EMBS Assignment

Jake Coxon

December 5, 2012

1 Design and Implementation

I designed the way the program should work in theory first. The point of the source node is to transmit a packet during the reception phase of one of the three sink motes. For a given mote, any time a beacon is received with a payload n, there is nt time until the beginning of the reception phase and then $12t$ time until the beginning of the next sync phase. This can then be repeated indefinitely.

The first part of the program then is calculating t, which can be done by finding the difference in time between two separate beacons from a mote (t_1 and t_2). The beacons are not necessarily consecutive, so

$$t = \frac{t_2 - t_1}{n_1 - n_2}$$

$$\text{reception}_{\text{next}} = t_2 + n_2 t$$

$$\text{phase}_{\text{next}} = \text{reception}_{\text{next}} + 11t$$

An additional step can be added in the case where $n_{\max}$ is 1, this is harder since there is not two beacons in a cycle. When a beacon is received with the n value of 1, the program can check if $n_{\max}$ is 1 by waiting for $12 \times 500\,\text{ms}$ and checking if the first beacon received is 1. The program can then divide the total time between the two beacons by 12 to find t.

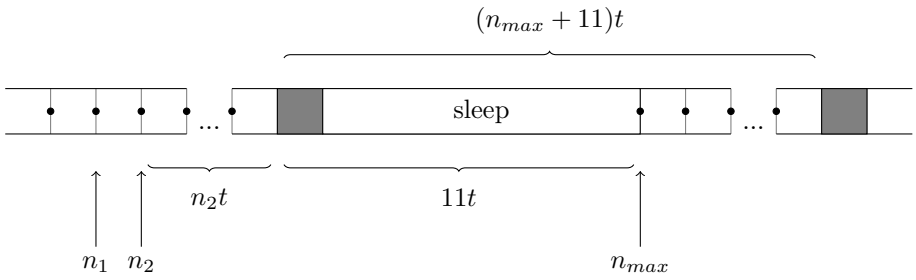

Figure 1: Timeline

2 Optimizations

As an optimization, after the second beacon has been received, the state of the sink is set to 'find-n'. When the the timer fires to start the synchronization phase the program knows that whatever n value is contained in the payload will be $n_{\max}$. This means the program can now skip out the synchronization phase and save power.

Minor optimizations that were considered included eliminating longs where possible—longs are slower on hardware because they are contained in two words—and using the *Immutable* tag on arrays that aren't written to. The program uses a timer to fire when it knows the radio needs to be in use.

3 Verification

My testing strategy was to simulate the source mote with three sink motes with different values of t and n. I wanted to automate as much as possible so a Ruby script was created. This script would—when given an index—pick three pairs of t and n values and generates a sink Java file respectively. This could then be automatically compiled, simulated with *mrsh* and then *grep*'ed to finally calculate the number of successful packets sent.

Testing arbitrary values helped me to find corner cases that I may have otherwise missed, for example when one sink needs to synchronize when another is already doing so. This testing strategy allowed me to find common problems such as this fairly quickly

It was important that the script run with an index would generate the same t and n values. This is so I could iteratively improve the algorithm and retest the same t and n values to see if the number of successful packets increased.

Sink A		Sink B		Sink C		No. Packets
n	t ms	n	t ms	n	t ms	
10	500	4	700	5	1500	14
2	541	3	912	6	1101	15
6	1407	5	567	2	1207	12
2	725	8	868	4	1043	15
10	683	7	1308	2	685	14
1	683	7	1308	2	685	17
2	500	2	500	2	500	9
7	1308	1	1500	2	685	4
8	1500	1	1000	7	1000	7

Figure 2: Test data and output