Comparing Neighbor Discovery Protocols

Filippo Tessaro, Mat. number 203346
filippo.tessaro@studenti.unitn.it

Abstract—**In this project two different primitives for continuous neighbor discovery will be developed: Burst and Scatter. Then an extensive evaluation will be done to compare performances and trade-offs of the implemented protocols.**

I. Introduction

Due to the possible mobility of IoT devices, it is common and critical for each IoT device to keep discovering who is in its neighborhood. Since the limited battery capacity of IoT devices, a challenging problem is to design an ad-hoc neighbor discovery protocol (NDP) that can achieve both low duty cycle and low discovery latency.

In this report two different neighbor discovery primitives will be developed and compared in terms of performance: neighbor discovery rate and radio duty cycle. The Neighbour Discovery (ND) primitives will need to operate with a level to reduce the radio duty cycle and also to maximise the discovery of nodes in the network. This is a challenging trade-off which is an active area of research.

The code of this project will be written in C and deployed with Contiki-OS. Contiki is an open-source operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things devices. It is a C based language and it is designed to run on types of hardware devices that are severely constrained in memory, power, processing power, and communication bandwidth. A typical Contiki system has memory on the order of kilobytes, a power budget on the order of milliwatts, processing speed measured in megaHertz, and communication bandwidth on the order of hundreds of kilobits/second.

The various tests and evaluations will be obtained through simulations with Cooja Simulator. The latter provides an excellent tool for the execution of primitives and reporting in *.log* files of the runtime node information. The devices used in the simulations are the TMote Sky. The Tmote Sky platform is a wireless sensor board from Moteiv. It is an MSP430-based board with an 802.15.4-compatible CC2420 radio chip, a 1-megabyte external serial flash memory, and two light sensors [1].

II. FUNDAMENTALS

In this section some theory will be described for the realisation of the project. Figure 1 and 2 show the structure of the ND primitives to be implemented. Both follow a sequence that repeats periodically which is called epoch and has duration D.

A. SCATTER MODE

Burst (Figure 1) starts the epoch by transmitting a beacon repeatedly for an entire transmission (*TX*) window of duration *T*. When the *TX* window finishes, Burst turns on the radio and listens for beacons. Shortly after, the radio is switched off until the next active Reception *RX* slot starts. A critical issue of this protocol is to decide in a proper way the *RX* slot duration and the amount of time the radio is on during an *RX* slot. Also anther critical factor is the number of *RX* windows. *RX* slots are repeated until the epoch finishes and a new *TX* window starts with the new epoch.

B. BURST MODE

Scatter (Figure 2) follows a different approach. In fact, it starts with a unique long RX window of duration T. Every beacon received during this window allows the listening node to discover new neighbors. When the RX window finishes at time T, Scatter stops listening, sends a single beacon for ND, and switches off the radio directly after to save energy. Scatter then periodically sends a beacon before turning off the radio until the epoch finishes, enabling other nodes to receive its beacons within every epoch. In this mode the number of transmission over the epoch is critical to discover new neighbors; also the duration of T is critical.

III. Implementation

This section will provide a precise description of the code and the implementation of the two primitives. The two primitives share the same callbacks but not the same protothread. For each protothread there are their respective primitives. The simulation over epochs takes place through an integer that takes into account the current epoch and a maximum number of epochs. In this way, executed all the tasks of a primitive, the starting thread is recalled to restart the execution.

A. Burst

The first primitive that will be described is the **burst** one. This primitive is implemented by starting a Process Thread called *burst_proc*. A process thread contains the code of the process and it is a single protothread that is invoked from the process scheduler. In this case, the scheduler of the process calls it at the beginning when *nd_start* callback is called. Then, the process thread will start his job by doing some different tasks. At the beginning of it, we define the *rtimer_clock_t next* variable as the sum between the *rtimer clock now* and a fraction of second which depends on the number of tasks. A task could be a TX window or an entire RX window. Once defined the variable, next we enter into a loop where every time we check

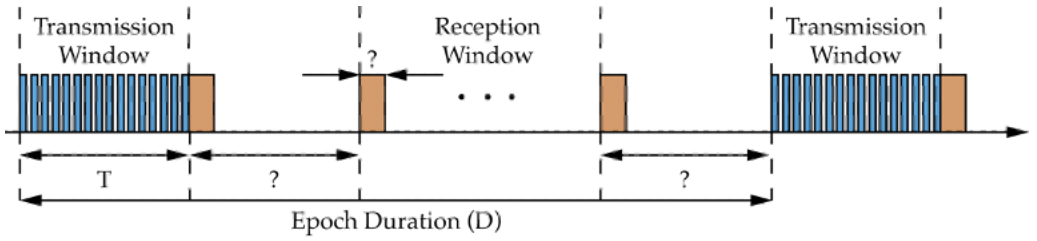

Figure 1: Diagram of Burst Primitive.

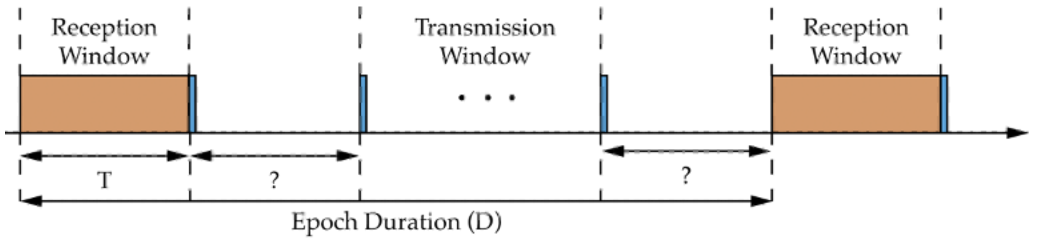

Figure 2: Diagram of Scatter Primitive.

if the timer of the next task is elapsed or not. If not, we send a beacon with the id of the sender as payload (which is a *uint_8*); after it, the node attends a fraction of time before restarting and checking the sending cycle. This phase is important to avoid sending too many beacons to congestion and saturate the channel, leading to collisions with other nodes. At the end of the TX window, we turn the radio on and we schedule the next turn off with an *rtimer* callback called *callback_turn_off*. The last is called after a time equal to the *next_off* variable (of value $RTIMER\_SECOND/100$) to distance the sending of each beacon. Passed the next off, the *turn_off* callback is called. This callback takes care of turning off the radio, increases the task variable (number of windows in the epoch), and decides whether to schedule the next *NETSTACK_RADIO.on()* or finish the epoch. The first action takes place if the *curr_task* variable is less than the number of tasks, while the second takes place in the opposite condition. Both are activated after a time equal to $(RTIMER\_SECOND/num\_task) - next\_off$ which is the remainder.

Now the two callbacks will be described:

- *turn_on_radio_callback* : takes care of reactivating the radio and scheduling the next turn off through the rtimer scheduler.
- *end_epoch_callback*: deals with the termination phase of the time. in this case, the neighbor discovered by the *nd_recv* inside the discovered_neighbour vector are counted by iterating and checking all the variables set to true. Then the *app_cb.nd_epoch_end(epoch, num_nbr)* is called. Then the variable *curr_task* is set to zero and if there are still periods available the initial process is restarted.

B. Scatter

The scatter primitive instead works differently. As mentioned in the previous section, the scatter protocol works with only one long RX window and sends one beacon in the remaining windows of the epoch. This primitive is simpler since we do not have to find the best number of packets to be sent to do not saturate the network; only one beacon has to be sent per window (which is considered as a task).

We start by declaring the starting protothreads called *scatter_-proc*. In this part define the *rtimer rt_off* and the *next_send* which is the fixed length of each window/task. The length of a window is given by the division between the epoch length (*RTIMER_SECOND*) and the number of tasks. Once defined, the radio is turned ON and the device could start to listen to the channel for incoming packets. Also, *callback_send_packet* is scheduled after next_send milliseconds by the *rtimer*. In this first amount of time the node could listen to the channel and process the beacons arriving from the neighbors with *ND_RECV* callback. This part is critical for the analysis since the Radio of the node has to be turned on, and this could be translated into energy consumption. Passed an amount of time equal to *next_send*, the *callback_send_packet* is called; in this function the radio is turned off and the current task counter is incremented by one at each call. If the last one is less than the declared number of tasks, the same callback is called after *next_send* milliseconds by the *rtimer*. On the contrary, if the number of current tasks exceeds the number of maximum tasks the callback *end_epoch* is called by the *rtimer* and it will start the end of the epoch processing to find the number of neighbors discovered from the reception window.

C. ND RECV - RECEPTION OF A BEACON

Every time that a non-corrupted packet arrived from the under layer, this callback is triggered. It is useful to extrapolate and process the payload of the packet containing the node id of the sender neighbor. After the decoding, the node id of the sender is stored into a static vector of Boolean as true in the position of the node id. Every element of the vector is set to false at the end of each epoch.

AVG_ON	2	5	10	20	30	50
10task - 10msRX - 14 sent beacon	9,6	9,6	9,6	9,6	9,6	9,6
10task - 10msRX - 20 sent beacon	9,25	9,25	9,25	9,25	9,25	9,25
10task - 10msRX - 28 sent beacon	9,62	9,62	9,62	9,62	9,62	9,62
20task - 10msRX - 7 sent beacon	18,43	18,38	18,36	18,34	18,33	18,34
20task - 10msRX - 11 sent beacon	18,66	18,62	18,59	18,59	18,59	18,59
20task - 10msRX - 14 sent beacon	18,86	18,75	18,75	18,75	18,77	18,74

Table I: Average Radio On wrt size of the network [BURST PROTOCOL]

Nodes configuration	2	5	10	20	30	50
10task - 10msRX - 14 sent beacon	100,00	61,75	51,00	64,53	58,00	40,45
10task - 10msRX - 20 sent beacon	100,00	80,00	70,33	63,11	56,38	46,18
10task - 10msRX - 28 sent beacon	100,00	90,25	75,78	71,00	56,83	40,59
20task - 10msRX - 7 sent beacon	100,00	88,00	88,31	82,96	77,49	70,2
20task - 10msRX - 11 sent beacon	100,00	88,88	79,74	85,29	76,84	69,03
20task - 10msRX - 14 sent beacon	80,00	93,73	76,96	75,33	74,05	61,58

Table II: Neighbor Discovery Rate over five simulations [BURST PROTOCOL].

IV. Evaluation

In this section a performance evaluation of the two primitives will be described talking in particular about metrics like the neighbor discovery rate and the radio duty cycle (percentage of Radio ON). The first metric is estimated by finding the average of discovered neighbors between all the nodes over the epochs. Instead, the average radio duty cycle is estimated automatically by the Cooja simulator by sampling the status of the Tmote for a certain time during the simulation. Some scripts have been developed to automate the simulation execution process and plot generation. The *run_all.py* is responsible for running all the simulations by type of primitive, then, it converts the results of the simulations into practical *.csv* useful for analysis through *arg_parser.py*. For the statistics, I decided to use R which is a statistical programming language.

neigh-discover.R is responsible for the plotting and statistical evaluation of all the metrics.
During the simulations the watchdog was disabled from */cpu/msp430/watchdog.c* (*watchdog_start* callback). For both primitive a fixed length epoch of 1 second was used ($EPOCH\_INTERVAL\_RT = RTIMER\_SECOND$).

The analysis will start from the Burst Primitive and then will follow with the Scatter one. In the following sections the *"task"* term will be mentioned more times; it is the division of the epoch in fixed and equal length intervals.

A. Burst Analysis

More tests were done for the burst configuration and more parameters were compared. The tested parameters were:

- number of tasks: 10 against 20.
- number of sent beacon in TX window (14-20-28 in the case of 10 tasks and 7-11-14 in the case of 20).

The analysis will start with two different tables which show the neighbor discovery rate and the radio duty cycle (Average Radio On) over the various parameter settings. Table I shows that decreasing the number of RX windows the Average on radio duty cycle decreases but, providing a low number of discovered neighbors. Table II instead, shows the neighbor discovery rate with respect to different number of tasks (10,20) and in the number of sent packet in the TX window. The best result is achieved with the configuration having 20 tasks and sending only 7 beacons. This is a good trade-off between the percentage of discovered neighbors and the radio duty cycle.

The best configuration is the one with 19 reception windows (with radio reception window = 10 ms) and one transmission window. 10 milliseconds is an excellent compromise to ensure low power consumption and an optimal number of received packets. Increasing the number of tasks increases at the same time the probability of discovering a new neighbor in the network but, worsening the radio duty cycle. Moreover, the lower is the number of sent beacons, the higher is the neighbor discovery rate; this is caused by the fact that if there are fewer packets in the channel, the lower is the probability of collision between them in big networks.

Figure 3 shows the results of the best burst configuration. This result is obtained by averaging the results of 5 different Cooja simulations with different seeds. By increasing the number of nodes in the network the neighbor discovery rate decreases because of packet collisions. Table I shows that the radio duty cycle is constant as the number of nodes within the network increases, keeping an average value of 18%.

Figure 4 shows the average duty cycle over 5 simulations. It is possible to see that the values plotted in the histogram are very similar in terms of energy consumption.

Nodes	ND	Variance	Percentage
2	1.00	0.00	100.00
5	3.52	0.24	88.00
10	7.94	0.82	88.30
20	15.76	2.20	82.95
30	22.47	4.28	77.48
50	34.40	7.83	70.20

Table III: ND and variance of the best configuration of Burst Primitive (20task - 10msRX - 7 sent beacon).

Table III shows the Mean ND per network configuration and the Variance. The Variance increases as the number of nodes

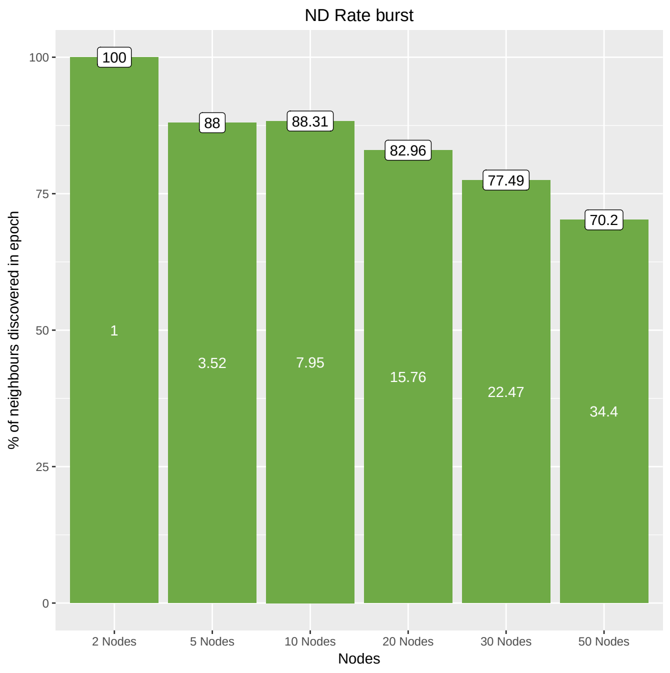

Figure 3: Burst neighbor discovery rate per network configuration.

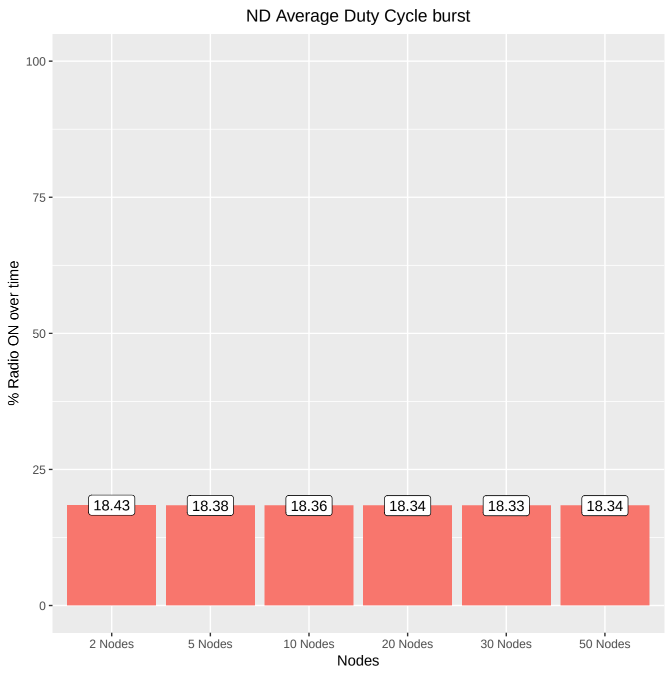

Figure 4: Average Radio Duty Cycle over five simulation of burst primitive.

increases inside the network. This last value seems to be pretty stable in this kind of primitive. We won't see the same in the next primitive.

B. Scatter Analysis

The scatter configuration was more "problematic" for the analysis. With this type of primitive, some nodes have failed in various simulations with large networks. It seems that by increasing the size of the network, the probability that some nodes fail increases. In the next part, two tables will be shown, as the previous section.

AVG_ON	**2**	**5**	**10**	**20**	**30**	**50**
5	16,79	16,75	16,7	17,66	17,77	27,09
10	9,65	9,64	9,6	9,62	9,61	9,61
15	14,14	14,1	14,09	14,09	14,06	14,06
20	18,85	18,8	18,8	18,75	18,77	18,74

Table IV: Average Radio ON [SCATTER PRIMITIVE]

As reported in Table IV it is possible to notice that the best configuration is the one dividing the time into 5 or 20 windows: one for RX while the others for beacon TX. As the number of tasks increases, the average on of the radio decreas, except for the 20 tasks configuration. Instead the neighbor discovery rate decrease by increasing the same value, as shown in Table V. Similar are the Neighbor Discovery Rates between the 5 and the 20 tasks configuration.

Table VI is useful to understand in a better way in which one of the two configurations is the best. The 20 tasks configuration seems to be better than the 5 tasks because it has a lower variance with the increasing of nodes in the network. In fact the variance with the 20 tasks is equal to $5.7 neighbors^2$ per epoch versus the $159.77 neighbors^2$ estimated from the other one.

% ND	**2**	**5**	**10**	**20**	**30**	**50**
5 tasks	90,00	77,02	85,02	79,81	77,00	63,34
10 tasks	80,00	82,96	68,43	64,15	54,85	39,86
15 tasks	60,00	83,51	78,95	73,22	66,82	53,77
20 tasks	80,00	89,98	84,20	76,64	70,29	63,54

Table V: Neighbor Discovery Rate over five simulations tests [SCATTER PROTOCOL]. An epoch is composed by a certain number of tasks which divides the time in fixed window. To obtain the number of sent packets on the scatter protocol it is needed to subtract one to the task value.

Image 5 shows the neighbor discovery rate in the primitive Scatter. One can immediately see that in the two-node network the primitive scatter (in the configuration with 5 windows) does not always find the neighbor every epoch (80%).

The Burst protocol seems to be much more performing than the scatter, obtaining higher percentages of discovered neighbors over the various node configurations. The scatter suffers a lot in configurations with many nodes (50), due to the low number of found neighbors. Having too much transmission windows many beacons collide worsening the results. Radio Duty cycle of the Scatter primitive with 20 tasks is quite similar to the best Burst configuration.

V. Conclusion

The Scatter primitive seems to be similar in terms of energy-consuming than the burst. This last, however, always obtains a higher neighbor discovery rate and an almost constant radio duty cycle over the various configurations and simulations. The

Nodes	ND	Variance	Percentage
2	0.80	0.16	80.00
5	3.59	0.41	89.97
10	7.57	0.73	84.20
20	14.56	2.62	76.63
30	20.38	5.71	70.28

Table VI: ND and variance of the best configuration of Scatter Primitive (20-task).

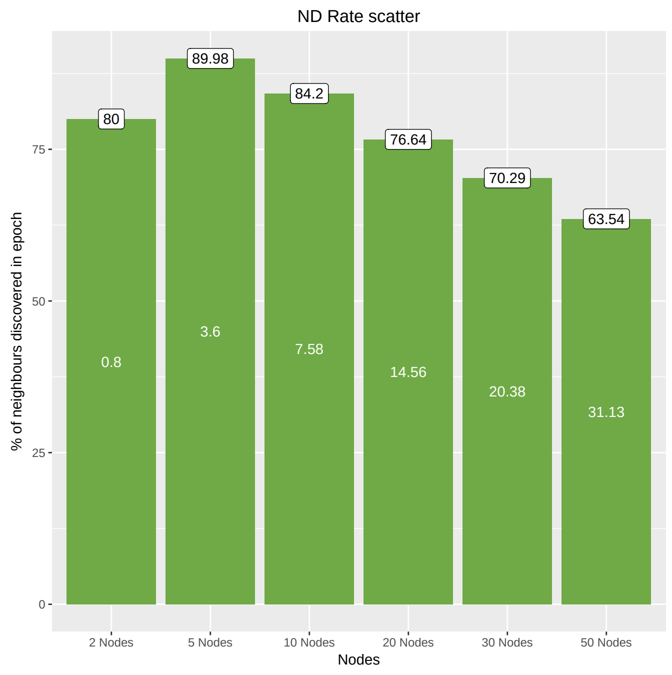

Figure 5: Scatter neighbor discovery rate per network configuration.

scatter seems to be more unstable as the number of transmission windows increases, leading to several node failures. This project was useful to put into practice the topics presented during the course.

REFERENCES

[1] “The tmote sky board.” [Online]. Available: http://contiki.sourceforge.net/docs/2.6/a01784.html

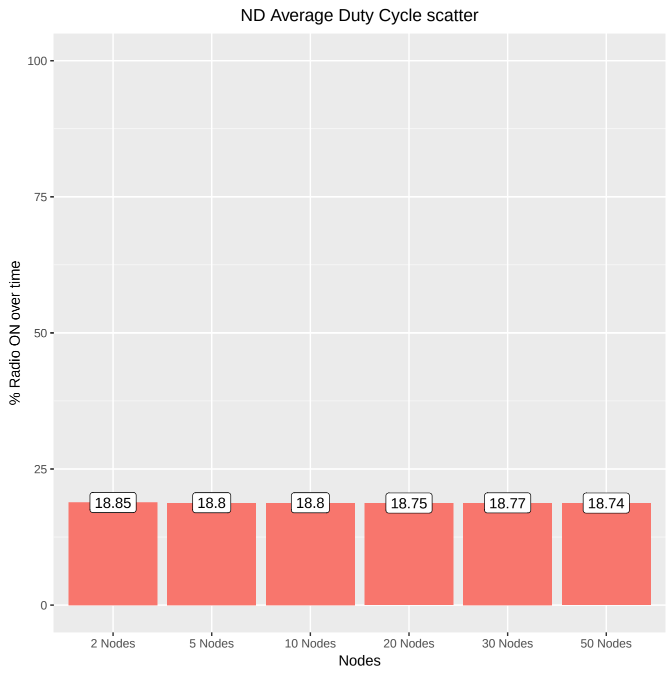

Figure 6: Average Radio Duty Cycle over five simulation of scatter primitive.