ensure that the maximum number of messages a node can send to the base station is **MAX** (a constant). The insertion of a maximum number of messages to send from a base station is due to the necessity of obtaining comparable results.

The **nodeAppC.nc** file is used to define the components and interfaces, while the **nodeC.nc** file has the code itself.

TOPOLOGY AND EXPECTATIONS

To correctly address the problem of hidden nodes, we wrote our project having a network where the Base station (node 1) sees all the nodes, while nodes 2 and 3 do not see node 6 and vice versa. We kept the noise parameter at -60 dB for all the links.

Simulating the network, we would expect nodes 2 and 3 to have a higher PER than 4 and 5, and node 6 to have the highest PER. Nodes 4 and 5 can overhear each node; therefore, they will be able to avoid sending messages while others are willing to send their own (which is shown by an RTS message to the base station).

TWO MODES: RTS ONLY

To fully understand the difference it makes in implementing the RTS/CTS protocol, we decided to provide our code with two operational modes that can be easily switched by changing the initial value of the boolean global variable ***only_RTS_mode***.

The first implementation uses only the first type of pre-emptive messages. Due to the absence of CTS messages, this implementation is particularly vulnerable to the hidden terminal problem: only the ones that are in range of a given node will receive its RTS and will wait before attempting a request themselves.

When the timer fires for a node and the probability distribution's outcome given by ***allowedtotransmit()*** tells to attempt a transmission, we call ***sendRTS()***. It broadcasts an RTS to the nodes that are in range. After the completion of its ***AMSend()***, the event ***AMSend.SendDone()*** is automatically called and the payload with the node counter is sent through the ***sendpayload()*** method with only the base station as recipient. Received messages are processed through the event ***Receive.receive()*** that in this mode calls the ***receiveRTSonly()*** method. If the received is an RTS, the nodes or the base station will respectively go to sleep or ignore it. If the received message is a CNT one with actual payload the receiver must be the base, and in that case it proceeds to update the PER for that node. If a node which isn't the base receives a CNT message, nothing will be done.

TWO MODES: RTS/CTS

This implementation provides more robustness wrt to the hidden node problem. When timer fires the nodes still broadcast an RTS message on all the links but does not send the payload immediately after. As the ***AMSend.SencDone()*** has no executable code in this mode. In this case, the method ***Receive.receive()*** calls ***receiveRTSCTS()*** method, which manages and sends CNT messages.

Upon receival of RTS, the base station responds with ***sendCTS()*** method, while simple nodes go to sleep (for an amount of time equivalent to 4 timers). This behaviour is the standard also when a CTS message is received and the node is not the one authorized by the base station (***CTS_authorized*** field). Particularly, if the node is the one indicated in ***CTS_authorized,*** it will reply with the CNT message with actual payload and increase its counter.

RANDOM ACCESS TIME

To handle a probability driven method to send messages from terminals to the base station, we decided to use a Poisson distribution and to perform sampling. We decide the rate of the Poisson distribution specifying the number of messages that the terminal will send. Let us call $r\ [s^{-1}]$ the rate of the distribution and $T\ [s]$ the duration of the timer. After one cycle (1 timer period is passed), the variance of the distribution is $\lambda = rT$. Given that the following is the probability of obtaining k messages:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

The probability that the terminal will have sent at least one message in the first period is:

$$P(X \geq 1) = 1 - P(X = 0) = 1 - e^{-\lambda}$$

Since we selected $r = 20s^{-1}$ and $T = 50ms$, we have $P(X \geq 1) = 0.6321$. We perform sampling on this value: we generate a random number between 0 and 1, if it is lower than the probability of sending at least one packer (0.6321 after one cycle) we will try to send a packet and refresh the variable ***num_of_failures*** to 0, otherwise we do not send the packet and we increment by 1 ***num_of_failures***. If we do not have to send, at the next iteration (2 cycles passed) we will have $\lambda = 2rT$. Then we repeat the process of computing the probability (0.8647 this time) and sampling previously introduced again and again. The more "failures" we will have, the higher will be the probability of sending a message. Moreover, if we analyse the expression of the probability of sending at least one packet after n failures letting n to go to infinity:

$$\lim_{n \to \infty} 1 - e^{-nrT} = 1$$

OTHER METHODS

Apart from the aforementioned methods we utilize a banal ***initialize_arrays()*** to fill with zeroes the variables used by ***computePER()***: this method computes the Packet Error Rate as the percentage of CNT packets that the base station doesn't receive. In order to do that, we memorize for each node the counter received at previous transmission, and we compare it to the current one to see if any values were lost. The method also counts the number of CNT that globally cross the network and stores it in ***base_counter*** so that when it reaches **5*MAX** it the calls ***printfinalstats()*** (this is not deterministic as the last packets may get lost and consequently ***base_counter*** may not reach that threshold, however, it is just utility printing since at each iteration we print statistics obtained so far for the node that has sent a CNT).

CONCLUSIONS

By running some simulations with MAX equal to 400 we obtained that in both cases the results are consistent with what we expected when designing the topology, but in RTSONLY mode the average number of lost packets is very high:

```
PRINTING FINAL STATS
DEBUG (1): [0:0:29.011611903]: At the end for node 2 the PER is 20.0501 %
DEBUG (1): [0:0:29.011611903]: At the end for node 3 the PER is 17.5439 %
DEBUG (1): [0:0:29.011611903]: At the end for node 4 the PER is 11.5288 %
DEBUG (1): [0:0:29.011611903]: At the end for node 5 the PER is 12.2807 %
DEBUG (1): [0:0:29.011611903]: At the end for node 6 the PER is 30.0752 %
```

While in RTS/CTS mode the average performance is a way better, also for the hidden nodes:

```
PRINTING FINAL STATS
DEBUG (1): [0:1:19.457046438]: At the end for node 2 the PER is 7.5188 %
DEBUG (1): [0:1:19.457046438]: At the end for node 3 the PER is 6.5163 %
DEBUG (1): [0:1:19.457046438]: At the end for node 4 the PER is 4.2607 %
DEBUG (1): [0:1:19.457046438]: At the end for node 5 the PER is 3.0075 %
DEBUG (1): [0:1:19.457046438]: At the end for node 6 the PER is 8.5213 %
```

Consequently, we can conclude that, even though the RTS/CTS protocol is far from being perfect, it provides a simple method to vastly ameliorate the performance in case of hidden nodes.