CS321

Information Processing for Sensor Networks

Achieving Desynchronization in Wireless Sensor Networks

Kevin Wong
Naef Imam

Contents

1. Introduction

There exist many applications in sensor networks that use time synchronization to distribute their processing over time intervals. This appears to be a contradictory method of achieving the inherent mechanism of interleaving processes, especially when time synchronization algorithms are expensive in terms of communication power and suffer many hardware issues. In their paper *DESYNC: Self-Organizing Desynchronization and TDMA on Wireless Sensor Networks*, Degesys, Rose, Patel and Nagpal aim to achieve this primitive via a biologically-inspired self-maintaining algorithm for a single-hop network.

Our work achieves an accurate implementation of the single-hop algorithm and aims to expand the algorithm to a multi-hop network by desynchronizing with two-hop neighbors. In this report we introduce implementation details for the single-hop algorithm, the enhancements in our application, testing strategy and results for the algorithm, multiple-hop algorithm including, implementation details, and evaluation.

2. Features and Implementation Details

The algorithm aims to desynchronize the nodes by scheduling the current node's next transmission time to fall midway between the measured transmission times of nodes that immediately preceded and succeeded its own transmission. A node achieves this by broadcasting Desync Messages on a periodic basis and listening to triggered events of other nodes to adjust its period offset. Below is a basic outline of a node's states:-

1. At startup a node initializes its period to a preset value sets up a timer to fire at regular intervals based on the period. If a separate period packet is received, the period packet is forwarded with a sequence number to all other nodes in the network.
2. Once the timer fires, a node broadcasts a Desync packet that contains its node ID and timing information of every one-hop neighbor. For a single hop network, this timing information is ignored. It then waits to record the incoming time of the

packet immediately after to schedule its next transmission time. This it done using the following formula:

Next Fire Time = Time Period + (1-α) Previous Fire Time
+ α x Avg. (Firing Time of Last Node, Firing Time of Next Node)
± a small random interval

We use the same value of $\alpha = 0.95$ as used by the authors of the paper, in combination with the random interval to account for errors. We make sure for now that the goal time is in the future.

3. Other nodes on hearing the message update their periods (if they are old, by comparing sequence number) and record the incoming packet time. The schedule their next transmissions if they were the last nodes to fire.

To achieve the above we have certain provisions in our implementation to take care of the boundary cases as well as improve the performance. Below are the salient features:-

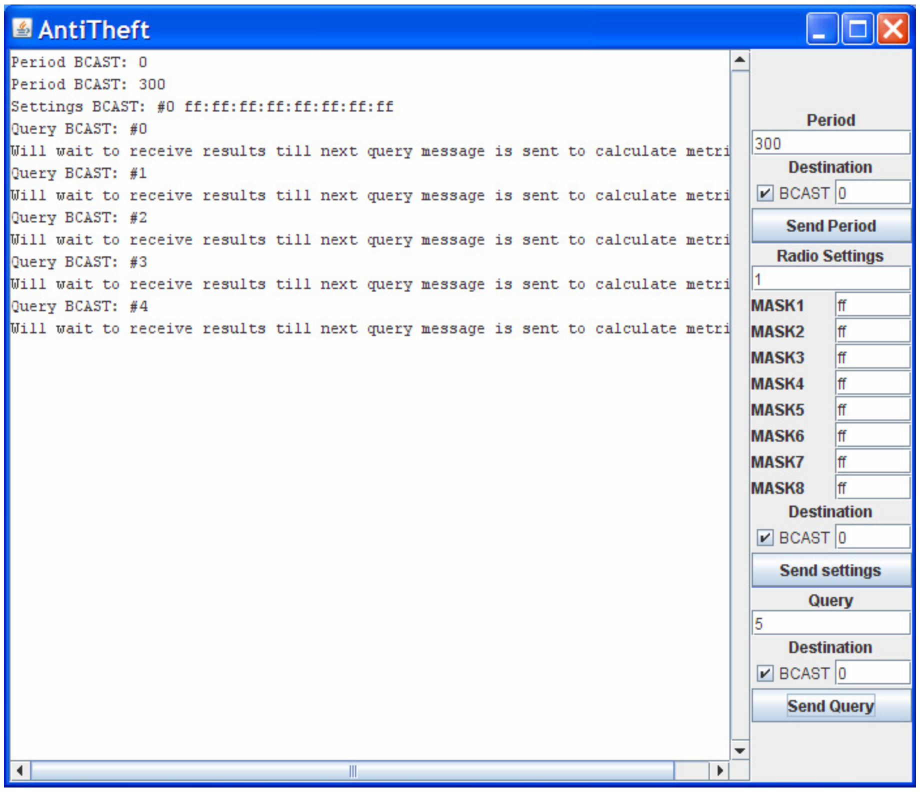
1. **Randomized firing time improvements** – There is a high possibility as period decreases and number of nodes in the connected graph increase, for two nodes to fire at the exact same time. This is especially true in the case when all nodes listen to the same initial period message and schedule the next trigger at the exact same time. Adding a small random value mitigates this risk, speeds up the convergence to steady state values and dampens wild fluctuations in future firing times. We add randomness by either adding or subtracting a 5 bit random number from the calculated packet time. This is accomplished by first subtracting 2^5 from the packet time, then adding a uniformly distributed random number from 0 to 2^10.
2. **Period message propagation** – Our implementation ensures the case that at least one node listens to the period message. Our implementation piggybacks on the Desync messages to propagate these messages. We also transmit 8 bit period sequence numbers to identify the latest period value in the network. A new period broadcast from the base station always increases our sequence number. If a node hears a Desync message containing an updated period value, it will change its period to that value as well as the sequence number to the new one. Nodes will ignore period updates with a smaller sequence number. This ensures that the

period propagates in the network even if just one node listens to the broadcast. This algorithm also does not differentiate whether the period value was transmitted at the beginning of the process or during the Desync process, hence giving us the flexibility to change the period value anytime.

3. **Counters and floating point operations** – Currently we use 32 byte unsigned integers and fixed point arithmetic to store and calculate the trigger times of various nodes. This is a problem when we have to do floating point operations as 64 bits are not supported in TinyOS 1.x. To avoid overflows on any of the intermediate arithmetic operations involved in calculating the proper transmission time, we had to reorder many operations. For example, we perform division operations before multiplication so the intermediate results will not exceed a 32bit number's range of expression. Initially, we had issues where the Desync calculations would fail after the local time of each node exceeded 2^30, since we multiplied by 95 and then divided by 100 to calculate a floating point multiplication of .95.
4. **Period timeout retransmission** – The main idea behind case is that if a node does not hear any transmission for more than one period, it defaults to a backup firing mechanism. This takes care of the degenerate case of a single node and also makes the algorithm robust to sudden large scale failures.
5. **TOS Changes** – We changed the size of the of the data length for iMote2 to include more data in our Desync packets. More specifically at path beta/platform/imote2/AM.h we changed the TOSH_DATA_LENGTH from 28 to 30. The original TinyOS 1.x default was 29, but the iMote2 requires and even byte count to achieve word aligned radio buffers, indicating that it is a 16-bit architecture.

3. Testing and evaluation

While evaluating our implementation we considered its accuracy, robustness and scalability. To ease our process we wrote a Java application to test the above implementation. The application was written on the lines of combining the AntiTheft.java application and TestDesyncM.nc. We can use this application to transmit a new period value to a particular node or broadcast it, change the RadioWrapper masks to test various topologies, send queries to the nodes to verify accuracy of our application as well as process incoming messages to calculate metrics. The implementation of this application is in the /MainProject_*/java folders in the submission. Below is a screenshot of the application.



```
Period BCAST: 0
Period BCAST: 300
Settings BCAST: #0 ff:ff:ff:ff:ff:ff:ff:ff
Query BCAST: #0
Will wait to receive results till next query message is sent to calculate metri
Query BCAST: #1
Will wait to receive results till next query message is sent to calculate metri
Query BCAST: #2
Will wait to receive results till next query message is sent to calculate metri
Query BCAST: #3
Will wait to receive results till next query message is sent to calculate metri
Query BCAST: #4
Will wait to receive results till next query message is sent to calculate metri
```

Period: 300
Destination: ✓ BCAST 0
Radio Settings: 1
MASK1: ff
MASK2: ff
MASK3: ff
MASK4: ff
MASK5: ff
MASK6: ff
MASK7: ff
MASK8: ff
Destination: ✓ BCAST 0
Query: 5
Destination: ✓ BCAST 0

Figure 1: Screen shot of testing GUI.

We conducted the following tests in series to ensure that our implementation is stable:

1. We started with a fully connected graph of 3 motes with T = 3 sec. After desyncing the motes settled to transmitting approximately every second.
2. We then switched off one mote and the other two motes settled to 1.5 seconds.
3. After switching off another one, the only node remaining fell back upon default firing times of 3 second in the absence of any other messages.
4. We then added back another mote back to reach the steady state in 2.

5. We then changes the time period to 5 seconds and added two more motes to observe that all motes were switching on at approximately 1.25 seconds and that hence period had propagated to nodes that hadn't heard it before.
6. We then increased the number of motes to 8 at once and saw the system stabilize; same for removing 4 nodes at one go.

To evaluate the application we aimed to minimize the error in the system that is given by:

$$\varepsilon = \frac{1}{N}\sum_{i=0}^{N-1}\left|\left(t_{(i+1)\%N} - t_i\right)\%T - \frac{T}{N}\right|$$

To calculate the metrics for a fully connected network we used the T = 2 seconds and vary the number of motes. Although our algorithm converges in a matter of seconds, we let the systems stabilize for 3 minutes before taking the readings:

1. For 3 motes we calculated the average error to be .008687 seconds = 0.43% of the time period, this can be easily accounted by clock drifts.
2. For 5 motes the average error was 0.0648615 seconds = 3.23%.
3. For 6 motes the error was 0.006502 seconds = 0.3251%

Convergence: We used the Java application to record the firing times of motes at the end of every time period. Over the above 3 experiments (each repeated multiple times) the nodes took an average of 15 time periods or an average of 30 seconds to reach the steady state.

Varying Time Periods – Reducing the period time will decrease the settling time, because nodes transmit more often. It will although raise the absolute error because of the timing inconsistencies. But because they settle soon we expect this to compensate. Increasing the time period should make the convergence proportional to the number of nodes, as how often I hear transmissions messages affects how fast I converge.

4. Multi-hop Desync

For a fully connected network, desynchronization works quite well as all nodes can hear each other and select an appropriate time to transmit between two other nodes. However, once we have multi-hop networks, nodes cannot hear all other members of the network, which can lead to hidden terminals and general difficulty in maintaining a perfect desynchronization. The regular desync algorithm would probably work adequately for some topologies, but for the case of a three node linear topology, the nodes could never settle on a set transmission schedule because the two outer nodes would always collide with the center node. Thus we decided that we should attempt to desync with all two hop neighbors, since beyond two hops, we are unlikely to cause interference and hidden terminal issues with the neighboring nodes. In effect, this approach attempts to simulate a fully connected graph for topologies with a diameter of 2.

Our algorithm approaches the problem by calculating optimal transmission times for each node, thus creating certain overlapping clusters. We discuss the algorithm in the next sub-section, then compare it with other algorithms and finally give some experimental results of implementation on iMote2 motes.

4.1 Algorithm

Below are the enhancements to the single-hop version that we have implemented:

1. When a node fires, it includes previous transmissions times of its one hop neighbors relative to the current time, in the Desync message. In our implementation we include 32 bit delta information from each of our neighbors to schedule our transmission time. The only extra information transferred is a table of firing times for all one hop neighbors that the node can hear. No time synchronization is required as all times transferred are relative values from current transmission time. This also makes the system robust by accounting for nodes that are currently up rather than track who joined and who left.
2. The transmission timing information is only forwarded to one hop neighbors. This limits our information transfer.
3. To schedule its next transmission time a node searches through information of the fire times and locates the times of nodes that fired just before it and just after it.
4. If a node dies, its delta will continue to increase until it is no longer considered by our algorithm.

4.2 Comparison with other algorithms

Our algorithm does not involve complete randomness, nor does it attempt to create a global structure. While creating a sub-network of two hop neighbors our algorithm creates overlapping pseudo-connected clusters (as nodes don't really hear each other but still know the transmission times). This is somewhat similar to creating maximal cliques, but not the same and we discuss the differences below:

1. Randomized algorithm is highly inefficient as they can lead to high fluctuations in firing times.
2. Creating a fully connected graph requires re-broadcasting transmission times across the network. Essentially, each node forwards any transmission time data it gets to all its neighbors leading to redundant packet transmissions and lowering the capacity of the network. This is highly inefficient especially when two nodes that don't affect each other in any way could schedule same alarm firing time. Hence, we include transmission time information that is directly relevant to us and leave the others out. A simulated fully connected graph is not a scalable option either, as synchronization overheads lead to barrage of messages when nodes frequently join or leave the network.
3. Creating maximal cliques also leads to a graph setup delay including meta-information including node-ID's being transferred. On the other hand it can also lead to wildly fluctuating firing times at particular nodes at the confluence of two cliques. These nodes would have to employ certain metrics like probabilistic weights to decide when to fire, especially when two cliques have huge difference in the number of nodes. For example, in Figure 2 assuming a Time Period of 1 second, there will be a node firing every 1/10 second in the right clique once every 1/3 seconds in the left clique. We overcome this problem by including 2-hop neighbor information and further simplify the NP-hard problem by not requiring any other kind of setup.

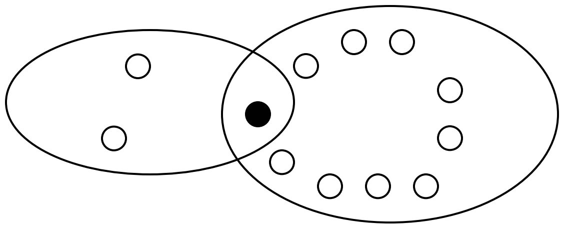
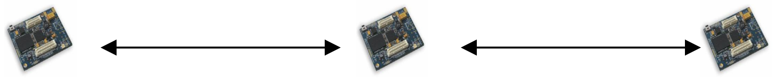


Figure 2

4.3 Experimental Results

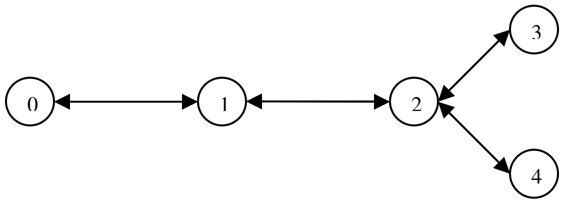
We base our evaluation on the same metrics as the single-hop case except we do it on a per node basis. So for $\varepsilon = \frac{1}{N}\sum_{i=0}^{N-1}\left|\left(t_{(i+1)\%N} - t_i\right)\%T - \frac{T}{N}\right|$; N is the number of 1-hop neighbors of a particular node. We evaluate our algorithm on a number of topologies representing special and general cases. For all the cases we use T = 2 seconds.

1. One of the degenerate cases that we tested was the linear 3 topology. This should in effect behave like fully connected graph according to our algorithm.



With T = 2 seconds, the average error of the central node was 0.098878 seconds and at end points was 0.34139seconds and 0.33834seconds. The global average error of the system was .259536 seconds or about 12% of T. This is due to the bias because of the end nodes that did not know the structure of the whole system. But for the central node that actually had the complete picture the error went down to 4%.

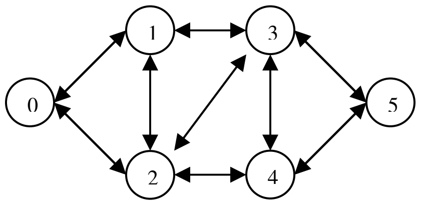
2. We then calculated a Y topology with 5 motes, as in the diagram below



The average global error was 0.14021 seconds and at nodes 0 to 4 were 0.36217 seconds, 0.054459 seconds, 0.075264 seconds, 0.46329 seconds and 0.46329 seconds respectively. The same observation about the metric accuracy can be made here with errors from nodes 1 and 2 being much lower.

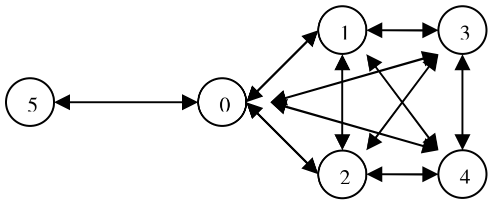
3. In a star topology with one central and 5 star nodes we tested the possible case when one node is a member of many cliques (in this case 5). While the central node has an error of 0.0082164 seconds the average error for the end nodes is .40783 seconds. Steady state for this situation is when central node goes off once and after a second all the end-point nodes transmit together. But it takes almost infinity to reach that state, during which the node assumes almost perfect synchronization with the end-points as shown by its error.

4. We used the following very generalized topology



The global error is 0.13255 seconds or 6.63% of the time period and at nodes 0 to 5 are 0.21835 seconds, 0.21835 seconds, 0.16553 seconds, 0.16553 seconds, 0.19392 seconds and 0.21291 seconds respectively.

5. We use the following topology



The error at 0 is 0.017195 seconds, average error of nodes 1 to 4 is 0.10771 seconds
And, at node 5 it is 0.6418seconds

4.4 Conclusion

Our solution for multi-hop appears to work for small number of nodes but we did not test or simulate for a large number of nodes hence we cannot conclude about its scalability and robustness. Although we are convinced that since the algorithm does not require any large scale information transfer and all events are localized, that it will scale well. As time period increases, the average settling time would depend upon the number of nodes as more nodes will have more transmissions and we will be able to calculate our transmission time. The above implementation is a more efficient, and practical solution for desynchronization is multi-hop networks as discussed in section 4.2, with minimal error rates.