IoT Optional Project - Hidden Terminal

Marco Somaschini 10561636

Prof. Matteo Cesana

Academic Year 2021/2022

Contents

1 Introduction **3**
1.1 Tools Used . . . 3
1.2 Assumptions . . . 3

2 Implementation **4**
2.1 Poisson Distribution . . . 4
2.2 Baseline Communication Protocol . . . 4
2.3 CSMA protocol . . . 4
2.4 RTS/CTS . . . 4
2.5 Logging . . . 4

3 Conclusions **5**
3.1 No RTS/CTS . . . 5
3.2 RTS/CTS (250ms of wait time) . . . 5
3.3 RTS/CTS (100ms of wait time) . . . 5

1 Introduction

The hidden terminal is one of the classical problems of wireless communication. A typical scenario is that of multiple different terminals trying to communicate with the same node, which acts as a base station for operations. The base station falls in the transmission range of all these terminals (and vice versa), but the same isn't always true between a terminal and the others. If two or more terminals can't communicate directly, then it's impossible for them to implement any multiple access communication protocol, like ALOHA or CSMA/CA, and collisions will likely occur as they try to simultaneously transmit to the station. This project aims at simulating this scenario and illustrating how the hidden terminal problem can be solved through the introduction of the RTS/CTS protocol.

1.1 Tools Used

The project was realized using TinyOS framework to code the network's behaviour and TOSSIM python library for simulating it. The starting point was the template application proposed during the course challenge on TOSSIM, which was modified accordingly. The whole project has been uploaded on Github. The git repo contains two different branches: master and rts-cts. The first branch hosts a version of the application that do not implement the RTS/CTS protocol, while the second branch hosts one that does. In this way it's possible to compare the network behaviour in both situations.

1.2 Assumptions

- The motes were assumed to produce packets according to a Poisson distribution with a specific lambda.
- The network was assumed to be composed by six motes: one base station and five terminals. Furthermore, three motes were arranged to create a subnet together with the base station, while the other two were isolated. As a consequence, these two isolated motes would result hidden to each other and to the other three.

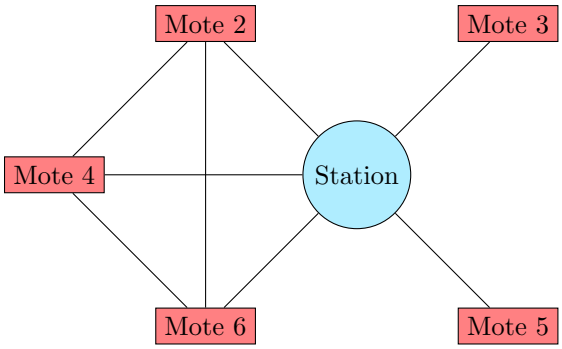

Figure 1: Network Topology

- The motes forming the subnet were assumed to implement a simple CSMA protocol to avoid collisions between them. This is needed because, without implementing any communication protocol, these motes would act just like they were hidden from one another.

2 Implementation

2.1 Poisson Distribution

The Poisson distribution was generated indirectly, by computing the inter-arrival time between each packet generation. The time interval between events of a Poisson process can be modelled as an exponential distribution $E(\lambda)$. To simulate such a distribution the *RandomC* component was used to extract a random 16-bit number and then mapping it to a value in $[0, 1]$, thus obtaining a sample from a uniform distribution. Then, leveraging the inverse cumulative distribution function for an exponential distribution $E(\lambda)$, the inter-arrival time was computed as: $F^{-1}(p, \lambda) = -\frac{\log 1-p}{\lambda}$. After booting, each mote generated a time interval according to its λ and set a timer of such length. The firing of the timer was interpreted as the arrival of a new packet. After the handling of the packet a new interval was computed, and a new timer set.

2.2 Baseline Communication Protocol

The baseline mote-station exchange worked as such: whenever a packet was ready, the mote would send it to the base station and wait for an ACK. This was done using the *AMSenderC* and *AMReceiverC* components. If a packet wasn't acked, the mote would simply retry sending it. The packets contained a sequence number and a field indicating how many times their transmission had been attempted. The base station, upon receiving a packet, would do some computation on it before replying with an ACK and logging its sequence number and total attempted transmissions, in order to compute the relative mote PER. The packet computation was simulated using the *FakeSensorC* component, that simply stalled for a short amount of time through a timer. This was needed as otherwise the entire exchange would be way too fast and unrealistic.

2.3 CSMA protocol

The CSMA protocol implemented by the three subnet motes consisted in sensing the channel before starting a transmission and backing-off if an undergoing communication was detected. The channel sensing was simulated using a global variable, which the subnet motes (only them) would check before attempting any transmissions. If the variable evaluated as FALSE, then the mote sensing it would set it to TRUE and start transmitting to the base station. Any subsequent mote would then detect that the channel was busy and back-off for a random amount of backoff periods randomly picked in $[0, 2^{BE} - 1]$. After receiving an ACK for the packet sent, the transmitting mote would set the sensing variable back to FALSE and thus free the channel.

2.4 RTS/CTS

The RTS/CTS version was built on top of the previous functionalities. In this version, before sending a packet, a mote sent an RTS message in broadcast and then stalled, waiting for a CTS from the base station. Upon receiving an RTS, any terminal mote was made to stall for a given amount of time. The base station, instead, would reply by sending a CTS message in broadcast and then wait for a packet. Again, any mote would stall for a given time upon receiving a CTS message, except for the terminal that requested it. Such mote would then proceed by sending the generated packet to the base station. The stalling was implement through the use of a timer and a Boolean variable, indicating if the mote was allowed to send.

2.5 Logging

The base station also kept track of the total elapsed time of the simulation, in order to stop accepting packets after a certain period. While ending the simulation, the base station printed the motes transmission stats, useful to understand the effectiveness of the protocols.

3 Conclusions

The simulations where set to account for five minutes of network activity and the following results were gathered.

3.1 No RTS/CTS

The first version, that didn't implement RTS/CTS, saw very high PERs, in the order of 10%. It can be noticed that the motes falling in the CSMA subnet have a slightly (around 1%) lower PER, due to collisions only happening with outsider motes. The average transmission rates approach the motes' λ as expected.

```
DEBUG (1): !END OF TRANSMISSIONS!
DEBUG (1):
DEBUG (1):
DEBUG (1): Base Station: Mote #2 AVG transmission rate is 4.810000 [msg/s]
DEBUG (1): Base Station: Mote #3 AVG transmission rate is 9.680000 [msg/s]
DEBUG (1): Base Station: Mote #4 AVG transmission rate is 4.540000 [msg/s]
DEBUG (1): Base Station: Mote #5 AVG transmission rate is 9.373333 [msg/s]
DEBUG (1): Base Station: Mote #6 AVG transmission rate is 4.583333 [msg/s]
DEBUG (1): Base Station: Mote #2 has a PER of 10.1%
DEBUG (1): Base Station: Mote #3 has a PER of 12.1%
DEBUG (1): Base Station: Mote #4 has a PER of 10.6%
DEBUG (1): Base Station: Mote #5 has a PER of 11.8%
DEBUG (1): Base Station: Mote #6 has a PER of 9.5%
```

Figure 2: No RTS/CTS motes transmissions stats

3.2 RTS/CTS (250ms of wait time)

The version implementing RTS/CTS, instead, showed much lower PERs, around 3-5%. Indeed, the introduction of the protocol managed to avoid a significant number of collisions. It must be said that the average average transmission rates of the motes decreased significantly, due to the overhead and extra synchronization effort required.

```
DEBUG (1): !END OF TRANSMISSIONS!
DEBUG (1):
DEBUG (1):
DEBUG (1): Base Station: Mote #2 AVG transmission rate is 0.540000 [msg/s]
DEBUG (1): Base Station: Mote #3 AVG transmission rate is 2.810000 [msg/s]
DEBUG (1): Base Station: Mote #4 AVG transmission rate is 0.450000 [msg/s]
DEBUG (1): Base Station: Mote #5 AVG transmission rate is 2.810000 [msg/s]
DEBUG (1): Base Station: Mote #6 AVG transmission rate is 0.516667 [msg/s]
DEBUG (1): Base Station: Mote #2 has a PER of 3.6%
DEBUG (1): Base Station: Mote #3 has a PER of 4.4%
DEBUG (1): Base Station: Mote #4 has a PER of 4.9%
DEBUG (1): Base Station: Mote #5 has a PER of 4.0%
DEBUG (1): Base Station: Mote #6 has a PER of 2.5%
```

Figure 3: RTS/CTS motes transmissions stats

3.3 RTS/CTS (100ms of wait time)

Running a simulation using the same version as before but with a significantly shorter wait time, which is the fixed amount of time nodes stops after receiving an RTS/CTS, it can be seen that the PERs increase by a couple percentage points but the average transmission rates increase as well. In conclusion, the wait time seems to be a parameter that controls the PER/throughput trade-off.

```
DEBUG (1): !END OF TRANSMISSIONS!
DEBUG (1):
DEBUG (1):
DEBUG (1): Base Station: Mote #2 AVG transmission rate is 1.386667 [msg/s]
DEBUG (1): Base Station: Mote #3 AVG transmission rate is 3.426667 [msg/s]
DEBUG (1): Base Station: Mote #4 AVG transmission rate is 1.100000 [msg/s]
DEBUG (1): Base Station: Mote #5 AVG transmission rate is 3.056667 [msg/s]
DEBUG (1): Base Station: Mote #6 AVG transmission rate is 1.190000 [msg/s]
DEBUG (1): Base Station: Mote #2 has a PER of 7.1%
DEBUG (1): Base Station: Mote #3 has a PER of 6.6%
DEBUG (1): Base Station: Mote #4 has a PER of 7.8%
DEBUG (1): Base Station: Mote #5 has a PER of 7.4%
DEBUG (1): Base Station: Mote #6 has a PER of 5.6%
```

Figure 4: RTS/CTS motes transmissions stats with shorter wait time