An Orthodontics-Inspired Desynchronization Algorithm for Wireless Sensor Networks

Pongpakdi Taechalertpaisarn
Department of Computer Engineering
Chulalongkorn University
Bangkok, Thailand
Email: pongpakdi.t@student.chula.ac.th

Supasate Choochaisri
Department of Computer Engineering
Chulalongkorn University
Bangkok, Thailand
Email: supasate.c@student.chula.ac.th

Chalermek Intanagonwiwat
Department of Computer Engineering
Chulalongkorn University
Bangkok, Thailand
Email: chalermek.i@chula.ac.th

***Abstract*—This paper proposes an Orthodontics-inspired desynchronization algorithm for scheduling wireless sensor nodes to avoid conflicts on resource sharing by not accessing the resource at the same time. Applications of desynchronization include TDMA scheduling, wake-sleep scheduling, and collision avoiding. Although existing desynchronization approaches perform reasonably well, their errors highly fluctuate. This high fluctuation implies overshooting and results in high errors even after convergence. For smoother convergence and lower error, we design a mechanism analogous to an orthodontic teeth-alignment technique. In Orthodontics, an orthodontist prevents the already-correct-positioning teeth from moving by tying them with a power chain. We apply the similar concept to the existing desynchronization method to prevent nodes with correct phases from adjusting their time phases.**

We evaluate our implementation of DESYNC-ORT on TOSSIM, a simulator for wireless sensor networks. The simulation result indicates that our method significantly helps the existing desynchronization algorithm to converge smoothly with lower error (reducing approximately 20-60% of errors caused by existing approaches).

I. Introduction

Wireless sensor networks (WSNs) have been deployed for a wide range of applications such as environmental monitoring, industrial monitoring, and target tracking. Several tasks of such applications require sensor nodes *not* to work at the same time (*i.e.* to be desynchronized). For example, nodes in the same wireless collision domain have to eschew simultaneous transmission for successful communication. Nodes in the same sensing area should not be on all the time simply to detect the redundant event. They should take turns to sleep and to wake up at different time for energy savings while maintaining the sensing coverage of the network [1].

In addition to TDMA scheduling and coverage preserving for WSNs, desynchronization can also be applied to other systems such as multi-core computers, analog-to-digital converters (ADC), and automated traffic light systems [2].

A desynchronization framework has been introduced by Degesys et al. [2] (Figure 1). The framework consists of nodes cycling counter clockwise around the time circle or the time period. A node position on a time circle represents a phase of that node. Each node rotates around the perimeter of the circle as time passes. The node accesses the shared resource (*e.g.,* fires a message in a wireless channel, wakes up for sensing)

978-1-61284-307-0/11/$26.00 ©2011 IEEE

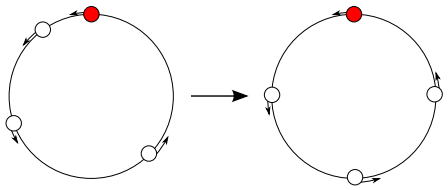


Fig. 1: Desynchronization Framework. Left: starting state. Right: Desynchronized state.

every time when its phase reaches zero. The system reaches the desynchronized state when nodes are equally distributed in the time space.

Degesys et al. have also proposed DESYNC, a simple algorithm to achieve the desynchronized state. Each node continuously adjusts its phase to the middle between its next and previous phase neighbors. By repeating this simple mechanism, nodes will be evenly distributed in the time domain or desynchronized. However, the desynchronization error of DESYNC highly fluctuates and remains rather high even after the algorithm converges due to overshooting.

We observe that, in DESYNC, nodes with correct phases may continuously adjust their phases simply because its phase neighbors are at the improper time position. This behavior is counter-intuitive. Nodes with incorrect phases should be the ones to adjust their time positions whereas nodes with correct phases should not. To reduce the desynchronization error and its fluctuation, we present DESYNC-ORT, an Orthodontics-inspired desynchronization algorithm for WSNs. In our work, phase-neighboring nodes form into a group and stay together if their phase differences are already correct. This mechanism is analogous to orthodontic teeth alignment. An orthodontist ties up the already-correct-positioning teeth with a power-chain in order to maintain the correct teeth gaps (*i.e.,* to prevent them from moving). We apply this same principle to the desynchronization algorithm to ensure that the nodes maintain correct phase differences.

DESYNC-ORT has the following contributions.

- DESYNC-ORT ensures the less-fluctuating error that leads to smoother convergence and lower error.
- DESYNC-ORT is a distributed algorithm that does not require neither global information nor a centralized node. Therefore, this algorithm is scalable since it uses only

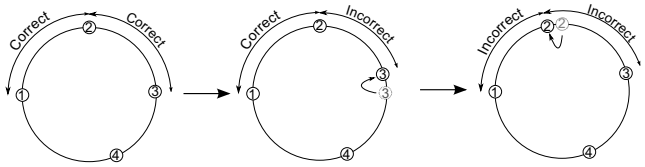


Fig. 2: Wrong adjustment

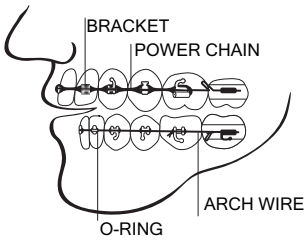


Fig. 3: Orthodontic braces diagram

local information to achieve the desynchronized state.
- DESYNC-ORT is lightweight and simple to implement. It requires no extra message or message payload.
- DSYNC-ORT concept can be applied to other desynchronization algorithms that adapt time phases dynamically.

We have implemented DESYNC-ORT on TinyOS (runnable on real hardware) and evaluated our approach on TOSSIM, a simulator for wireless sensor networks. Our simulation results indicate that DESYNC-ORT significantly outperforms DESYNC by achieving 20-60% reduction in error, especially in dense networks.

The rest of the paper is organized as follows. We explain how orthodontic teeth alignment motivates our work in Section 2 and describe our algorithm in Section 3. We evaluate and compare our work with DESYNC in Section 4. Related works are overviewed and compared with our work in Section 5. Finally, section 6 concludes the paper and discusses limitations as well as future works.

II. Motivation

Our work is motivated by the following two observations. First, we observe that the DESYNC protocol converges with a high error. In addition, even after the protocol converges, the error still fluctuates. This high fluctuation of error is caused by the phase adjustment of the already-correct-positioning nodes. Second, we also observe that the orthodontic teeth arrangement shares a similar idea in some aspects with desynchronization. In Orthodontics, already-correct-positioning teeth are tied up to prevent them from moving. We believe that this technique can reduce fluctuating errors.

In this section, we begin with an overview of DESYNC and explain its pitfall. Then, we describe how to apply an orthodontic method to our desynchronization protocol.

In DESYNC, each node fires a message periodically. After each firing, a node listens to others' firings. The first firing neighbor is selected as the next phase neighbor whereas the last firing neighbor (before the node fires again) is selected as the previous phase neighbor. Then, the node calculates the midpoint ϕ_{mid} between its previous and next phase neighbors as follows:

$$\phi_{mid} = [\phi_{prev} + \phi_{next}]/2, \quad (1)$$

where ϕ_{prev} is the phase of the previous phase neighbor and ϕ_{next} is the phase of the next phase neighbor. The phase is calculated from the firing time modulo the time period T. Finally, the node adjusts its phase to ϕ_{new} by the exponential moving average method as follows:

$$\phi_{new} = (1 - \alpha)\phi_{current} + \alpha\phi_{mid}, \quad (2)$$

where $\phi_{current}$ is the current phase and α is a step size for adjustment.

However, the above adjustment mechanism (based on two phase neighbors of DESYNC) has a pitfall (see Figure 2). On the left figure, node 1, 2, and 3 have correct phase differences between each pair of them, but node 4 is at a wrong position. Later, node 3 adjusts its phase to the midpoint between node 2 and 4 (the middle figure). Then, in the right figure, the node 2 adjusts its phase to the midpoint between node 1 and 3. Consequently, only one wrong node can adversely affects other correct nodes. This pitfall results in the high-fluctuating error.

In order to reduce such an error and its fluctuation, nodes that are in correct positions should stay together whereas the wrong nodes should adjust themselves to the correct positions. This is analogous to orthodontic teeth alignment. An orthodontist uses brackets, arch wires, O-rings, and power chains to align the teeth (Figure 3). Brackets are attached to the teeth. An arch wire links up each bracket and puts pressure on the teeth. O-rings hold arch wires to brackets and help put more pressure. The pressure forces the teeth to align into desirable positions. In order to put more pressure to move teeth into a specific direction or to close gaps between each teeth, an orthodontist uses a rubber band called *power chain* to tie up teeth that are already in desirable positions together.

We can apply the similar principle to a desynchronization algorithm as follows. A node that is already in a correct phase position should be tied up with its neighbor and should not move away. If several consecutive nodes can be tied up as a large group, we put more pressure to them to stay together. To put more pressure, we add one weight unit for each consecutive tying up. The node that is tied up with others has more weight than a single node. This weight lets a node know how correct its position is. A heavy weight of a node means that the node is tied up with many nodes. Therefore, the node implicitly knows that it stays in a correct position with a high probability. If a node is heavy, it should not adjust its phase or should adjust only slightly (see Figure 4).

III. Orthodontics-Inspired protocol

In this section, we present our Orthodontics-inspired desynchronization protocol in details. The core of our algorithm

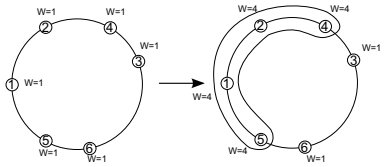


Fig. 4: Nodes with correct phase differences (1, 2, 4, 5) are tied-up and their weights are increased. Light nodes (3, 6) can move whereas heavy nodes attempt to stay together.

Algorithm 1 Initialization

1: $T \leftarrow TimePeriod$ {Configurable Time Period}
2: $PGAP \leftarrow T/NumNeighbors$ {Perfect phase diff}
3: $nextNbrTime, prevNbrTime, lastReceiveTime \leftarrow 0$
4: $lastFiringTime, bckwdWeight, fwdWeight \leftarrow 0$
5: $chaining \leftarrow BACKWARD$
6: call SetFiringTimer(T)

consists of weight determination and phase adjustment mechanisms.

A. Weight Determination

When a node starts, it initializes variables and timers (see Algorithm 1). In order to determine its weight, each node calculates the total number of nodes that can be consecutively tied up with itself. A node counts the number of consecutive nodes before its firing and after its firing.

Given a time period T, a node i can be tied up with its consecutive phase neighbors $n_1, n_2, ..., n_k$ if and only if

$$\forall j \in [1, k] : |\Delta\phi_{i,n_j} - (T/N) * j| \leq \epsilon, \quad (3)$$

The total number of node N is calculated each time the node fire its timer. If the node can not hear a neighbor fire for 3 consecutive time then it will delete that neighbor from the list. Whereby T/N is a perfect phase difference, and ϵ is an allowable error.

We note that the allowable error ϵ is a configurable parameter that depends on a system running this desynchronization mechanism. In our investigated scenario, our system runs this mechanism on the application layer. Thus, we set ϵ to 4 milliseconds to compensate for data delay and MAC access time.

After setting the timer, each node listens to neighbors' firings. During this period, a node counts consecutive tied-up nodes on its backward side. Upon receiving a firing from its neighbor (line 1 in Algorithm 2), the node records the time if the firing is from the next phase neighbor (line 2-5). Then, the node checks whether this firing can be tied up with itself as in eq. 3. However, for the backward side, the node does not know the sequence order of the received firing. Therefore, the node checks that the backward phase difference between itself and the received firing is a multiple of the perfect phase difference with the allowable error . In addition, it also checks that the difference between the current received firing and the last received firing is about the perfect phase difference.

Algorithm 2 Weight Determination Algorithm

1: **Upon receiving a firing message**
2: $receiveTime \leftarrow currentTime$
3: **if** $lastReceiveTime \leq lastFiringTime$ **then**
4: $nextNbrTime \leftarrow receiveTime$
5: **end if**
6: **if** $chaining$ is $BACKWARD$ **then**
7: $diff \leftarrow T - (receiveTime - lastFiringTime)$
8: $neighborDiff \leftarrow receiveTime - lastReceiveTime$
9: **if** $(neighborDiff \leq \epsilon)$ AND $((diff \bmod PGAP \leq \epsilon)$ OR $(PGAP - (diff \bmod PGAP) \leq \epsilon))$ **then**
10: $bckwdWeight \leftarrow bckwdWeight + 1$
11: **else**
12: $bckwdWeight \leftarrow 0$
13: **end if**
14: **else** {$chaining$ is $FORWARD$}
15: $diff \leftarrow receiveTime - lastFiringTime$
16: **if** $|diff - (PGAP * (fwdWeight + 1))| \leq \epsilon$ **then**
17: $fwdWeight \leftarrow fwdWeight + 1$
18: **else** {chain is broken}
19: call **Phase adjustment**
20: **end if**
21: **end if**
22: $lastReceiveTime \leftarrow receiveTime$

23: **Upon firing timer expire**
24: call SendFiringMessage()
25: $lastFiringTime \leftarrow currentTime$
26: $diff \leftarrow currentTime - lastReceiveTime$
27: **if** $|diff - PGAP| \leq \epsilon$ **then**
28: $bckwdWeight \leftarrow bckwdWeight + 1$
29: **else**
30: $bckwdWeight \leftarrow 0$
31: **end if**
32: $prevNbrTime \leftarrow lastReceiveTime$
33: $chaining \leftarrow FORWARD$
34: call SetFiringTimer(T)

If errors are in the allowable range, the node increases its backward weight by 1 (line 6-10).

Otherwise, the node resets its backward weight (line 11-13) since the consecutive chain is broken before reaching itself. Then, the node records the current neighbor's firing as the latest received firing for comparing the phase difference with the next neighbor's firing (line 22). The node repeats this procedure for each received firing until the firing timer expires.

When the timer expires, the node fires a message (line 23-24) and checks whether it can be tied up with the previous phase neighbor (line 25-31). Then, the node sets the timer of the next firing and begins counting consecutive nodes on its other side (line 32-34 and 14-17). The counting process is repeated until the consecutive chain is broken. Then, the node invokes the phase adjustment mechanism (line 18-20).

B. Phase Adjustment Mechanism

After the weight determination process is over, the node invokes the phase adjustment mechanism (line 44) to calculate the next phase to move as follows:

- if the node is tied with both previous and next phase neighbors, it sets α to 0 to stay at the same phase position (line 1-2 in Algorithm 3).
- if the node is the end of the consecutive chain, it sets α to 0.1 (line 3-4). This allows the node to slightly move.
- if the node is not tied up, it sets α to 0.95 as same as in DESYNC (line 5-7).

Then, the node calculates and sets the next firing time as in eq. 1 and 2 (line 8-9) and repeats the weight determination procedure (line 10-11).

Algorithm 3 Phase adjustment algorithm

1: **if** $bckwdWeight \geq 1$ AND $fwdWeight \geq 1$ **then**
2: $\alpha \leftarrow 0$
3: **else if** $bckwdWeight \geq 1$ XOR $fwdWeight \geq 1$ **then**
4: $\alpha \leftarrow 0.1$
5: **else**
6: $\alpha \leftarrow 0.95$
7: **end if**
8: $goalTime \leftarrow (((1-\alpha) * lastFiringTime) + (\alpha * (prevNbrTime + nextNbrTime)/2)) + T$
9: call SetFiringTimer($goalTime - currentTime$)
10: $bckwdWeight, fwdWeight \leftarrow 0$
11: $chaining \leftarrow BACKWARD$

C. Algorithms Complexity

According to the pseudo code, the running time of our algorithm in weight determination and phase adjustment is *O(1)*. Given that the algorithm is executed once per received message from a neighbor, we conclude that the algorithm complexity is *O(N)* per period where N is the number of neighbors.

IV. Evaluation

In this section, we evaluate our algorithm (DESYNC-ORT) by comparing with DESYNC to measure the impact of our algorithm on an existing protocol. The performance metrics of this experiment are converging time and desynchronization error. The former indicates how fast an algorithm can converge. The latter indicates how close the system is to the perfect desynchronized state. The fast algorithm that incurs low errors is preferred.

A. Evaluation Environment

We implement DESYNC-ORT (an integration of our mechanism with DESYNC) on TinyOS (an operating system for WSNs) so that our code can run on TOSSIM (a TinyOS simulator) as well as real hardware. We conduct our evaluation on TOSSIM and compare the performance of DESYNC-ORT

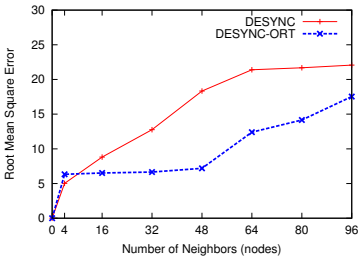


Fig. 5: Root mean square error after 300 time periods

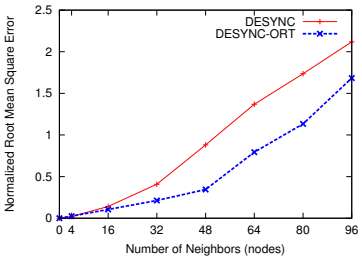


Fig. 6: Root mean square error normalized by perfect phase difference after 300 time periods

with that of DESYNC. We test both algorithms on single-hop networks of 4-96 nodes because DESYNC is originally designed to work in this environment. It is possible to extend DESYNC to work in multi-hop networks and to integrate our mechanism on the extended DESYNC. However, such an extension is out of scope for this paper.

Each node fires a message that consists of node ID and a sequence number every time period. We set the time period (T) to 1,000 milliseconds and all nodes randomly start within the range of 0-1000 milliseconds. The step size (α) of DESYNC is 0.95 (the same value used in [2]).

B. Desynchronization Error

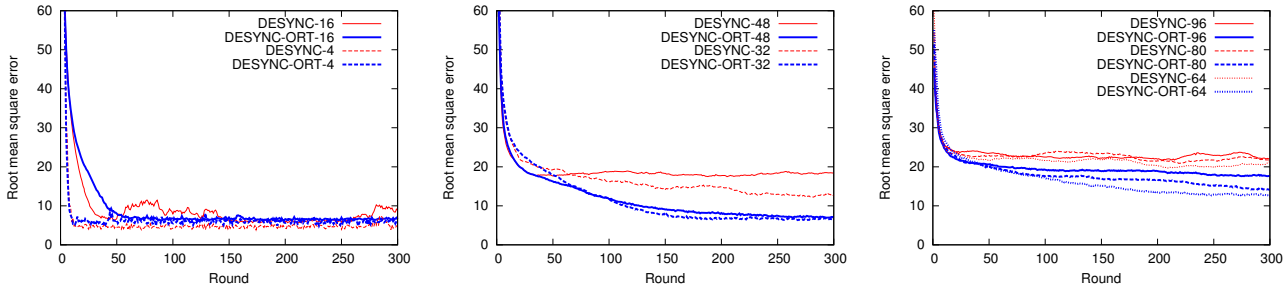
In order to obtain the convincing simulation result, we simulate each algorithm 30 times per one network size and average the results. We run each simulation for 300 time periods (T) and calculate the desynchronization error of each approach. The desynchronization error is the average root mean square error (RMSE) shown in the equations below.

$$ERR_i = \Delta\phi_{ij} - T/N, \tag{4}$$

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} ERR_i^2}{N}}. \tag{5}$$

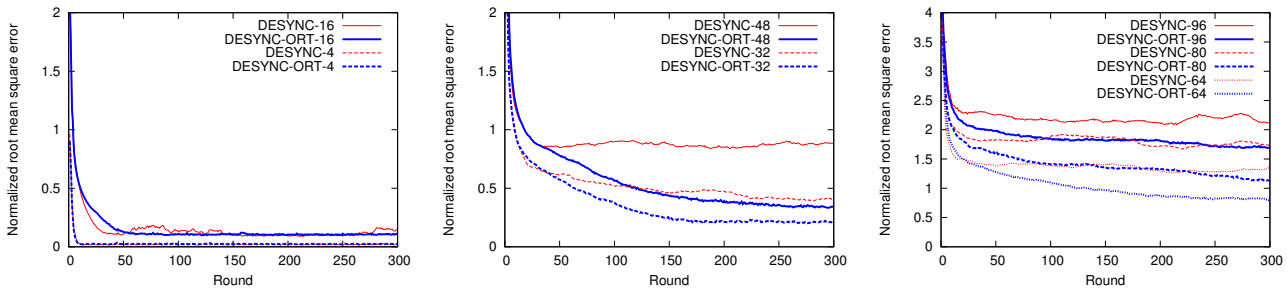
The error (ERR) is the measured phase difference minus by the perfect phase difference whereby node j is the next phase neighbor of node i. $\Delta\phi_{ij}$ is the actual phase difference between node i and node j on the time period T. Given that N is the total number of nodes, T/N is the perfect phase difference.

Our simulation result after 300 time periods (Figure 5) indicates that DESYNC-ORT significantly outperforms DESYNC as expected. However, the absolute phase error can not be used to compare the performance of algorithms since the



(a) Sparse (b) Dense (c) Extremely Dense

Fig. 7: Convergence time and root mean square error



(a) Sparse (b) Dense (c) Extremely Dense

Fig. 8: Convergence time and root mean square error normalized by expected phase difference

perfect phase difference is not the same for each network size. We normalize the absolute phase error by the perfect phase difference (T/N) to obtain the comparable view of our result (see Figure 6).

Our result indicates that our approach significantly improve the performance of DESYNC from 20-60%, especially in the dense network scenarios. In the sparse networks, both protocols converge very fast since the error of only phase neighbors will be partially eliminated per round. Thus, the system with fewer nodes converges faster. In the dense networks, DESYNC-ORT converges with less-fluctuating errors and achieves up to 60% reduction in desynchronization errors. In extremely dense networks, even though both protocols suffer from the high message loss , DESYNC-ORT still converges with a lower bound of errors.

In DESYNC-ORT, the lessen step size α of nodes in the group prevents the nodes from over-adjusting their phases. Additionally, an error from a phase neighbor cannot easily propagate to a node in the group any more. In DESYNC, a part of this error will propagate back and forth between two phase neighbors as well as circulate inside the network. As a result, DESYNC's error after convergence is still quite large. DESYNC-ORT successfully prevents this from happening as evident in less-fluctuating RMSE.

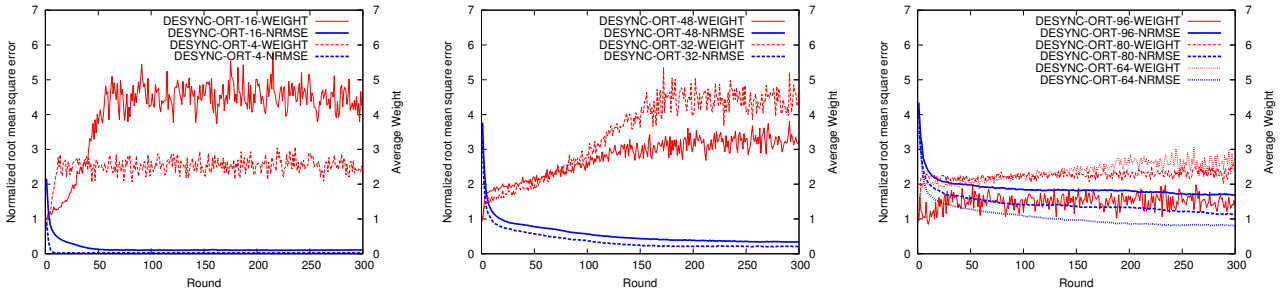
C. Convergence Time

The above result only indicates the error after 300 time periods of adjustment. The result does not indicate whether the protocols converge or not. Neither does it indicate how fast they converge. Hence, we measure the root mean square error (RMSE) and normalized RMSE for each time period to observe the convergence property and time in this experiment.

In sparse networks (Figure 7a and 8a), both protocols converge with the similar small errors and the same rate of convergence but DESYNC-ORT converges with less-fluctuating errors. In dense networks (Figure 7b and 8b), DESYNC-ORT significantly outperforms DESYNC up to 60%. Even though DESYNC's error is stabilized earlier, DESYNC-ORT's error at that time is already lower. In extremely dense networks (Figure 7c and 8c), both protocols converge with a slower rate due to message collision and loss.

D. Correlation of Weight and Desynchronization Error

In this section, we investigate the relationship between a weight and a desynchronization error. For the purpose of comparison among different network sizes, we use NRMSE (normalized root means square error) to represent the desynchronization error. In sparse networks, the average weight is rapidly increased and stabilized near the network size when our protocol converges. However, in dense networks, the average weight is gradually increased and stabilized just below 5. This indicates that the convergent state is not the perfect desynchronized state even with our approach. We notice that the higher weight usually implies the lower error. In extremely dense networks, the average weight increases only slightly and fluctuates throughout the simulation. This is due to high packet collisions and losses in such environments. The stabilized weight below 3 implies the difficulty in grouping the nodes. As a result, the desynchronization error is quite high when compared with that of sparser networks.



(a) Sparse (b) Dense (c) Extremely Dense

Fig. 9: Correlation of average weight and desynchronization error

V. Related Work

Our work has been motivated by previous research efforts described in this section. The most cited paper on desynchronization is DESYNC due to its originality and simplicity. In DESYNC, each node adjusts its phase to the middle between two phase neighbors. By repeating this simple algorithm, nodes will be equally spread out in the time circle. Using only local information from phase neighbors for phase adjustment is the strength of DESYNC. However, this is also its weakness (see Section II). The correct-positioning node may adjust its phase only because its phase neighbor is wrong. This results in high errors and fluctuations even after convergence. Our work addresses this weakness by grouping the correct nodes together and preventing them from adjusting their phases.

Inspired by the firefly synchronization work of Mirollo and Strogatz [3], INVERSE-MS [4] allows all nodes to adjust their phases once they hear a firing message from any node. INVERSE-MS converges rather fast but at the expense of time-period distortion. As a result, INVERSE-MS only provides weak desynchronization whereas DESYNC and our work support strong desynchronization.

Based on the concept of an artificial force fields, DWARF [5] achieves significantly less error than DESYNC by comparing phase difference with all neighbors to calculate proper phase adjustment. Degesys and Nagpal [6] also extend DESYNC to support multi-hop networks by reducing the desynchronization problem in to graph-coloring problem. Recently, several multi-hop desynchronization protocols have been proposed. These include M-DESYNC [7] and Lightweight Coloring protocol [8]. However, none of the above protocols possesses a mechanism to prevent fluctuation errors. We believe that our approach can also be adapted to suit these desynchronization algorithm as well. We plan to incorporate our Orthodontics-inspired method with these protocols in our future work.

VI. Conclusion and Discussion

In this paper, we present DESYNC-ORT, an Orthodontics-inspired desynchronization technique for wireless networks. DESYNC-ORT is completely distributed and localized. No additional message overhead is required. Our technique is simple, intuitive, and effective in providing smoother convergence as well as incurring lower errors. We implement DESYNC-ORT on TinyOS (runnable on real hardware) and conduct our experiment on TOSSIM for performance evaluation. Our result indicates that our technique can significantly improve the performance of DESYNC by lowering error fluctuation and achieving up to 60% reduction in desynchronization errors.

We believe that our technique is sufficiently general and undoubtedly applicable to other desynchronization protocols. Furthermore, we currently use weights only to determine whether nodes should adjust their phases or not. In our future work, we plan to investigate other variations of this scheme as well as integrate this concept with other desynchronization techniques to prove its generality.

VII. Acknowledgement

This work was supported by grant CU CP Academic Excellence Scholarship from Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.

References

[1] S. Jenjaturong and C. Intanagonwiwat, "A set cover-based density control algorithm for sensing coverage problems in wireless sensor networks," in *Cognitive Radio Oriented Wireless Networks and Communications, 2008. CrownCom 2008. 3rd International Conference on*, may 2008, pp. 1 –6.

[2] J. Degesys, I. Rose, A. Patel, and R. Nagpal, "DESYNC: Self-Organizing Desynchronization and TDMA on Wireless Sensor Networks," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, 2007, pp. 11–20.

[3] R. E. Mirollo and S. H. Strogatz, "Synchronization of pulse-coupled biological oscillators," *SIAM J. Appl. Math.*, vol. 50, pp. 1645–1662, November 1990.

[4] A. Patel, J. Degesys, and R. Nagpal, "Desynchronization: The Theory of Self-Organizing Algorithms for Round-Robin Scheduling," in *Self-Adaptive and Self-Organizing Systems, 2007. SASO '07. First International Conference on*, 2007, pp. 87–96.

[5] S. Choochaisri, K. Apicharttrisorn, K. Korprasertthaworn, and C. Intanagonwiwat, "Desynchronization with an artificial force field for wireless networks.", (under submission).

[6] J. Degesys and R. Nagpal, "Towards Desynchronization of Multi-hop Topologies," in *Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Second IEEE International Conference on*, 2008, pp. 129–138.

[7] H. Kang and J. Wong, "A localized multi-hop desynchronization algorithm for wireless sensor networks," in *INFOCOM 2009, IEEE*, 2009, pp. 2906 –2910.

[8] A. Motskin, T. Roughgarden, P. Skraba, and L. Guibas, "Lightweight Coloring and Desynchronization for Networks," in *INFOCOM 2009, IEEE*, 2009, pp. 2383–2391.