IoT + WI Joint Project nr 2: The (Hidden) Terminal

A.Y. 2020/21

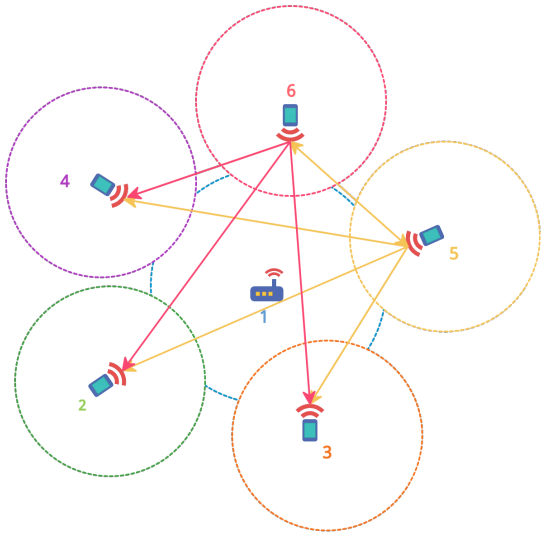
Stefano Maxenti – 10526141 Elisabetta Cainazzo - 10800059

Introduction

The hidden terminal problem is a transmission problem that arises when two or more stations who are out of range of each other transmit simultaneously to a common base station.
The goal of our project is to implement a basic version of the RTS/CTS mechanism to improve performance when there are hidden terminals in the network.
We used the *TinyOs* framework and the *TOSSIM* simulator. The RTS/CTS function can be enabled during compiling by setting a specific preprocessor flag in the source code. We create two different *makefiles* (*Makefile-NoRTS* and *Makefile-RTS*) in which we specify, among others, the lambda parameter and the RTS timeout value. A script used is employed to automate the compiling and the testing, called "*test_script.sh*". Logs are in the folder "*log*".

Topology

Motes are positioned in the following way:



Mote #2, #3 and #4 are hidden terminal, because they only see the base station. Mote #5 and #6, on the other hand, can see all the nodes of the networks.
The arrow direction indicates whether a mote can see another mote's transmission. Each of the five mote sees, of course, the base station (mote #1).
Topology is described in the "*topology.txt*" file using the TOSSIM approach: each link is unidirectional (the opposite direction must be specified, this leads to the possibility of analysing asymmetric channels) and labelled with the signal strength, which is kept constant at -60.0dBm.

Random generator

Packets must be sent according to a Poisson process. Among many definitions, a Poisson Process is a random process where interarrival times are exponential. It is a memoryless distribution, which means each new arrival has no recalling of the previous ones.
To generate random arrivals, we sample from a uniform distribution. In fact, it is possible to obtain exponential values with parameter lambda by applying the following method (inverse transformation):

$$U \sim \mathcal{U}[0,1]$$
$$X = -\frac{1}{\lambda}\ln(U) \sim Exp(\lambda)$$

where U is a random value taken from a uniform distribution between 0 and 1.

In our code, we generate random values using the *RandomGeneratorC* component, which triggers the *Read* interface of the mote. We divide the random value by 65535 ($2^{16} - 1$) so that the result is normalized between 0 and 1; then, we extract the exponential value by applying the above formula; finally, the result is multiplied by 1000 so that the obtained time is expressed in milliseconds.
Since service time is not instantaneous, calculating the next interarrival time just after a packet has been sent doesn't lead to a precise Poisson distribution. For this reason, we use an approximation function (which can be disabled at compile time) that subtracts, to the new interarrival value, the time the previous packet has spent in the system (found using *sim_time* variables). In the situation without RTS/CTS, then, the number of packets that are actually sent in the time interval is coherent with the average of a Poisson distribution in a time interval T ($E[X] = \lambda T$): for example, the first simulation leads to a total time of 98 seconds that, multiplied by the average number of packets sent in one second, gives $98s \cdot 100\ pkt/s = 9800\ pkt \approx 10000\ pkt$.
Motes are booted all together at time 0 but start sending after a random exponential time.

Some comments on parameters

In the *makefile*, we set the following parameters:

- **_RTS_CTS** = 0/1 (0 if RTS is enabled, 1 otherwise).
- **_LAMBDA** = 100 (average of 100 packets per second).
- **_RTS_TIMEOUT** = 0.06 (seconds, timeout after sending the RTS and waiting for the CTS)
- **_MAX_NUMBER_OF_PACKETS** = 10000 (maximum number of a packets a mote sends before shutdown).
- **_SIMULATION_FLAG** = 1 (allows the use of *sim_time* variables in the simulation).
- **_ACKs** = 0 (ACK flags are not used).

In the second test of the RTS/CTS implementation, *_RTS_TIMEOUT* is halved to 0.03 seconds.

Implementation without RTS/CTS

As a ground truth, we first simulate packet generation when RTS/CTS is disabled.
TinyOS, by default, employs CSMA/CA in order to avoid collisions. Hidden terminals, however, cause problems because CSMA/CA is not able to listen to transmissions made by those stations.
In this implementation, we use these timers:

- *PacketGeneratorTimer*: it follows the Poisson distribution and it is in charge of generating the new packet with the increased counter
- *PacketSendTimer*: it is used to schedule the packet to the sent

The flow of action is the following:

1) A mote generates the request of packet with a counter value and starts *PacketSendTimer* immediately
2) The sending routine is invoked, and the packet is transmitted
3) After that, a new interarrival time is created and *PacketGeneratorTimer* is called again after that interval

We ran two simulations:

Mote #	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Average	Std. Dev.	Packet Loss
2	5390	5328	5413	5445	5394	49	**46%**
3	5430	5393	5478	5466	5442	38	**46%**
4	5326	5274	5474	5411	5371	89	**46%**
5	6775	6775	6879	6886	6829	62	**32%**
6	6847	6699	6773	6758	6769	61	**32%**
Run time	01:38:38	01:37:59	01:37:58	01:38:00			

We can easily notice the big amount of packet that are lost due to collisions. Of course, not-hidden terminal performs better than the counterpart since they can listen to the channel and avoid sending when it is not idle.

Implementation with RTS/CTS

In this implementation we add another timer:

- *RTSTimer*: it is used to schedule the sending of the Request-To-Send, both instantaneously and after a specific timeout defined by a parameter

The flow of action is the following:

1) A mote generates the request of a packet with a counter value, and we start *RTSTimer* immediately
2) A mote sends a RTS in broadcast; it starts another *RTSTimer* after the timeout so that it can send the RTS again if it doesn't receive a CTS
3) - If it receives a CTS with its own destination field, we stop the *RTSTimer* and we start the sending routine using the *PacketSendTimer* as above
   - If it receives a CTS with a different destination field, we stop any possible transmission and we start the *RTSTimer* again
   - If we receive a RTS coming from another motes, we reset *RTSTimer* and we start a new RTS request.
4) The sending procedure, when activated, is the same as before

We ran two simulations:

Mote #	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Average	Std. Dev.	Packet Loss
2	9667	9673	9608	9675	9656	32	**3%**
3	9653	9626	9626	9664	9642	19	**4%**
4	9645	9641	9652	9647	9646	5	**4%**
5	9700	9714	9731	9742	9722	19	**3%**
6	9748	9714	9726	9704	9723	19	**3%**
Run time	19:50:15	19:53:23	19:50:57	19:48:30			

The first thing to notice is that it takes ten times more to send the same amount of packets. This is because each packet requires a CTS before being actually sent.
However, we have a very low packet loss rate, since we avoid most of the collisions.

We modify *RTS_TIMEOUT* to a lower value (0.03).

Mote #	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Average	Std. Dev.	Packet Loss
2	9235	9271	9249	9250	9251	15	**7%**
3	9275	9239	9240	9259	9253	17	**7%**
4	9272	9230	9258	9240	9250	19	**8%**
5	9571	9547	9597	9595	9578	24	**4%**
6	9630	9596	9598	9605	9607	16	**4%**
Run time	17:19:59	17:14:01	17:16:39	17:16:10			

We notice that a smaller timeout leads to higher packet loss but the overall simulation is about 10% faster.
Without RTS/CTS, setting the ACK flag for each packet doesn't lead to better results since there are more collisions due to the ACK themselves. In the RTS/CTS scenario, however, results are similar, because of the timeout period that is much longer than the average service time, so the ACK is sent without colliding with other packets.

Conclusions

As shown by our empirical results, RTS/CTS can be successfully employed to increase the efficiency of a transmission by solving the hidden terminal problem.
For small values of λ, the number of collisions decreases, whereas for high values, such as the one we use in the simulation to underline the problem, the number of collisions increases dramatically.
The main drawback of this approach is the more time needed for sending the same amount of packets (ten times more).
Sending an RTS for each (small) packet is quite a waste of time, but it definitely reduces packet loss to interesting values.
In real life, RTS/CTS is normally employed only when there is a significant amount of hidden terminals and usually by setting a threshold on the size of the packet, so that shorter ones (and less likely to collide) are sent directly without sending the RTS and disrupting others' transmissions.