Design and implementation of a multi-hop desynchronization algorithm for Wireless Sensor Networks

Ian Downes, Gonzalo Vazquez Vilar, Yiannis Yiakoumis
{downes,gvazquez,yiannisy}@stanford.edu

1 Table of contents

2 Introduction

This report discusses the implementation of the single-hop desynchronization algorithm as proposed in the IPSN 2007 paper *“DESYNC: Self Organizing Desynchronization and TDMA on Wireless Sensor Networks”*. The algorithm was implemented and tested on the Intel iMote2 hardware platform running the 1.x branch of the TinyOS operating system.

The original single-hop algorithm has also been extended and modified to allow network wide updates to the desync period and to ensure that entire network maintains a consistent period even when nodes enter or leave the network.

Furthermore, an algorithm has been developed and tested for multi-hop networks. The algorithm defines an error function and performs steepest descent optimization where each node directs neighbouring nodes to reduce the error. The algorithm has been tested under a custom Matlab simulator and has also been implemented on the iMote2 platform.

3 Original algorithm: One hop networks

For a single-hop network, our implementation is strictly based on the desync paper. The main issue here – apart from translating the desync idea to TinyOS code - is how to maintain the period. In addition, we took care of clock overflow (~20 mins) by checking (and when needed, fixing) our timer-variable relations. Both of these solutions make our algorithm robust and reliable for long term usage. We have tested the alpha parameter and found the value also described in the original paper (alpha = 0.95) to be suitable.

3.1 Period management scheme

The period management scheme described below applies both for the single-hop and the multi-hop versions. While trying to implement desync, we had to decide what kind of packets we should use and what information they should contain. To maintain the simple nature of the algorithm nodes only send (modified) desync packets. Period packets are sent only by the base-station and are interpreted by the nodes.

The original desync messages are modified to carry extra information, necessary for our algorithm. Understanding the impact on the communication overhead, we tried to eliminate this piggybacked information, and send it only when necessary. In the single-hop version, a desync package includes 5 bytes (besides those for a standard TOS message):

- 1 byte for source address
- 2 bytes for period
- 2 bytes for period-timestamp

They are necessary to maintain the period provided by the base station, and their use becomes clear while explaining our algorithm. We assume that there is only one base-station, and that the nodes should always run using the most recent period given by the base-station. In order to do this, we use the timestamp idea. Every node maintains its period with an associated timestamp.
When a node starts, it has a default period (in our case 1 second) and timestamp 0. When it receives a new period message from the base station, it adopts the new period and increases its timestamp to match that of the new period. Supposing that all its neighbours had the same period-timestamp pair, it advertises the new pair which is then adopted by them. As we said before, extra effort was paid to keep bytes transmitted as low as possible. There are four cases when a node should send its period-timestamp pair.

- When a node starts up, it actually requests for a valid period using a meaningless timestamp 0 (the first period provided by the base-station will take timestamp 1).
- When a node renews its period (after receiving a more recent (= higher timestamp) period), it advertises-propagates it to others.
- When a node realizes that another node has an older period (receiving period with lower timestamp than its own), it informs the node about the current period.
- When a node meets new neighbours, it informs them about its period-timestamp, thus initiating a procedure which will end up in a neighbourhood where all nodes have the same - and more recent - period.

The latter case represents the scenario where a node with an incorrect period enters the network. Nodes will see that they have a new neighbour and will try to ensure whether it has the right period or not. In order to do that, we keep a one-period history of our neighbours (source address used). If we have seen a node in the previous period, we can assume that we already have a common period - and if not we will through the other mechanisms. Otherwise, we send our period-timestamp pair, and the other node will either adopt our period or inform us of its more recent period.

Thus, we have limited sending extra information in the afore-mentioned cases. In all other cases, our desync messages will include only one byte of information (source address which is always necessary to keep 1-hop neighbours information). Note here that currently, for testing needs (our Java testing requires that all desync messages should be of equal size), every desync packet contains all extra information plus a byte saying whether this extra info is valid or not. This is just for testing and plays no role in our algorithm (we actually discard these bytes in the receiver part).

4 Design of a desyncronization algorithm for multi-hop networks

4.1 Introduction

The previously discussed algorithm is suitable for one hop networks where each node essentially has global information. That is, each node can hear the firing time of every other node that is present in the network. Such fully connected networks are very rare, especially in wireless networks where fading, obstacles and irregular placement disrupt communication links. It is natural therefore to consider extending the algorithm for multi-hop networks.

4.2 Single-hop desync applied to a multi-hop network

Before continuing with the development of a multi-hop algorithm it is worthwhile to first consider the performance of the original single-hop desync algorithm if it is applied to a simple multi-hop network. In Figure 1(a) a three node network is shown where the centre node can hear both neighbours but each neighbour can only hear the centre node. The single-hop desync algorithm will quickly converge to the phase diagram shown in (b). From the perspective of the two outer nodes this is the optimum layout – each is maximally distant from their single neighbour (the centre node) in phase. For the centre node it sees that its two neighbours are actually completely synchronized. Depending on the use of the network (e.g. desynchronized sampling, TDMA, etc.) this likely to be undesirable.

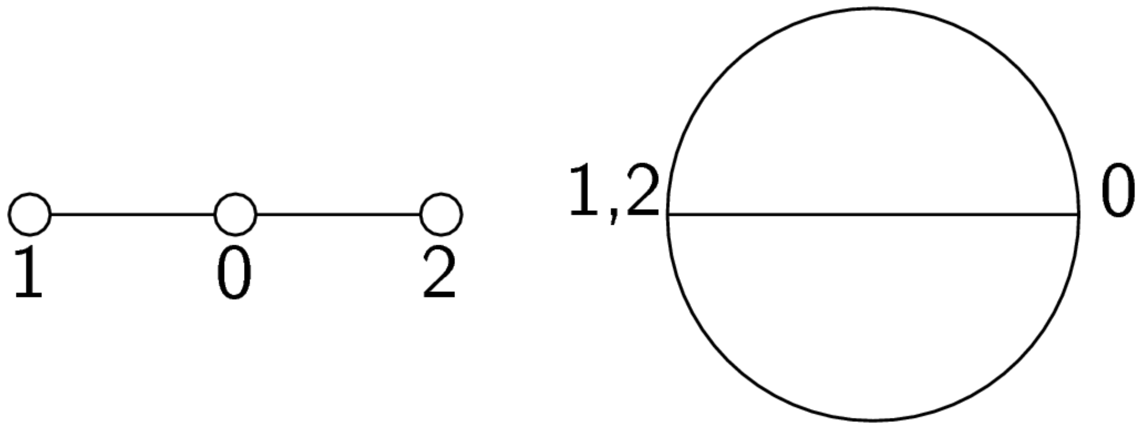


Figure 1 Original desync algorithm applied to a simple multi-hop network. (a) Three node network where the centre node can hear both neighbours but each neighbour can only hear the centre node. (b) Optimum phase diagram using the original desync algorithm.

4.3 Metrics

One hop network

In the one hop case, we have that the network of N nodes is fully connected. If we consider the firing time of each mote globally we sort the times and denote them as follows:

$t_0 < t_1 < \ldots < t_{N-1}$

We can then express the average phase error as

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

The operator % represents the operator modulo as we are computing the relative error between all the nodes within a desync period. Since under ideal conditions the *N* motes will have a spacing of T/N the above equation denotes the average error of each interval from the optimum separation.

Multi-hop network

In the project proposal, this metric has been extended for multi-hop networks. In this case there is still a global ordering on the motes, but it is not clear what the “correct" metric is. The metric of the proposal is an average of the metric defined in (1) taken at each mote:

$$\begin{aligned} \varepsilon_j &= \sum_{i=0}^{n_j-1} \left| \left(t_{(i+1)\%n_j} - t_i \right) \%T - \frac{T}{n_j} \right| \\ \varepsilon &= \frac{1}{N} \sum_{i=0}^{N} \varepsilon_i \end{aligned} \tag{2}$$

Proposed metric

However the metric defined above does not take into account the “importance” of the nodes and fails under certain conditions. A more suitable metric may be to weight the error at each node by the degree of the node. In such a metric, nodes with high degree are deemed more important. This would be appropriate for example if the algorithm was used to determine TDMA slot times. High degree nodes would be seen as aggregation centres or perhaps as bottlenecks. In either case it would be important to optimize the throughput at these nodes, possibly at the expense of other nodes.

In light of this the metric is modified to include weighting the error by the node degree.

$$\varepsilon_j = \sum_{i=0}^{n_j-1}\left|\left(t_{(i+1)\%n_j} - t_i\right)\%T - \frac{T}{n_j}\right|$$
$$\varepsilon = \sum_{i=0}^{N}\frac{n_i}{N}\varepsilon_i \tag{3}$$

Consider for example a star configuration with one centre node and four surrounding nodes (similar to Figure 1). Due to the high number of neighbours the unweighted error is dominated such the minimum error configuration is as shown in Figure 2 (a). This is clearly not optimal for a TDMA scenario as all packets would collide at the centre node. If the weighted metric is used then the configuration of Figure 2 (b) is obtained where packets sent to the centre node would not collide.

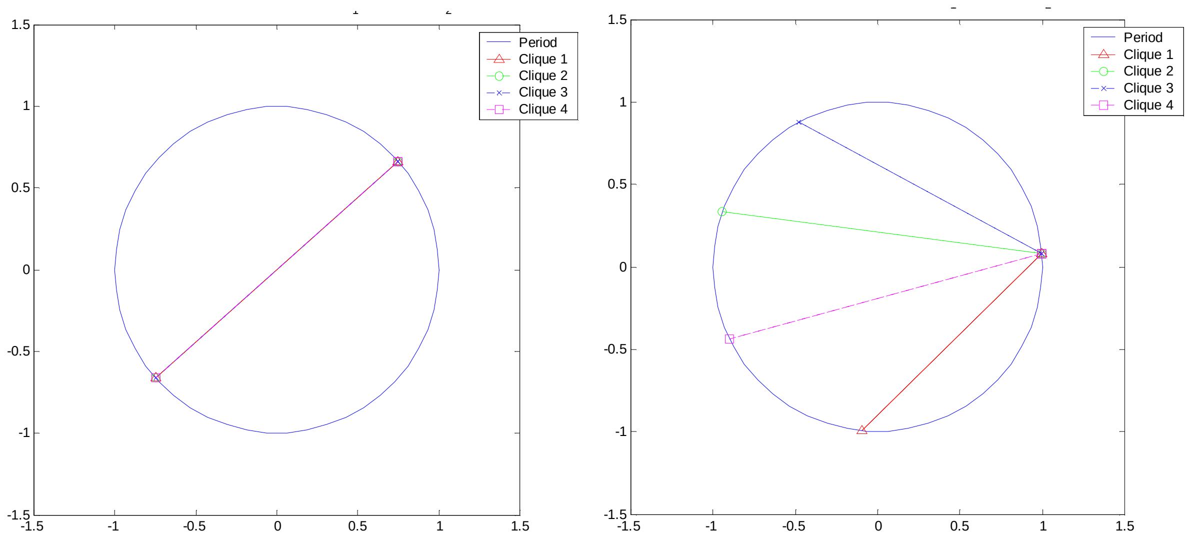


Figure 2 Representation of the beacons in the period: Situation (a) $\varepsilon = 0.24$ following metric (2) $\varepsilon = 1.2$ following metric (3). Situation (b) $\varepsilon = 0.29$ following metric (2) $\varepsilon = 0.81$ following metric (3).

4.4 Suboptimal approximations

Since an algorithm computing the optimal position explicitly is too expensive for a sensor network, we studied two suboptimal approximations which try to find one solution “close” enough to the optimal in a distributed way.

- In the first one, the approximation taken is to include a node’s one-hop neighbours in its desync message. This provides each node with information about its two-hop neighbourhood.

- The second one uses the fact that after a desync period each node has enough information to calculate the exact error of its neighbourhood, and therefore try to minimize it by changing its own position and influencing the position of the other neighbours.

In the next two sections we will describe these two approximations:

4.5 Two-hop-neighbourhood data

The idea of this approximation is that most of the information about a network is contained in a node's two-hop neighbourhood. If each node can cooperatively determine its optimal phase for its two hop neighbourhood then the arrangement may not be globally optimal but it will ensure that collisions are avoided. Such an algorithm may therefore be most suited for determining TDMA slots.
The main idea of the algorithm is that at each update step a node uses its two-hop neighbour information to decide how to shift its phase. It evaluates the error metric to determine which out of several possibilities will locally produce the best solution. Note that this approach permits a node to knowingly worsen its own error if it believes that it will improve the combined error.
The three classes of phase operations are as follows

1. SWAP:- Evaluate swapping phase with a one-hop neighbour. If a phase swap is determined to have a net reduction in the error metric then either a specialized swap message can be sent to the node or simply jump to a time just before the node will be fire, effectively displacing it from its phase.
2. JUMP:- Evaluate jumping to a new phase. To limit the number of phase points considered only the midpoints of neighbouring phase points are considered.
3. DESYNC:- If neither a SWAP or a JUMP is determined to reduce the net error then adjustment of the phase is performed using the normal desync algorithm.

This algorithm has the advantage that it easy to implement different error metrics without affecting the overall algorithm. However, a major disadvantage is that it requires the continual transmission of two-hop information. Each node transmits O(d) extra bytes of information (d = degree of node). In most networks d is relatively small however this is a significant increase in communication.

4.6 Neighbour directed gradient descent algorithm

The main idea of this approximation is that each of the nodes can calculate an exact error associated to its neighbourhood. Remember that the error associated to the node j in metrics (2) and (3) was obtained by:

$$\varepsilon_j = \sum_{i=0}^{n_j-1} \left| \left(t^{(j)}_{(i+1)\%n_j} - t^{(j)}_i \right) \%T - \frac{T}{n_j} \right|$$

Note the change in the notation to make clear that $t^{(j)}_i$ represents the fire time of the ith node in the neighbourhood of the node j. We can define a new error $\tilde{\varepsilon}_j$ for the node j by squaring the terms of the sum of ε_j.

$$\tilde{\varepsilon}_j = \frac{1}{2} \sum_{i=0}^{n_j-1} \left(\left(t^{(j)}_{(i+1)\%n_j} - t^{(j)}_i \right) \%T - \frac{T}{n_j} \right)^2$$

Note that the error evaluates to zero when all the neighbour nodes of j are equally spaced around the period and that $\tilde{\varepsilon}_j \to 0 \Rightarrow \varepsilon_j \to 0$. Using the error $\tilde{\varepsilon}_j$ a node can construct a gradient function depending on the firing time of each of the nodes in its neighbourhood:

$$\frac{\partial \tilde{\varepsilon}_j}{\partial t_k} = \frac{\partial}{\partial t_k} \frac{1}{2} \sum_{i=0}^{n_j-1} \left(\left(t^{(j)}_{(i+1)\%n_j} - t^{(j)}_i \right) \%T - \frac{T}{n_j} \right)^2 =$$

$$= \frac{2}{2} \left(\left(t^{(j)}_k - t^{(j)}_{(k-1)\%n_j} \right) \%T - \frac{T}{n_j} \right) - \frac{2}{2} \left(\left(t^{(j)}_{(k+1)\%n_j} - t^{(j)}_k \right) \%T - \frac{T}{n_j} \right) =$$

$$= \left(\left(t^{(j)}_k - t^{(j)}_{(k-1)\%n_j} \right) \%T - \frac{T}{n_j} \right) - \left(\left(t^{(j)}_{(k+1)\%n_j} - t^{(j)}_k \right) \%T - \frac{T}{n_j} \right)$$

This discussion suggests a steepest descent algorithm for changing the position of the node firing a time $t^{(j)}_k$ in order to reduce the contribution to $\tilde{\varepsilon}_j$ by the error of the node j. Since we are interested in reducing the error we will choose the negative descent direction:

$$t^{(j)}_k(m+1) = T + t^{(j)}_k(m) - \alpha \frac{\partial \tilde{\varepsilon}_j}{\partial t_k} =$$

$$= T + t^{(j)}_k(m) - \alpha \left(\left(\left(t^{(j)}_k - t^{(j)}_{(k-1)\%n_j} \right) \%T - \frac{T}{n_j} \right) - \left(\left(t^{(j)}_{(k+1)\%n_j} - t^{(j)}_k \right) \%T - \frac{T}{n_j} \right) \right)$$

where α represents the step size of the algorithm and $t^{(j)}_k(m)$ the firing time for the desync period number m. Note that these correction quantity to apply in the node firing at $t^{(j)}_k$ can be easily calculated with the information present in the node j, since a node knows its number of neighbours and the fire times of each of them.

Global error minimization

However we pretend to reduce the error on the whole network and not only at one node. We can write the error for the whole network following the metric (2) (extension to metric (3) is immediate) as:

$$\varepsilon = \frac{1}{N}\sum_{i=0}^{N}\varepsilon_i \approx \frac{1}{N}\sum_{i=0}^{N}\tilde{\varepsilon}_i$$

If we consider now the contribution of the position in the period of one node j to the whole network error we have that

$$\frac{\partial\varepsilon}{\partial t_j} \approx \frac{\partial}{\partial t_j}\frac{1}{N}\sum_{i=0}^{N}\tilde{\varepsilon}_i = \frac{1}{N}\sum_{i=0}^{N}\frac{\partial\tilde{\varepsilon}_i}{\partial t_j}$$

That is, each node contributes to the global gradient $\partial\varepsilon/\partial t_j$ with a sum of all the individual gradients $\partial\tilde{\varepsilon}_i/\partial t_j$ with respect to its neighbours. Thus we can write the global update formula for the node j as:

$$t_j(m+1) = T + t_j(m) - \alpha\frac{\partial\varepsilon}{\partial t_j} \approx$$

$$\approx T + t_j(m) - \alpha\frac{1}{N}\sum_{i=1}^{n_j}\frac{\partial\tilde{\varepsilon}_i}{\partial t_j} =$$

$$= T + t_j(m) - \alpha\frac{1}{N}\sum_{i=1}^{n_j}\left(\left(\left(t_j - t^{(i)}_{(k-1)\%n_i}\right)\%T - \frac{T}{n_i}\right) - \left(\left(t^{(i)}_{(k+1)\%n_i} - t_j\right)\%T - \frac{T}{n_i}\right)\right)$$

where k is the position of the node j in the neighbourhood of i; $t_k^{(i)} = t_j$

Information exchanged

Note that the update formula for the timing of the node j depends on the corrections calculated in the own node $\partial\tilde{\varepsilon}_j/\partial t_j$ and in the corrections from all its neighbours i: $\partial\tilde{\varepsilon}_i/\partial t_j$. Thus, for each iteration of the algorithm a node j has to share a number of corrections $n_j - 1$ addressed to its neighbours.

To avoid this overhead of communication we chose to reduce the number of neighbours addressed in each iteration to one. Moreover, to avoid flooding the network with extra messages, the information will be attached to the current DESYNC message sent by the node. The neighbour to which the gradient is sent can be chosen by a round robin or a randomized selection procedure. All the nodes must now keep a table with the last error-reports received from their neighbours and they keep updating it when they receive a new message addressed to them. The size of the information exchanged by the algorithm is described in section 5 which discusses the practical implementation of the algorithm.

This design decision is however arbitrary and we could choose to send the error calculated to two or even more nodes. This allows a trade-off between the overhead introduced by the algorithm, and the velocity of convergence and stability one convergence is achieved.

Local minima

The gradient function $\partial\varepsilon / \partial t_k$ presents multiple local minima and the algorithm can get stuck depending on the initial firing times of the motes, on the desync period and on the topology of the network. That can happen for example when one node is in the way of one of its neighbour nodes to their optimal position. In this case the algorithm will stabilise in this local minimum and not evolve towards the global minimum.

To avoid these situations we propose a series of heuristics that detect when the algorithm is stuck in one of these local minima, and to jump to a situation where the algorithm can continue evolving.

To understand the JUMP conditions we must remember the equation controlling the evolution of each of the nodes:

$$t_j(m+1) = T + t_j(m) - \alpha \frac{1}{N} \sum_{i=1}^{n_j} \frac{\partial \tilde{\varepsilon}_i}{\partial t_j}$$

Since we are stuck in a local minimum we have that:

$$\frac{\partial \varepsilon}{\partial t_j} \approx \frac{1}{N} \sum_{i=1}^{n_j} \frac{\partial \tilde{\varepsilon}_i}{\partial t_j} = 0$$

And the node will not change its position. We can see this situation as a force equilibrium since a node is receiving the same force to move in one direction as the force to move in the opposite direction. Figure 3 shows graphically this interpretation of the errors received from the neighbours. If the two opposite forces present in a node are large it means that this node is contributing to the global error by a large amount and that it does not allow the neighbours to evolve to their equilibrium position.

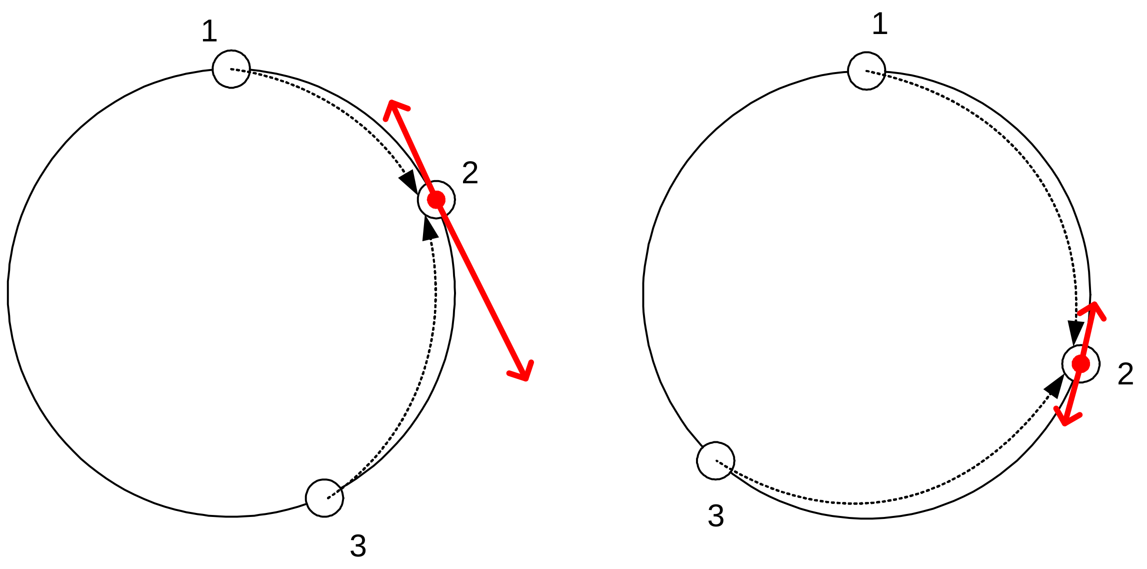


Figure 3   Example of the forces applied to node 2 in the case of a three node full connected network.

The heuristic we propose is that if the force in one direction is larger than the distance to the closest neighbour in this direction we choose to jump over it with a small probability. We jump with a small probability to avoid multiple simultaneous jumps, since in a stuck situation it is possible that several nodes are with high tension and decide to jump simultaneously. This situation is shown in the Figure 4.

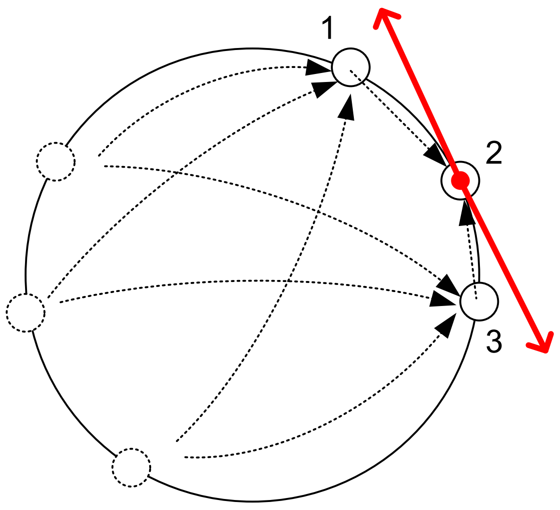


Figure 4   Example of a local minimum in a multi-hop network. Nodes 1 and 3 can hear all the nodes on the network while node 2 can only hear nodes 1 and 3. The forces applied to 2 are bigger than the threshold defined by the positions of its neighbours and therefore would choose to jump with a small probability.

Robustness of the algorithm

It is interesting to remark that this algorithm makes no assumptions about the topology of the network, or about the stability of the nodes. If the topology changes the algorithm adapts itself to the new situation and derives the global minimum of the new configuration.

Another advantage of using this scheme is the scalability of the solution. Since the algorithm is totally distributed it could be extended to arbitrary topologies without limiting the number of nodes.

The result of this simplicity and adaptability is in the convergence speed, which cannot be the same as in the case of calculating directly the optimal situation and the transmitting it to the rest of the nodes. Our algorithm needs several desync periods in order to converge to a stable solution.

4.7 Simulations

The capabilities and performance of the algorithm exposed in 4.6 have been assessed by computer simulation using Matlab. This event oriented simulator shows in a diagram the evolution of the different nodes in a period and is able to perform simulations of homogeneous networks (all the nodes with the same code) with a specified topology and a certain probability of packet loss.

The simulations of the algorithm showed good behaviour in most of classical topologies such as star, ring and one hop networks. The algorithm was shown to be capable of finding a quasi optimal solution for most of the more complex scenarios studied. In the following points we will show the solutions obtained by the algorithm for some of this topologies.

Graphical representations used in this section

We chose to present the firing times of the different nodes in a circle representing the desync period. The nodes are connected by lines of different colours for different cliques in order to make it easy to see which neighbours can hear each of the nodes.

Five nodes ring

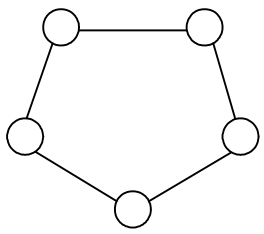


Figure 5 Topology of a ring of five nodes.

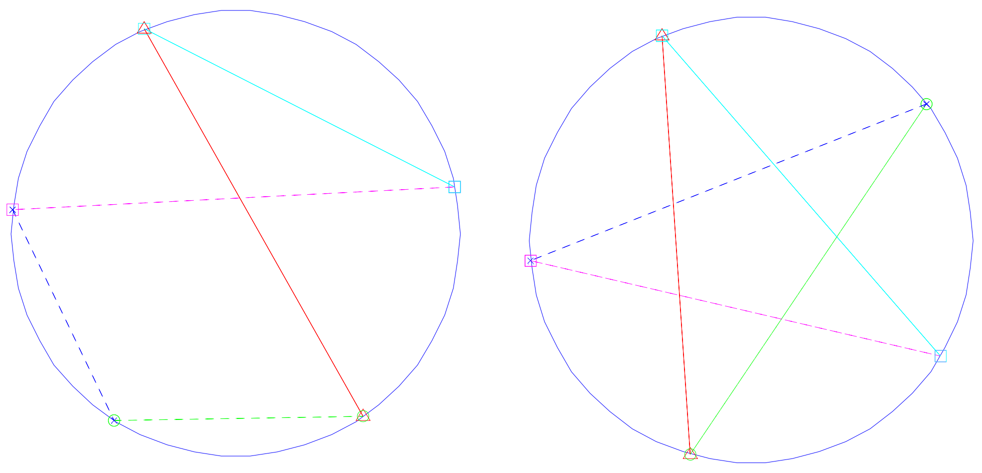


Figure 6 Evolution of the nodes in the period cycle:
(a) Local minima (b) Final solution

5 Algorithm implementation for the platform: imote2

The algorithms described in the sections 4.6 (Neighbour directed gradient descent algorithm) and 3.1 (Period management) have been implemented for the imote2 platform and tested for different multi-hop topologies with up to 8 nodes.

To simplify the implementation it has been assumed that the nodes of the network have address numbers ranging from 0 to 7. This assumption is arbitrary and the implementation can easily be extended for the more general case where the space address is bigger. The only assumption of the algorithm presented in 4.6 is that the two hop neighbours of a node must have different addresses or identification numbers.

We have chosen to send only three extra bytes during the normal operation of the algorithm. In each desync period each node randomly chooses one of its neighbours which will be the receiver of our gradient information. The data sent is:

- 1 byte for source address
- 1 byte for report receiver address
- 1 byte for the error report

Four additional bytes are attached to the message as necessary to update and maintain the desync period. This has been described in section 2.1

Another problem in the implementation of the algorithm for the imote2 platform is the lack of floating point support and that the original algorithm was designed to work with infinite precision. To minimize the complexity of the compiled code and to avoid the use of software floating point emulation we decided to use fixed point arithmetic.

For the implementation of the algorithm further problems had to be solved. Some of them were the following:

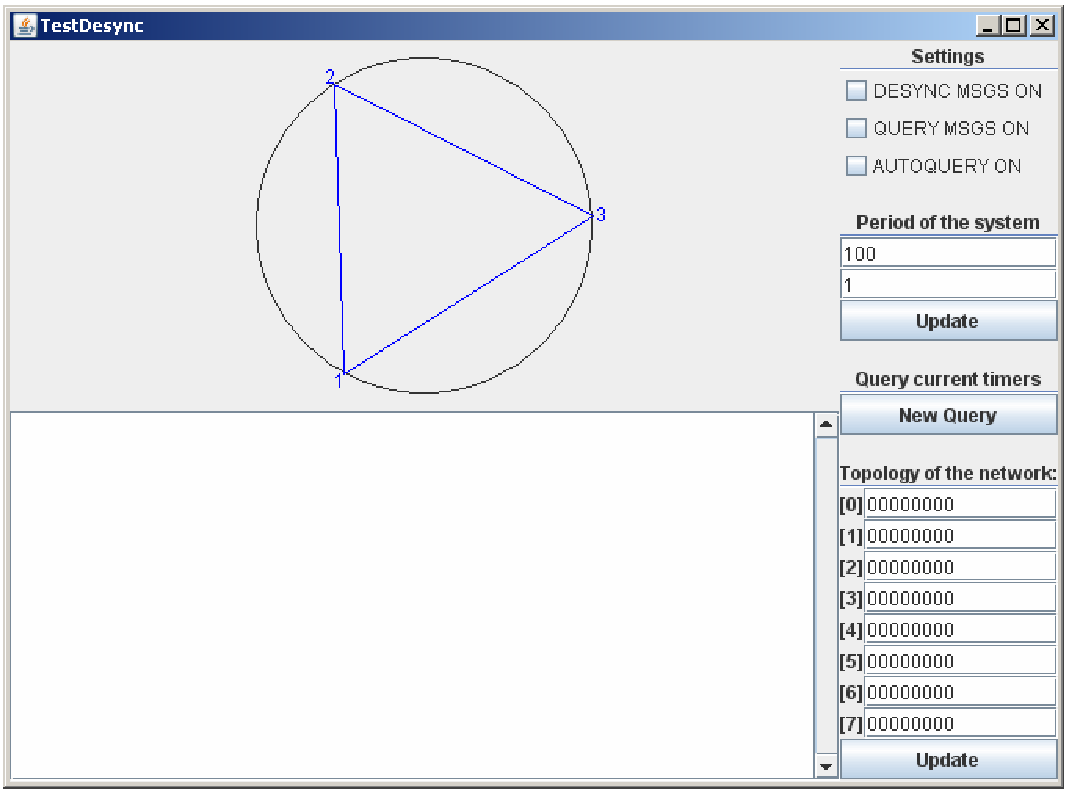
- Estimation of the number of neighbours. For all the calculus present in the algorithm the number of neighbours has to be known. Since in each desync period a node can hear the messages of all its neighbours, the estimation reduces to counting the number of desync packets received. However, since some packets can get lost we chose to smooth the variations of the estimation of the number of neighbours, which in the final version is a fractional number.
- Management of the old information present in each node. Since each node keeps the last information received from all its neighbours, it has to decide what to do when the information turns old, or when it already used

the information to correct its position. The solution chosen involves the progressive attenuation of the old information.

- Compression of the error report to one byte. Internal operations of the nodes related to the time are performed with 32 bits precision. On the other hand we chose to send only one byte indicating the error to the nodes. We had to define a way to map between the two representations.

5.1 Debug and testing

For the debug of the coded protocol we designed a Java application using the infrastructure provided by the Radiowrapper and the TestDesync available modules. The Java application is capable to configure the topology of the network, to send an update period message to one of the nodes and to perform a query of all the internal timers programmed on the nodes.


TestDesync

Settings

DESYNC MSGS ON

QUERY MSGS ON

AUTOQUERY ON

Period of the system

100

1

Query current timers

Topology of the network:

[0]	00000000
[1]	00000000
[2]	00000000
[3]	00000000
[4]	00000000
[5]	00000000
[6]	00000000
[7]	00000000

Figure 7 Interface of the java application used for debug purposes

Test scenarios:

- Original desync algorithm running in eight of the motes simultaneously in a fully connected network. Various period management tests as, for instance, changing the period when a big part of the network is unconnected before the network is merging again.
- Multi-hop algorithm for a star topology with one central node and five neighbours. No perfect solution is possible in this case.
- Ring of 5 nodes.
- Ring of 6 nodes. Perfect solution is found in this case.

6 References

[1] DESYNC: Self Organizing Desynchronization and TDMA on Wireless Sensor Networks
Julius Degesys, Ian Rose, Ankit Patel, Radhika Nagpal
Division of Engineering and Applied Sciences, Harvard University