Hidden Terminal

Approaches to targets

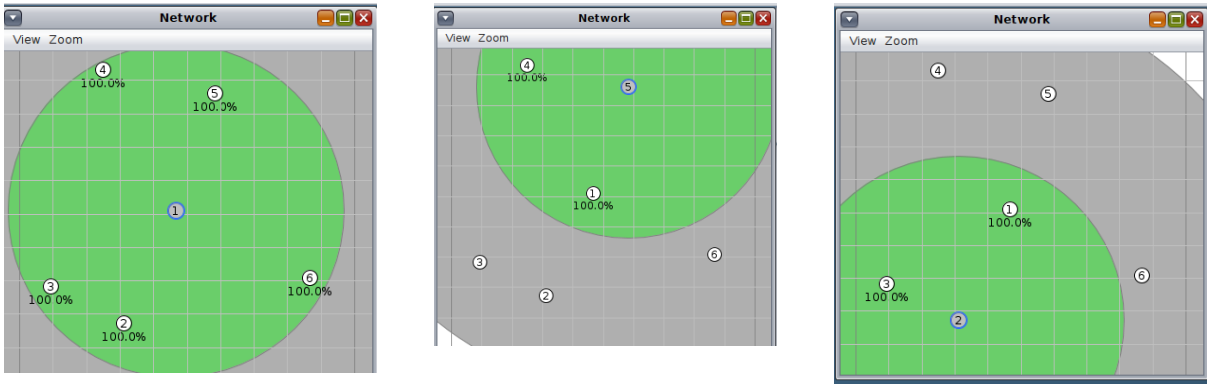
Network Layout

We have used *COOJA* to simulate a network composed of 6 motes divided as follow:

- One mote with (ID: 1) which acts as our base station;
- 5 motes (with IDs from 2 to 6) that send data to the base station.

Among the 5 motes we have considered (screenshots below):

- [4, 5] hidden to [2, 3], and vice-versa;
- 6 is hidden to all.



Network Layout

Some Important Constants

- `#define X` → it is the constant defining the time of the wait phase. It is also used to defining the time slots for drawing from Poisson distribution and to schedule the sending of the packets.
- `#define LAMBDA 3` → this is the expected value of the Poisson distribution 3 packets/X_milliseconds
- `#define NODES 100` → max number of nodes allowed other than the base station
- `uint8_t to_send;` → variable to keep tracks of the packets still to send every time we draw from the Poisson distribution

Implementation of baseline transmission protocol

There are two main functions that are in charge of managing the transmission:

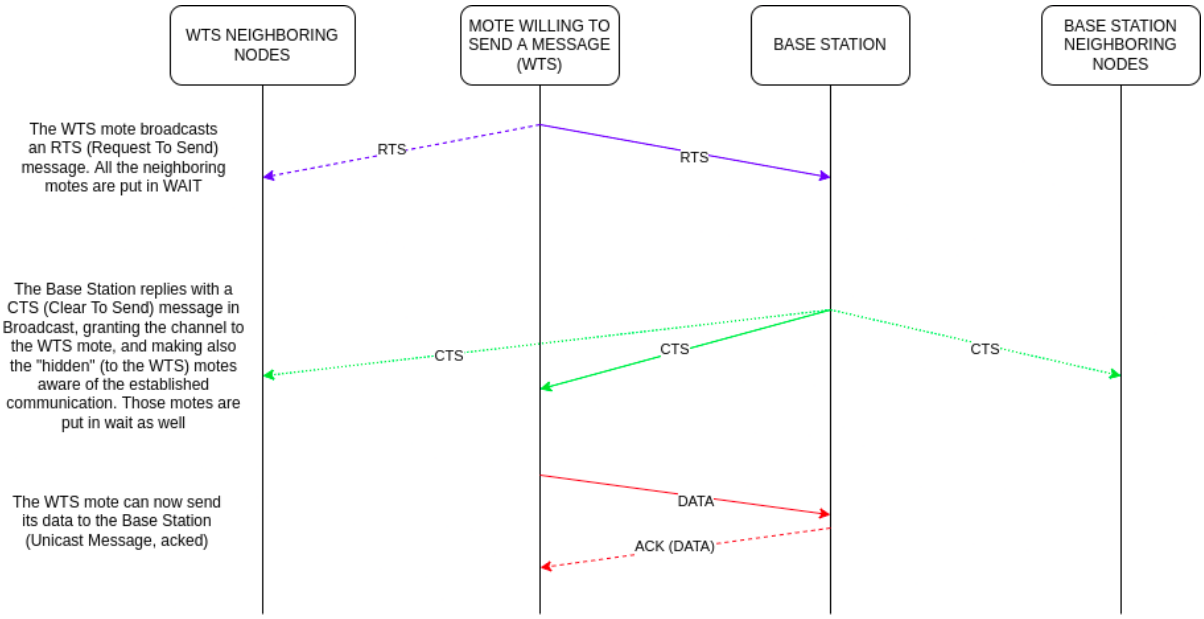
`void sendingProcedure();`

Given a time slots of X milliseconds, this function controls how the mote sends its sequential data to the base station. Indeed, using the `uint8_t samplePoisson();` which simulates the Poisson distribution.

Two design choices not explicitly requested by the assignment:

1. the Poisson distribution is sampled using Δ_t = X
2. each motes tries to send a packet every $X / sampledValue$ using the ***timerSend.*** When the latter fires, the RTS/CTS procedure restarts.

RTS/CTS mechanism implementation



Some important assumptions about the implementation:

- **RTS packets** are destined to the base station, which, if it's available (RTS mode), accepts the first incoming request (that became the current_sender).
- **CTS packets** are sent from the base station to current_sender (race winner described in the point above).
- **DATA packets** are sent after that the current_sender has received the CTS. The mote pass in DATA mode.
  In addition, the DATA packets are the only both sent in ***unicast*** and ***ack-ed***.

To manage all possible situation we have identified **4 possible status** of a mote:

- **RTS phase:** all the motes that want to transmit a packet must be in this status, in which they sent only RTS packet destined to the base station. Note that for the base station be in this state meaning to be free and available to receive a new incoming transmission request.
- **CTS phase**: Only the base station can enter in this phase, this happens after that it has accepted an RTS request and it has to send a CTS message to the mote the communication has been granted to. Notice that also the CTS message can be lost, for this reason a **timeoutCTS** timer has been implemented: when it fires (after a time of 3*X), the Base station can assume the packet as lost and it can return to the RTS phase in order to accept new incoming RTS .
- **DATA phase:** Only motes that have win the RTS race can enter in this phase, during which is possible to transmit the data to the station.
- **WAIT phase:** During this phase motes are waiting X milliseconds before retrying to send a new packet again. They can enter in this phase whenever are received RTS from other motes or a CTS destined to someone else. More in detail, receiving this type of messages the motes calls **timerWait.startOneShot(X)**

Nodes’ Packet Error Rates

Computation of those statistics are up to the base station. We opted for an incremental approach whenever a new packet arrive, updating scores of each node. Information is stored in two different arrays, total and success. They keep track,respectively, of the total incoming packets and of those resulting from a successful communication.

In particular:

- RTS received implies +1 on both total and successful
- DATA received implies +2 on both total and successful (CTS + DATA correctly delivered)
- CTS sent but DATA not received means +1 on successful and +2 on total (note that we have considered, on average, that just one of the two is lost)

The metric is computed as:

PER = (total - successful_message) / tot * 100