

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

**Towards Secure and Self-stabilizing
Sensor Network Services
for Civil Security**

ANDREAS LARSSON

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2011

Towards Secure and Self-stabilizing Sensor Network Services for Civil Security

Andreas Larsson

Copyright © Andreas Larsson, 2011.

Technical report 78L

ISSN 1652-876X

Department of Computer Science and Engineering

Distributed Computing and Systems Research Group

Division of Networks and Systems

Chalmers University of Technology

SE-412 96 GÖTEBORG, Sweden

Phone: +46 (0)31-772 10 00

Author e-mail: larandr@chalmers.se

Printed by Chalmers Reproservice

Göteborg, Sweden 2011

Towards Secure and Self-stabilizing Sensor Network Services for Civil Security

Andreas Larsson

Division of Networks and Systems, Chalmers University of Technology

ABSTRACT

Wireless sensor networks, consisting of a vast number of small sensor nodes that can monitor large areas, are a promising field with many possible applications in many different application areas. In civil security settings, e.g., they can be of great help by monitoring things like disaster areas, restricted areas, crowds and structural integrity. The sensor nodes are deployed in an area that is to be monitored in some way. Typically the nodes do not have any preexisting information about network topology and instead communicate wirelessly to organize the network.

Sensor nodes are often very limited in computing power, memory and battery life. In addition the traffic patterns are generally different than for other types of networks. Therefore, algorithms often need to be tailor-made for sensor networks. Furthermore, for networks that consists of a very large amount of nodes, algorithms have to scale well. Security and fault tolerance is of high importance for many sensor network applications and for civil security in particular. The sensor network application needs to remain functioning even when nodes fails or are attacked in different ways. Sensor nodes often reside in harsh environments that can destroy them, during or after deployment. One potent form of fault tolerance is Self-stabilization. A self-stabilizing system can recover from an arbitrary state within a finite amount of time. Security in wireless sensor networks is further complicated by the fact that the nodes often are physically available for attackers to destroy, capture or manipulate in other ways. The threat of compromised nodes inside the network that are controlled by an attacker is a concern that needs to be taken into account.

High precision synchronized clocks are a fundamental need of many applications and of other services. We present the first secure and self-stabilizing

algorithm for sensor networks that is resilient towards attacks both from the outside and by compromised nodes from the inside. Sensor nodes also needs to organize their own network. A common way is to cluster nodes together into groups. They are used by many applications and other fundamental services. We present a secure and self-stabilizing algorithm for clustering. It uses redundant paths to be resilient against captured nodes in the network.

Keywords: Secure and Resilient Computer Systems, Sensor-Network Systems, Ad-hoc Networks, Clock-synchronization, Clustering, Self-Stabilization

Acknowledgments

I dedicate this thesis to Dag Mathiesen, my late father in law. His curiosity, his constant stream of new ideas and his open mind are sorely missed.

I would like to start with thanking my supervisor Philippas Tsigas for his guidance and support. This thesis would not have been possible without him. His insights into all matters has helped me greatly. No matter the hurdle, he has got some idea on how to proceed.

I am honored to have Oliver Theel from the University of Oldenburg as my discussion leader.

I thank Elad Michael Schiller for the collaboration, discussions and his never-ending support in all matters. I thank Jaap-Henk Hoepman for the collaboration and discussions. I thank Daniel Cederman for discussions, support and fun times. I want to give my appreciation to the rest of the members of the Distributed Computing and Systems group: Farnaz Moradi, Georgios Georgiadis, Marina Papatriantafidou, Nhan Nguyen Dang and Zhang Fu; and to the former members Anders Gidenstam, Boris Koldehofe, Håkan Sundell, Niklas Elmqvist, Phuong Hoai Ha and Yi Zhang. All the times spent with you guys have been great!

I would like to give my appreciation to the people, outside of the Distributed Computing and Systems group, that are involved or previously were involved in the Security Arena project: Anna Gryszkiewicz, Erland Jonsson, Fang Chen, Laleh Pirzadeh, Magnus Almgren, Morten Fjeld, Staffan Björk, Tomas Olovsson, Ulf Larson and Vilhelm Verendel. I thank Swedish Civil Contingencies

Agency for funding and making the Security Arena project a possibility. I thank all the other participants in the Security Arena project as well for cooperations, the interesting seminars and the stimulating environment. I also would like to thank all the master students I have had the joy of working with and in particular Afshan Samani and Carlos Aleixandre Tudó.

I like to thank all the people I have had fun teaching together with and everyone else at the department for all the great fun and the stimulating environment. This very much includes the administrative staff that are always willing and able to help or point in the right direction.

I sincerely thank all my friends for all the fun and all crazy ideas and projects that takes thoughts off research and charges the mental batteries. Thank you Björn, Henrik, Per, Perjohan, Peter, Rikard, all the gaming friends, the Sinus gang, the Chalmers study gang and all the other good friends from growing up and living in Västerås and living in Göteborg.

Last, but certainly not least, I cannot thank my family enough for their endless love and support and all the fun and sad things I have shared with them. I thank my wife Malin for all the love, the fun, the gumption and for always being there for me. I thank my daughter Elvira for her love, her endless energy and curiosity, and for just being. I thank my father Leif-Göran for the love, the great inspiration and having solutions to any practical problem. I thank my mother Anne-Marie for the love, the care and laying the ground for so many of my values. I thank my brother Kristian for all great fun, all crazy ideas and always having a welcoming couch. I also thank Ingrid, Ernst, Roland, Märta and all my other relatives for good times together, and Rose-Marie and the rest of my family in law for taking me to heart and for more good times.

Andreas Larsson
Göteborg, 2011

List of Appended Papers

- I Jaap-Henk Hoepman, Andreas Larsson, Elad M. Schiller, Philippas Tsigas, “Secure and Self-stabilizing Clock Synchronization in Sensor Networks,” in *Theoretical Computer Science* (2010), doi:10.1016/j.tcs.2010.04.012. A preliminary version of this work has also been published in the proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS). November 2007, Paris, France, LNCS 4838 pp. 340-356.

- II Andreas Larsson, Philippas Tsigas, “Self-stabilizing (k,r)-Clustering in Wireless Ad-hoc Networks with Multiple Paths,” Technical Report no. 2010:06, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2010. This work will appear in the proceedings of The 31st International Conference on Distributed Computing Systems (ICDCS). June 2011, Minneapolis, Minnesota, USA. A preliminary version of this article appeared in the proceedings of 14th International Conference On Principles Of Distributed Systems (OPODIS). December 2010, Tozeur, Tunisia, LNCS 6490 pp. 79-82.

Contents

Abstract	i
Acknowledgments	iii
List of Appended Papers	v
I INTRODUCTION	1
1 Introduction	3
1.1 Background	3
1.1.1 Wireless Sensor Networks and Ad Hoc Networks	3
1.1.2 Security in Wireless Ad Hoc and Sensor Networks	5
1.1.3 Self-stabilization	6
1.2 Our Approach	7
1.3 Contributions	8
1.3.1 Paper I	8
1.3.2 Paper II	9
1.4 Conclusions & Future Work	11
Bibliography	12
II PAPERS	15
2 PAPER I	19

2.1	Introduction	20
2.1.1	Our Contribution	23
2.1.2	Document structure	24
2.2	System Settings	24
2.2.1	Time, Clocks, and Their Notation	24
2.2.2	Communications	25
2.2.3	The Adversary	28
2.3	Secure and Self-Stabilizing Clock Synchronization	31
2.3.1	Beacon and Response Aggregation	32
2.3.2	The Algorithm's Pseudo-code	34
2.4	Execution System Model	35
2.4.1	The Interleaving Model	35
2.4.2	Tracing Timestamps and Communications	38
2.4.3	Concurrent vs. Independent Broadcasts	39
2.4.4	Fair Executions	39
2.4.5	The Task	39
2.5	Correctness	40
2.5.1	Scenarios in which balls are thrown into bins	41
2.5.2	The task of random broadcast scheduling	49
2.5.3	Nice executions	52
2.6	Performances of the algorithm	54
2.6.1	Optimizations	55
2.7	Discussion	55
2.7.1	Conclusions	58
2.7.2	Acknowledgments	58
	Bibliography	59
3	PAPER II: Self-stabilizing (k,r)-Clustering in Wireless Ad-hoc Networks with Multiple Paths	65
3.1	Introduction	66
3.1.1	Our Contribution	68
3.2	System Settings	69

3.3	Self-stabilizing (k, r) -clustering algorithm	69
3.4	Correctness	72
3.4.1	Getting enough cluster heads	73
3.4.2	Convergence to a local minimum	79
3.5	Discussion	85
3.5.1	Conclusions	88
	Bibliography	88

List of Figures

2.1	Constants, variables and external functions	36
2.2	Macros and inlines	37
2.3	Secure and self-stabilizing native clock sampling algorithm . .	38
3.1	Constants, variables, external functions and macros	70
3.2	Pseudocode for the clustering algorithm	71
3.3	Simulation results	86

Part I

INTRODUCTION

1

Introduction

1.1 Background

1.1.1 Wireless Sensor Networks and Ad Hoc Networks

A wireless sensor network is a network of small computers, sensor nodes, that can gather information via its sensors, do computations and communicate wirelessly with other sensor nodes. In general a wireless sensor network is an ad hoc network in which the nodes organize themselves without any preexisting infrastructure. Nodes could be deployed randomly, e.g., by being thrown out from an helicopter over an area that is to be monitored. Once in the area, the nodes that survived the deployment procedure communicate with the other nodes that happened to end up in its vicinity, and they set up an infrastructure.

There are many application areas for sensor networks. The possibilities spans areas as civil security, health care, agriculture, research, environmental, commercial and military applications [1, 2]. There are many things in these areas that a sensor network can monitor, e.g., disaster areas, restricted areas, wildlife, crowds, manufacturing machinery, structural integrity, earthquakes, agriculture, traffic, pollution or even heart rates.

The sensor nodes in a sensor network is often small and quite cheap. They can therefore be used in great numbers over a large area. This can provide fault tolerance, in which the system can withstand loss of sensor nodes without loosing coverage of the monitored area or loosing functionality of the network. In addition, compared to more centralized long range sensors, such a sensor network can give a high number of more precise local readings over large areas. The areas monitored can be chosen in according to needs and can change over time [3]. The possibility of rapid deployment is of high value for civil security applications, e.g. monitoring disaster areas.

Sensor nodes, in contrast to computers in general ad hoc networks, are often very limited in computing power and memory capacity. As an example, the popular MICAz sensor node has a 16 MHz processor and only 4 kB of RAM memory and 128 kB of program memory [4]. These limitations restricts the algorithms that feasibly can be used.

Furthermore, the nodes typically run on battery power and communicating is usually the most expensive activity of a sensor node. A MICAz node in receive mode uses around 20mA [5], which would empty 1000mAh batteries in just 50 hours. The corresponding lifetime for an idle node that does not communicate or sense could be several years. Thus, it is important in sensor networks to be conservative in communication.

A sensor network often consists of a large number of nodes. Furthermore, nodes eventually run out of batteries and new nodes are deployed to maintain the network. Therefore, even if the nodes are immobile, the network topology changes over time. Thus, algorithms both have to scale well [6] and need to cope with topology changes.

1.1.2 Security in Wireless Ad Hoc and Sensor Networks

Security is critical for many applications of sensor networks, just as for applications in other kinds of networks. Confidentiality and privacy is needed for sensitive, classified or proprietary information, e.g. medical data, sensitive information in civil security, industrial secrets or military information. It is important to be able to withstand attacks that aims to degrade the functionality of the network. Any kind of application can come under attack from someone that wants to disturb the network. For some applications it is critical to keep as much functionality as possible during an attack. Applications, e.g., that monitors restricted areas might have active attackers that have an interest in making the sensor network report erroneous information and the sensor network plays a critical role in maintaining security and/or safety of the facility. Wireless sensor networks comes with additional security challenges that needs to be addressed and researched [7–10].

Sensor networks are deployed in areas that is to be monitored. This usually implies that they are physically available for attackers. Furthermore, to feasibly deploy large number of nodes, they need to be inexpensive. Tamper-proof nodes are therefore often out of the question. The limitations in computing power, memory and battery makes many security algorithms inappropriate for use in sensor networks [11]. This also limits the cryptography possibilities, especially for public key cryptography. Sensor networks often have very different traffic patterns than other networks. Information usually flows between the sensor nodes and the base station, or between nodes close to each other, but not between any pair of nodes in general. In addition, information is often aggregated on the way to decrease the total amount of needed traffic. The wireless medium makes it easy for an attacker to eavesdrop on the traffic, to jam communication or to inject messages into the network. This combination of circumstances that holds for many sensor networks opens up a set of security issues that needs to be considered. It also means that security protocols that are used in other networks, e.g. the Internet, are often not suitable at all for the sensor network setting.

The physical access to nodes, the environment and the open communication medium makes security for sensor networks especially tricky. There are many ways an attacker can use this to attack the network [12]. The attacker could place own sensor nodes in the area that is used to disturb or infiltrate the network. The attacker can capture and reprogram nodes that are part of the network. A much stronger node, e.g. a laptop, can be used to infiltrate and attack the network either as a new node or to replaced a captured node after extracting secret information, like cryptographic keys. To have malicious nodes like this inside the network, *compromised nodes*, is a challenge to deal with and is an important area for research. Compromised nodes can do a lot of damage to the network. They can steal encrypted information, they can report erroneous information and they can degrade routing in the network. They can behave in arbitrary ways and break protocols that are not resilient to misbehavior. If countermeasures against misbehaving nodes are taken, they can report innocent nodes as misbehaving.

Security is not something that can be added to an insecure system to be able to withstand attacks. Security needs to be part of most protocols and algorithms in the system. Otherwise the attacked can chose to attack the unsecured parts. Therefore, it is important to have secure algorithms for all the basic services that are needed in sensor networks.

1.1.3 Self-stabilization

Self-stabilizing algorithms [13–15] cope with the occurrence of transient faults in an elegant way. Starting from an arbitrary state, self stabilizing algorithms let a system stabilize to and stay in a consistent state as long as the algorithms' assumptions hold for a sufficiently long period.

There are many reasons why a system could end up in an inconsistent state of some kind. Assumptions that algorithms rely on could temporarily be invalid. Memory content could be changed by radiation or other elements of harsh environments. Messages could temporarily get lost to a much higher degree than anticipated. Topology changes happens when nodes eventually run out of mem-

ory, if they get physically destroyed in harsh environments or when new nodes are added to the network to maintain coverage. Such topology changes could break assumptions and lead to temporary inconsistencies. It is often not feasible to manually reconfigure large ad hoc networks to recover from events like this. Self-stabilization is therefore often a desirable property of algorithms for ad hoc networks and especially for sensor networks [16].

In the sensor network setting assumptions about the system could eventually be violated when an attacker, far more powerful than the limited sensor nodes, starts disturbing the sensor network. It is hard to anticipate all possible states the network could end up in after an attack. Large number of nodes could get compromised and send incorrect information, nodes could be physically attacked in different ways or the attacked might jam the communication medium. Self-stabilization makes sure that the network can recover from any state as long as assumptions hold once again, e.g., after the attacker has been chased away or more nodes have been added to the network.

Thus, in civil security settings it is very useful for disaster areas where the harsh environment might destroy or corrupt the nodes and also in situations like intrusion detection scenarios where attackers have interest in disturbing the network.

Just like security, fault tolerance is not something that can be added in retrospect to a system. It needs to be an integral part of the system. Therefore it is important that basic services for sensor networks are fault tolerant from the beginning.

1.2 Our Approach

As we have seen above both security and self-stabilization are preferable characteristics for algorithms used in sensor networks in open and unattended environments. We have also seen that compromised nodes can do a lot of damage. For many needs there are either secure or self-stabilizing algorithms, but often not secure and self-stabilizing. Moreover, many algorithms that takes security into account does not take compromised nodes into account. In civil security

settings we can anticipate bursts of events, unpredictable and/or malicious behavior and natural circumstances that destroys nodes. This strengthens the requirement for security and self-stabilization.

Our approach is to provide high level networking protocols for sensor networks and/or ad hoc networks that are both secure and self-stabilizing. We aim for solutions that can withstand both faults and attacks. We saw above that compromised nodes inside the network can mislead other nodes in the network and/or disturb the functionality of the network. This is an important area of research and a serious threat. We take this into account in our research.

So far we have looked at two fundamental network services that are secure and self-stabilizing, clock synchronization and clustering. For many applications it is critical that the nodes have a shared view of time with high precision. For that, clock synchronization protocols are needed. Examples of areas that requires high precision global time include pinpointing and tracking events, e.g. fire propagation and intrusions, scheduling shared radio medium, e.g. using Time Division Multiple Access (TDMA), and detecting duplicate events. Clustering nodes together into groups is a basic need for sensor networks. Sensor networks and other ad hoc networks need to organize themselves after deployment. Clustering sets up a structure that, e.g., can be used for forming backbones, for routing in general, for aggregating data from many nodes to reduce the amount of data that needs to be sent through the network, for building hierarchies that allow for scaling and for nodes to take turns doing energy intensive tasks.

1.3 Contributions

1.3.1 Paper I

As we have seen above, accurate clock synchronization is imperative for many applications in sensor networks, such as mobile object tracking, detection of duplicates, and TDMA radio scheduling. Broadly speaking, existing clock synchronization protocols are too expensive for sensor networks because of the

nature of the hardware and the limited resources that sensor nodes have. The unattended environment, in which sensor nodes typically reside, necessitates secure solutions and autonomous system design criteria that are self-defensive against a malicious adversary.

In the first paper, we propose the first self-stabilizing algorithm for clock synchronization in sensor networks with security concerns. We consider an adversary that captures nodes and intercepts messages that it later replays – a so called *pulse delay attack*. Our algorithm guarantees automatic recovery after the occurrence of arbitrary failures. Moreover, the algorithm tolerates message omission failures that might occur, say, due to the algorithm’s message collisions or due to ambient noise.

The core of our clock synchronization algorithm is a mechanism for sampling the clocks of neighboring nodes in the network. Of especial importance is the sampling of clocks at reception of broadcasts called beacons. A beacon acts as a shared reference point because nodes receive it at approximately the same time (propagation delay is negligible for these radio transmissions).

The algorithm secures, with high probability, sets of complete neighborhood clock samples with a period that is $O((\log n)^2)$ times the optimum. The optimum requires, in the worst case, the communication of at least $O(n^2)$ timestamps. Here n is a bound on the number of sensor nodes that can interfere with a node. Our design tolerates transient failures and self-stabilizes from an arbitrary configuration that could have been created when assumptions did not hold. Once all assumptions hold again, the system will stabilize within one communication timeslot (that is of size $O(n \log n)$).

1.3.2 Paper II

As we have seen, in large sensor networks it is often important for the nodes to organize themselves into some infrastructure. Thus, an algorithm for clustering nodes together in an ad hoc network serves an important role. Back bones for efficient communication can be formed using cluster heads. Clusters can be used for routing messages. Cluster heads can be responsible for aggregating

data, e.g. sensor readings in an ad hoc sensor network, into reports to decrease the number of individual messages needed to rout through the network. Hierarchies of clusters on different levels can be used for improved scaling of a large network. Nodes in a cluster could take turns doing energy costly tasks to save power over all.

One way of clustering nodes in a network is for nodes to associate themselves with one or more cluster heads. In the (k,r) -clustering problem. Each node in the network should have at least k cluster heads within r communication hops away. This might not be possible for all nodes if the number of nodes within r hop from them is smaller than k . In such cases a best effort approach can be taken for getting as close to k cluster heads as possible for those nodes. Assuming that the network allows k cluster heads for each node, the set of cluster heads forms a total (k,r) -dominating set in the network. In a *total* (k,r) -dominating set the nodes in the set also needs to have k nodes in the set within r hops, in contrast to an ordinary (k,r) -dominating set in which this is only required for nodes not in the set.

In this second paper, we present the first self-stabilizing (k, r) -clustering algorithm for ad hoc networks. The algorithm is based on synchronous rounds and makes sure that, within $O(r)$ rounds, all nodes have at least k cluster heads (or all nodes within r hops if a node has less than k nodes within r hops) using a deterministic scheme. A randomized scheme complements the deterministic scheme and lets the set of cluster heads to stabilize to a local minimum, with high probability, within $O(gr \log n)$ rounds, where g is a bound on number of nodes within $2r$ hops, and n is the size of the network. Multiple paths are used to improve security in presence of compromised nodes, to improve availability and fault tolerance.

The communication costs in this algorithm might be too steep for some sensor network nodes. We discuss some simplifications of the network structure to reduce message complexity to make it more suitable for such sensor networks.

1.4 Conclusions & Future Work

In this thesis, we have presented two secure and self-stabilizing high level networking protocols. One for clock synchronization and one for clustering – both crucial needs for many sensor networks and other ad hoc networks.

With all the potential application areas security is going to become more and more important for sensor networks in the future. Mass production of small cheap nodes will open up endless possibilities, but also open up easy venues of attacks. The general physical availability of sensor nodes together with the possibility for an attacker to capture and/or insert controlled nodes implies that it is of utmost importance to defend against insider attacks in the network. Intrusion detection is also something that could be of help in these situations. Together with the possibilities to monitor all kinds of data in all kinds of places comes the importance of privacy, especially in areas like medicine and monitoring of public places.

An interesting future direction is to provide additional fundamental network services for which no present algorithms take both self-stabilization and security into account. Routing is needed in any sensor network application that does not merely store sensor readings locally. Thus, to set up a secure sensor network, secure routing is one such needed service. Another interesting direction is to combine different protocols together into a secure and fault tolerant package for increased efficiency and ease of use.

There are more to be done in terms of analyzing our clustering algorithm and the result of the clustering. We would like to quantitatively measure what security properties we can get from the multiple paths that are provided in the clustering algorithm. We also would like to analyze how close the local minima we achieve, of cluster heads, are to corresponding global minima. The clustering algorithm relies upon synchronous rounds. A good direction for future work is to get rid of that dependency or to combine the clustering with our clock synchronization algorithm to achieve synchronous rounds.

Bibliography

- [1] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar, “Next century challenges: scalable coordination in sensor networks,” in *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, New York, NY, USA, 1999, pp. 263–270, ACM.
- [2] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci, “A survey on sensor networks,” *Communications Magazine, IEEE*, vol. 40, no. 8, pp. 102 – 114, aug. 2002.
- [3] J. Agre and L. Clare, “An integrated architecture for cooperative sensing networks,” *Computer*, vol. 33, no. 5, pp. 106 –108, may. 2000.
- [4] “Atmega128(1),” http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf, 2009.
- [5] “Micaz data sheet,” www.openautomation.net/uploads/productos/micaz_datasheet.pdf.
- [6] Roger Wattenhofer, “Sensor networks: Distributed algorithms reloaded - or revolutions,” in *13th Colloquium on Structural Information and Communication Complexity (SIROCCO)*, United Kingdom, 2006, pp. 24–28.
- [7] X Chen, K. Makki, Kang Yen, and N. Pissinou, “Sensor network security: a survey,” *Communications Surveys Tutorials, IEEE*, vol. 11, no. 2, pp. 52 –73, 2009.
- [8] Yun Zhou, Yuguang Fang, and Yanchao Zhang, “Securing wireless sensor networks: a survey,” *Communications Surveys Tutorials, IEEE*, vol. 10, no. 3, pp. 6 –28, 2008.
- [9] Yong Wang, G. Attebury, and B. Ramamurthy, “A survey of security issues in wireless sensor networks,” *Communications Surveys Tutorials, IEEE*, vol. 8, no. 2, pp. 2 –23, 2006.
- [10] Adrian Perrig, John Stankovic, and David Wagner, “Security in wireless sensor networks,” *Commun. ACM*, vol. 47, no. 6, pp. 53–57, 2004.
- [11] E. Shi and A. Perrig, “Designing secure sensor networks,” *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 38 – 43, dec. 2004.
- [12] Xiangqian Chen, K. Makki, Kang Yen, and N. Pissinou, “Node compromise modeling and its applications in sensor networks,” in *Computers and Communications, 2007. ISCC 2007. 12th IEEE Symposium on*, jul. 2007, pp. 575 –582.

- [13] Edsger W. Dijkstra, “Self-stabilizing systems in spite of distributed control,” *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.
- [14] Shlomi Dolev, *Self-Stabilization*, MIT Press, March 2000.
- [15] Z. Shi and P. K. Srimani, “Self-stabilizing distributed systems & sensor networks,” in *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad-Hoc Wireless, and Peer-to-Peer Networks*, chapter 23, pp. 393–402. Auerbach Publications, 2005.
- [16] Ted Herman, “Models of self-stabilization and sensor networks,” in *Distributed Computing - IWDC 2003*, Samir R. Das and Sajal K. Das, Eds., vol. 2918 of *Lecture Notes in Computer Science*, pp. 836–836. Springer Berlin / Heidelberg, 2004.

Part II

PAPERS

PAPER I

Jaap-Henk Hoepman, Andreas Larsson, Elad M. Schiller, Philippas Tsigas

Secure and Self-stabilizing Clock Synchronization in Sensor Networks

Theoretical Computer Science (2010), doi:10.1016/j.tcs.2010.04.012

In order to fit the thesis layout, some small non-technical changes has been done and a figure has been split into two.

A preliminary version of this work has also been published in the proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS). November 2007, Paris, France, LNCS 4838 pp. 340-356.

2

PAPER I

In sensor networks, correct clocks have arbitrary starting offsets and nondeterministic fluctuating skews. We consider an adversary that aims at tampering with the clock synchronization by intercepting messages, replaying intercepted messages (after the adversary's choice of delay), and capturing nodes (i.e., revealing their secret keys and impersonating them). We present an efficient clock sampling algorithm which tolerates attacks by this adversary, collisions, a bounded amount of losses due to ambient noise, and a bounded number of captured nodes that can jam, intercept, and send fake messages. The algorithm is self-stabilizing, so if these bounds are temporarily violated, the system can efficiently stabilize back to a correct state. Using this clock sampling algorithm, we construct the first self-stabilizing algorithm for secure clock synchronization in sensor networks that is resilient to the aforementioned adversarial attacks.

2.1 Introduction

Accurate clock synchronization is imperative for many applications in sensor networks, such as mobile object tracking, detection of duplicates, and TDMA radio scheduling. Broadly speaking, existing clock synchronization protocols are too expensive for sensor networks because of the nature of the hardware and the limited resources that sensor nodes have. The unattended environment, in which sensor nodes typically reside, necessitates secure solutions and autonomous system design criteria that are self-defensive against a malicious adversary.

To illustrate an example of clock synchronization importance, consider a mobile object tracking application that monitors objects that pass through the network area (see [1]). Nodes detect the passing objects, record the time of detection, and send the estimated trajectory. Inaccurate clock synchronization would result in an estimated trajectory that could differ significantly from the actual one.

We propose the first self-stabilizing algorithm for clock synchronization in sensor networks with security concerns. We consider an adversary that captures nodes and intercepts messages that it later replays. Our algorithm guarantees automatic recovery after the occurrence of arbitrary failures. Moreover, the algorithm tolerates message omission failures that might occur, say, due to the algorithm's message collisions or due to ambient noise.

The core of our clock synchronization algorithm is a mechanism for sampling the clocks of neighboring nodes in the network. Of especial importance is the sampling of clocks at reception of broadcasts called beacons. A beacon acts as a shared reference point because nodes receive it at approximately the same time (propagation delay is negligible for these radio transmissions). Elson et al. [2] use such samples to approximate the clocks of neighboring nodes. They use linear regression to deal with differences in clock rates. The basic algorithm synchronizes a cluster. Overlapping clusters with shared gateway nodes can be used to convert timestamps among clusters. Karp et al. [3, 4] input clock samples of beacon receipts into an iterative algorithm, based on resistance networks,

to converge to an estimated global time. Römer et al. [5] give an overview of methods that use samples from other nodes to approximate their clocks. They present phase-locked looping (PLL) as an alternative to linear regression and present methods for estimating lower and upper bounds of neighbors' clocks. Note that none of these articles takes security or self-stabilization into account.

As mentioned above, the short propagation delay of messages in close range wireless communications allows nodes to use broadcast transmissions to approximate pulses that mark the time of real physical events (i.e., beacon messages). In the *pulse-delay* attack, the adversary snoops messages, jams the synchronization pulses, and replays them at the adversary's choice of time (see [6–8] and Section 2.2.3). We are interested in fine-grained clock synchronization, where there are no cryptographic countermeasures for such pulse-delay attacks. For example, the *nonce* techniques strive to verify the freshness of a message by issuing pseudo-random numbers for ensuring that old communications could not be reused in replay attacks (see [9]). Unfortunately, the lack of fine-grained clock synchronization implies that the round-trip time of message exchange cannot be efficiently estimated. Therefore, it is not clear how the nonce technique could detect pulse-delay attacks.

The system strives to synchronize its clocks while forever monitoring the adversary. We assume that the adversary cannot break existing cryptographic primitives for sensor networks by eavesdropping (e.g., [9, 10]). However, we assume that the adversary can *capture* nodes, reveal their entire state (including private variables), stop their execution, and impersonate them. The adversary can also lead them to send erroneous information and launch jamming (or collision) attacks.

We assume that, at any time, the adversary has a distinct location in space and a bounded influence radius, uses omnidirectional broadcasts from that distinct location, and cannot intercept broadcasts for an arbitrarily long period. (Namely, we consider system settings that are comparable to the settings of Gilbert et al. [11], which consider the minimal requirements for message delivery under broadcast interception attacks.) We explain how to sift out responses

to delayed beacons by following the above assumptions that consider many practical issues.

A secure synchronization protocol should mask attacks by an adversary that aims to make the protocol give an erroneous output. Unfortunately, due to the unattended environment and the limited resources, it is unlikely that all the designer's assumptions hold forever. We consider systems that have the capability of monitoring the adversary, and then stopping it by external intervention. In this case, the nodes start executing their program from an arbitrary state. From that point on, we require rapid system recovery. Self-stabilizing algorithms [12, 13] cope with the occurrence of transient faults in an elegant way. Bad configurations might occur due to the occurrence of an arbitrary combination of failures. Self-stabilizing systems can be started in *any* configuration. From that arbitrary starting point, the algorithm must ensure that it accomplishes its task if the system obeys the designer's assumptions for a sufficiently long period.

We focus on the fault-tolerance aspects of secure clock synchronization protocols in sensor networks. Uncaptured nodes behave correctly at all times. Furthermore, the communication model is fair. It resembles that of [14] and does not consider Byzantine behavior in the communication medium. However, captured nodes can behave in a Byzantine manner at the processor level. We design a distributed algorithm for sampling the clocks of g neighboring nodes in the presence of f captured and/or pulse-delay attacked nodes. Although captured nodes remain captured, a node whose pulse-delay attacked messages are no longer in the buffer of any uncaptured node will not count toward f anymore. We focus on captured nodes and delay attacks, but f can be extended to include nodes with timing failures and other ways of not following protocol.

The clock sampling algorithm facilitates clock synchronization using a variety of existing masking techniques to overcome pulse-delay attacks in the presence of captured nodes. For example, [7] uses Byzantine agreement (this requires $3f + 1 \leq g$), and [8] considers the statistical outliers (this requires $2f + O(1) \leq g$). (See Section 2.7 for details on the masking techniques.)

Although Byzantine agreement is one possible filtering technique, we do not consider Byzantine faults, as stated above.

The execution of a clock synchronization protocol can be classified between two extremes: *on demand* and *continuous*. Nodes that wish to synchronize their clocks can invoke a distributed procedure for clock synchronization on demand. The procedure terminates as soon as the nodes reach their target precision. An execution of a clock synchronization program is classified as continuous if no node ever stops invoking the clock synchronization procedure. Our generic design facilitates a trade-off between energy conservation (i.e., on-demand operation) and fine-grained clock synchronization (i.e., continuous operation). The trade-off allows budget policies to balance between application requirements and energy constraints (more details appear in [15]).

2.1.1 Our Contribution

We present the first design for secure and self-stabilizing clock synchronization in sensor networks resilient to an adversary that can capture nodes and launch pulse-delay attacks. The core is a secure and self-stabilizing algorithm for sampling clocks of neighboring nodes.

The algorithm secures, with high probability, sets of complete neighborhood clock samples with a period that is $O((\log n)^2)$ times the optimum. The optimum requires, in the worst case, the communication of at least $O(n^2)$ timestamps. Here n is a bound on the number of sensor nodes that can interfere with a node (potentially the number of nodes within transmission range of the node). It is of high importance for high-precision clock synchronization that the clock sampling period is small since the offsets and frequencies of the nodes' clocks change over time.

Our design tolerates transient failures that may occur due to temporary violation of the designer's assumption. For example, the number of captured and/or pulse-delay attacked nodes could exceed more than f and then sink below f (delayed messages eventually vanish from queues). After the system resumes operation according to the designer's assumption, the system will stabilize within

one communication timeslot (that is of size $O(n \log n)$). We assume that (before and after the system's recovery) there are message omission failures, say, due to ambient noise, attacks or the algorithm's message collisions.

The correct node sends beacons and responds to the other nodes' beacons. We use a randomized strategy for beacon scheduling that guarantees regular message delivery with high probability.

2.1.2 Document structure

We start by describing the system settings (Section 2.2) and formally present the algorithm (Section 2.3). A description of our execution system model (Section 2.4) and a proof of the algorithm correctness (Section 2.5) are followed by a performance evaluation (Section 2.6). Then we review the literature and draw our conclusions (Section 2.7).

2.2 System Settings

We model the system as one that consists of a set of communicating entities, which we call processors (or nodes). We denote the set of processors by P . In addition, we assume that every processor $p_i \in P$ has a unique identifier, i . A processor identifier can be represented by a known and fixed number of bits in memory. In that respect there is a known upper bound on the number of processors.

2.2.1 Time, Clocks, and Their Notation

We follow settings that are compatible with those of Herman and Zhang [16]. We consider three notations of time: *real time* is the usual physical notion of continuous time, used for definition and analysis only; *native time* is obtained from a native clock, implemented by the operating system from hardware counters; *logical time* builds on native time with an additive adjustment factor. This

factor is adjusted to approximate a shared clock, whether local to a neighborhood or global to the entire network.¹

We consider applications that require the clock interface to include the *read* operation, which returns a *timestamp* with T possible states.² Let $C^i(t)$ denote the value $p_i \in P$ gets from a *read* of the native clock at real time t .

Clock counters do not increment at ideal rates, because the hardware oscillators have manufacturing variations and the rates are affected by voltage and temperature. The clock synchronization algorithm adjusts the logical clock in order to achieve synchronization, but never adjusts the native clock. We define the native clock *offset* between any two processors p_i and p_j as $\delta_{i,j}(t) = C^i(t) - C^j(t)$. We assume that, at any given time, the native clock offset is arbitrary. Moreover, the *skew* of p_i 's native clock, ρ_i , is the first derivative of the clock value with respect to real time. Thus $\rho_i = \lim_{\tau \rightarrow 0} (C^i(t + \tau) - C^i(t)) / \tau$. We assume that $\rho_i \in [\rho_{\min}, \rho_{\max}]$ for any processor p_i , where $\rho_{\min} = 1 - \kappa$ and $\rho_{\max} = 1 + \kappa$ are known constants, 1 is the real time unit and $\kappa \geq 0$. The second derivative of the clock's offset is called *drift*. We allow non-zero drift as long as $\rho_i \in [\rho_{\min}, \rho_{\max}]$.

2.2.2 Communications

Wireless transmissions are subject to collisions and noise. The processors communicate among themselves using the primitives *LBcast* and *LBrecv*, for local broadcast, with a transmission radius of at most R_{lb} . We consider the potential of any pair of processors to communicate directly, or to interfere with each other's communications.

We associate every processor, p_i , with a fixed and unknown location in space, L_i . We denote the potential set of processors that processor $p_i \in P$ can directly communicate with by $G_i \subseteq \{p_j \in P \mid R_{lb} \geq |L_i - L_j|\}$. Furthermore, we denote the set of processors that can interfere with the communications of

¹Lenzen et al. [17, 18] and Sommer and Wattenhofer [19] also refer to the term of logical time as "logical clock values". Herman and Zhang [16] refer to it as local time and build global time on top of the local time. See Section 2.7.

²In footnote 6 we show what the minimal size of T is.

p_i by $\vec{G}_i \subseteq \{p_j \in P \mid 2R_{lb} \geq |L_i - L_j|\}$. We note that G_i is not something processor p_i needs to know in advance, but something it discovers as it receives messages from other processors.

A successful broadcast by a processor p_i occurs when the message is received by all other processors in G_i . A successful broadcast to a set $K \subseteq G_i$ occurs when the message is received by all other processors in K .

We assume that $n \geq |\vec{G}_i|$ for any processor p_i . In other words, n is a known upper bound on the number of nodes that can interfere with any one node's communication (including that node itself). In the worst-case scenario $G_i = \vec{G}_i$ and thus potentially $|G_i| = n$. Furthermore, a node will receive information from neighbors about their neighbors, so in the worst-case scenario a node needs to keep track of data about n nodes. For simplicity we therefore use n as a bound of the number of neighbors (including the node itself) as well. This does not mean that we only consider a cluster of n nodes.

Communication Operations

We model the communication channel, $queue_{i,j}$, from processor p_i to processor $p_j \in G_i$ as a FIFO queue of the messages that p_i has sent to p_j and p_j is about to receive. When p_i broadcasts message m , the operation *LBcast* inserts a copy of m to every $queue_{i,j}$, such that $p_j \in G_i$. Every message $m \in queue_{i,j}$ is associated with a particular time at which m arrives at p_j . Once m arrives, p_j executes *LBrecv*. We require that the period between the time at which m enters the communication channel and the time at which m leaves it is at most a constant, d . We assume that d is a known and efficient upper bound on the communication delay between two neighboring processors. It includes both transmission delay and propagation delay, even though the propagation delay is negligible in comparison with the transmission delay.

We associate each *LBcast* and *LBrecv* operation with a native clock timestamp for the moment of sending and receiving. We assume the existence of an efficient algorithm for timestamping a message in transfer and a message being received as close to the physical layer as possible (see [10]).

The Environment

Messages might be lost to ambient noise as well as collisions of the nodes' transmissions. Collisions due to attacks made by the adversary or by captured nodes are called *adversarial collisions*. Message collisions due to concurrent transmissions of nodes that follow the message scheduling of the algorithm are called *non-adversarial collisions*. A broadcast that is not lost due to ambient noise or adversarial collisions is said to be *fair*. We note that a fair broadcast can still be lost due to non-adversarial collisions.

The environment can execute the operation *omission*(m_i) (which is associated with a particular message, m_i , sent by processor p_i) immediately after $L\text{Bcast}_i(m_i)$. The environment selects a (possibly empty) subset of p_i 's neighbors ($K_i \subseteq G_i$) and removes any message m_i from their queues $queue_{i,j}$ (such that $p_j \in K_i$).

Below we talk about what “the environment” selects when it comes to message omission. Here we see the environment as a global adversary, separate and independent from the “regular” malicious and locally bound adversary of Section 2.2.3. The term “adversary” is only used for that “regular” malicious adversary.

When a processor p_i and a processor $p_j \in \overrightarrow{G}_i$ do concurrent broadcasts of messages m_i and m_j we assume that the environment arbitrarily selects $K_i \subseteq G_i \cap G_j$ when invoking *omission*(m_i) due to the collision (and vice versa for m_j). For details on what it means in our execution system model see Section 2.4.3. In other words, when two processors with overlapping communication ranges broadcast concurrently, there are no guarantees of delivery, for those messages, within the overlap (regardless of noise). This is a simple and general model for message collisions. It is possible to let a more specialized physical layer model resolve the subset K_i .

The environment selects messages to omit due to ambient noise as described at the end of Section 2.2.2. The adversary selects messages to omit due to omission attacks as described at the end of Section 2.2.3.

Ambient noise

The parameter $\xi \geq 1$ denotes the maximal number of repeated transmissions required (by any particular processor) to get at least one fair broadcast. Such a broadcast can still be lost due to non-adversarial collisions. These assumptions model the ambient noise of the communication channel, as well as omission attacks by the adversary and by captured nodes (see Section 2.2.3). Furthermore, we assume that all processors know ξ .

The environment selects messages to remove due to ambient noise, but is limited by ξ as described above. We assume that the choice of messages omitted due to ambient noise is independent from the choice of messages omitted due to non-adversarial collisions.

2.2.3 The Adversary

We assume that there is a single adversary. The goal of the adversary is to disturb the clock synchronization algorithm so that clock samplings become erroneous, or even misleading. At the same time, the adversary does not want to let its presence be known by launching obvious attacks.

Omission Attacks and Delay Attacks

The adversary can launch omission and delay attacks against a message sent by another processor. We assume that at any time the adversary, just like all processors, has a distinct (unknown) location in space. We assume that the adversary's radio transmitter sends omnidirectional broadcasts (using antennas that radiate equally in space). Therefore, the adversary cannot arbitrarily control the distribution in space of the set of recipients for which a beacon's broadcast is omitted or delayed.

Consider a message, m_i , broadcast by a processor, p_i , and attacked by the adversary. We assume that the adversary chooses a sphere with its own location in the center. We denote the set of processors within the sphere S . The nodes in $S \cap G_i$ will be affected by the attack against m_i .

The adversary launches message omission attacks (also known as interception attacks) by jamming the medium. The environment invokes *omission*(m_i) for all processors in $S \cap G_i$. This selection is limited by the assumptions regarding ξ , as described in Section 2.2.3.

For delay attacks, we follow the model of Ganeriwal et al. [6, 7]. The adversary can receive (at least part of) a message, jam the medium for a set of nodes before they receive it in whole, and then replay the message slightly later. The adversary resends the message to the processors in $S \cap G_i$ after a chosen delay. The resent message is potentially lost due to ambient noise or collisions, like any other message. The processors in $S \cap G_i$ that receive m_i thus receive it later than they normally would have.

Other ways to do delay attacks include considering an adversary with directional antennas (which we do not consider) sending the same message at slightly different times in different directions, or having a captured node sending a message within a smaller radius and having the adversary repeating that within an area that was left out (see [8] for details). Both these delay attacks require the delayed message to originate from the adversary impersonating a captured node or from a captured node. We make the weaker assumption that a message from any processor can, potentially, be delayed by the adversary.

Omission Attack Limitations

We let ξ (see Section 2.2.2) include ambient noise as well as collisions deliberately produced by the adversary and by captured nodes. The adversary or the captured nodes could jam the medium such that the assumption of ξ does not hold. If too many messages are lost, however, that can act as an alarm that an adversary is present. This is something that the adversary, who wants to go undetected, wants to avoid. Furthermore, if the adversary totally jams the communication medium, clock synchronization will not take place. As a result, the adversary has no possibility to directly influence the logical clock. Thus, this is not an option for an adversary that wants to manipulate tracking algorithms to present a misleading view of its whereabouts and movements.

We note that the adversary cannot predict the broadcasting schedule of uncaptured nodes. Thus, adversarial collisions, covered by ξ (together with ambient noise), are independent from non-adversarial collisions.

Gilbert et al. [11] consider the minimal requirements for message delivery under broadcast interception attacks. They assume that the adversary intercepts no more than β broadcasts of the algorithm, where β is an unknown constant that reflects the maximum amount of energy an adversary wants to use for disruption of communications. We note that the result of Gilbert et al. is applicable in a model in which, in every period, the algorithm is able to broadcast at most α messages and the adversary can intercept at most β of the algorithm's messages. Our system settings are comparable to the assumptions made by Gilbert et al. [11] on the ratio of β/α . However, in contrast to the unknown β , we assume that the maximum ratio is a known constant that reflects the maximum amount of disruption the adversary can get away with, without being detected.

Captured Nodes

The adversary can capture nodes by moving to their location and accessing them physically. For any processor p_i , we assume that the number of captured and/or pulse-delay attacked nodes is no more than f , within its neighborhood, G_i . Here, f depends on $|G_i|$ and the filtering mechanism that is being used. (For example, $3f + 1 \leq |G_i|$ for the Byzantine agreement masking technique as in [7] and $2f + \epsilon \leq |G_i|$ for the outlier masking technique as in [8]; see Section 2.7 for more details.)

When the adversary captures a processor p_i , the adversary gains all information contained in the processor's memory, like secret keys, seeds for pseudorandom generators, etc. The adversary can lead a captured processor p_i to send incorrect data to processors in G_i . It can also lead the captured node to jam the communication media with noise or with collisions among processors in \vec{G}_i . The set of target processors are further limited to a sphere with the captured node in the center (cf. the sphere limitation for attacks launched directly by the adversary, in Section 2.2.3.) These noise and collision attacks are also

limited by ξ as described in Section 2.2.3, just like attacks launched directly by the adversary.

Security Primitives

The existing literature describes many elements of the secure implementation of the broadcast primitives *LBcast* and *LBrecv* using symmetric key encryption and message authentication (e.g., [9, 10]). We assume that neighboring processors store predefined pairwise secret keys. In other words, $p_i, p_j \in P : p_j \in G_i$ store keys $s_{i,j} : s_{i,j} = s_{j,i}$. The adversary cannot efficiently guess $s_{i,j}$. Confidentiality and integrity are guaranteed by encrypting the messages and adding a message authentication code. We can guarantee messages' freshness by adding a message counter (coupled with the beacon's timestamp) to the message before applying these cryptographic operations, and by letting receivers reject old messages, say, from the clock's previous incarnation. Note that this requires maintaining, for each sender, the index of the last properly received message. As explained above, the freshness criterion is not a suitable alternative to fine-grained clock synchronization in the presence of pulse-delay attacks.

2.3 Secure and Self-Stabilizing Clock Synchronization

In order to explain better the scope of the algorithm, we present a generic organization of secure clock synchronization protocols. The objective of the clock synchronization protocol is (1) to sample the clocks of its neighbors by periodically broadcast beacons, (2) respond to beacons, and (3) aggregate beacons with their responses in records and deliver them to the upper layer. Every node estimates the logical clock after sifting out responses to delayed beacons. Unlike objectives (1) to (3), the clock estimation task is not a hard real-time task. Therefore, the algorithm outputs records to the upper layer that synchronizes the logical clock after neutralizing the effect of pulse-delay attacks (see Section 2.7

for details on techniques for filtering out delayed messages). The algorithm focuses on the following two tasks.

- *Beacon Scheduling*: The nodes sample clock values by broadcasting beacons and waiting for their response. The task is to guarantee round-trip message exchange.
- *Beacon and Response Aggregation*: Once a beacon completes the round-trip exchange, the nodes can deliver to the upper layer the records of the beacon and its set of responses.

We present a design for an algorithm that samples clocks of neighboring processors by continuously sending beacons and responses. Without synchronized clocks, the nodes cannot efficiently follow a predefined schedule. Moreover, assuring reliable communication becomes hard in the presence of noise and message collisions. The celebrated Aloha protocol [20] (which does not consider nondeterministic fluctuating skews) inspires us to take a randomized strategy for scheduling broadcasts. We overcome the difficulties above and show that, with high probability, the neighboring processors are able to exchange beacons and responses within a short period. Our scheduling strategy is simple; the processors choose a random time to broadcast from a predefined period D . We use a redundant number of broadcasting timeslots in order to overcome the clocks' asynchrony. Moreover, we use a parameter, ℓ , used to trade off between the minimal size of D and the probability of having a collision-free schedule.

2.3.1 Beacon and Response Aggregation

The algorithm allows the use of clock synchronization techniques such as *reference broadcasting* [2] and *round-trip synchronization* [6, 7]. For example, in the round-trip synchronization technique, the sender p_j sends a timestamped message $\langle t_1 \rangle$ to receivers, $p_k \in G_j$, which receive the message at time t_2 . The receiver p_k responds with the message $\langle t_1, t_2, t_3 \rangle$, which p_k sends at time t_3 and p_j receives at time t_4 . Thus, the output records are in the form of

$\langle j, t_1, \{\langle k, \langle t_2, t_3, t_4 \rangle\} \rangle$, where $\{\langle k, \langle t_2, t_3, t_4 \rangle\} \}$ is the set of all received responses sent by nodes p_k .

We piggyback beacon and response messages. For the sake of presentation simplicity, let us start by assuming that all beacon schedules are in a (deterministic) Round Robin fashion. Given a particular node p_i and a particular beacon that p_i sends at time t_s^i , we define t_s^i 's *round* as the set of responses, $\langle t_s^j, t_r^j \rangle$, that p_i sends to node $p_j \in G_i$ for p_j 's previous beacon, t_s^j , where t_r^j is the time in which p_i received p_j 's beacon t_s^j . Node p_i piggybacks its beacon with the responses to nodes, p_j , and the beacon message, $\langle v_i \rangle$, is of the form $\langle t_s^i, \langle t_s^{j_1}, t_r^{j_1} \rangle, \langle t_s^{j_2}, t_r^{j_2} \rangle, \dots \rangle$, which includes all processors $p_{j_k} \in G_i$.

Now, suppose that the schedules are not done in a Round Robin fashion. We denote p_j 's sequence of up to $BLog$ most recently sent beacons with $[t_s^j(k)]_k$, where $0 \leq k < BLog$, among which $t_s^j(k)$ is the k -th oldest and $BLog$ is a predefined constant.³ We assume that, in every schedule, p_i receives at least one beacon from $p_j \in G_i$ before broadcasting $BLog$ beacons. Therefore, p_i 's beacon message, $\langle v_i \rangle$, can include a response to p_j 's most recently received beacon, $t_s^j(k)$, where $0 \leq k < BLog$.

Since not every round includes a response to the last beacon that p_i sends, p_i stores its last $BLog$ beacon messages in a FIFO queue, $q_i[k] = [t_s^j]_{0 \leq k < BLog}$. Moreover, every beacon message includes all responses to the $BLog$ most recently received beacons from all nodes. Let $q_j = [q_j[k]]_{0 \leq k < BLog}$ be p_i 's FIFO queue of the last $BLog$ records of the form $\langle t_s^j(k), t_r^j(k) \rangle$, among which $t_s^j(k)$ is p_i 's k -th oldest beacon from p_j , $t_r^j(k)$ is the time at which it was received and $i \neq j$. The new form of the beacon message is $\langle q_i, q_{j_1}, q_{j_2}, \dots \rangle$, which includes all processors $p_{j_k} \in G_i$. In the round-trip synchronization, the nodes take the role of a *synchronizer* that sends the beacon and waits for responses from the other nodes. The program of node p_i considers both cases in which p_i is, and is not, respectively, the synchronizer.

³We note that $BLog$ depends on the safety parameter, ℓ , for assuring that nodes successfully broadcast and other parameters such as the bound on number of interfering processors, n , and the bound on clock skews ρ_{\min} and ρ_{\max} (see Section 2.2).

2.3.2 The Algorithm's Pseudo-code

The pseudo-code, in Fig. 2.3, includes two procedures: (1) a do-forever loop that schedules and broadcasts beacon messages (lines 66 to 80) and (2) an upon message arrival procedure (lines 82 to 87).

The Do-Forever Loop

The do-forever loop periodically tests whether the “timer” has expired (in lines 67 to 74).⁴ In case the beacon’s next schedule is “too far in the past” or “too far in the future”, then processor p_i “forces” the “timer” to expire (line 69). The algorithm then removes data, gathered by p_i itself, that are too old (lines 70 to 71). (Note that under normal circumstances, the data never become too old before they are pushed out by new data at line 77 or line 86). The algorithm then tests that all the stored data (including data received from others) are ordered and timely (line 72). Timely here means that timestamps collected by a processor p_j is not too old or in the future compared to the latest time of p_j ’s native clock, that p_i has received. In the case where the recorded information about beacon messages is incorrect, the algorithm flushes the queues (line 73). The data received by others are tested at line 72 in the same way as at reception (line 83). Data that do not pass the test at line 83 are never stored. Therefore, if the buffers are flushed it is due to internal data corruption (in the starting configuration), and not due to receipt of bad data (during execution). We note that transient faults can be the source of such internal data corruption. However, bad data may be received (and therefore rejected) at any time during the execution, say, from captured nodes.

When the timeslot arrives, the processor outputs a synchronizer case record for the oldest beacon, in the queue with its own beacons (line 76). It contains, for each of the other processors, $p_j \in G_i$, the receive time of that beacon. Moreover, it contains for processor p_j , the send and receive times for a later message

⁴Recall that by our assumptions on the system settings (Section 2.2), the do-forever loop’s timer will go off within any period of $u/2$. Moreover, since the actual time cannot be predicted, we assume that the actual schedule has a uniform distribution over the period u . (A straightforward random scheduler can assist, if needed, to enforce the last assumption.)

back from p_j to p_i . These data can be used for the round-trip synchronization and delay detection in the upper layer. Then, p_i enqueues the timestamp of the beacon it is about to send during this schedule (line 77). The next schedule for processor p_i is set (lines 78 and 79) just before it broadcasts the beacon message (line 80).

The Message Arrival

When a beacon message arrives (line 82), processor p_i gets j , the id of the sender of the beacon, r , p_i 's native time at the receipt of the beacon, and v , the message of the beacon. The algorithm sanity checks the received data (line 83). If they are ordered and timely (not too old or in the future compared to the latest timestamp from p_j) the data are processed (lines 84 to 87). Otherwise the message is ignored.

Passing the sanity check, processor p_i then outputs a record of the non-synchronizer case (lines 84 to 85). These data can be used for the reference broadcast technique in the upper layer. It finds the oldest beacon in the queue with data on beacons received by p_j . The record contains responses from processors $p_k \in G_j$ that refer to this beacon. Furthermore, it contains data about later messages back, from the receiving processors p_k to processor p_j . Now that the information connected to the oldest beacon from p_j has been output, processor p_i can store the arrival time of newly received message (line 86) and the message itself (line 87).

2.4 Execution System Model

2.4.1 The Interleaving Model

Every processor, p_i , executes a program that is a sequence of (*atomic*) steps. For ease of description, we assume the interleaving model where steps are executed atomically, a single step at any given time. An input event, which can be either the receipt of a message or a timer going off, triggers each step of p_i . Only steps that start from a timer going off may include (at most once) an *LBCast*

Constants:

2 i = id of executing processor
 n = bound on # of interfering processors (incl. itself)
4 w = compensation time between lines 67 and 80
 d = upper bound on message communication delay
6 u = size of a timeslot in time units ($u > d + w$)
 ℓ = tuning parameter (see Corollary 2.1)
8 $BLog = 2 \lceil \xi \frac{\ell + \log_2((\lceil \rho_{\max} / \rho_{\min} \rceil + 1)^n)}{-\log_2(1 - 1/e)} \rceil$, backlog size
 $D = 3n (\lceil \rho_{\max} / \rho_{\min} \rceil + 1)$, the broadcast timeslots
10 T = number of possible states of a timestamp
 ρ_{\min} = lower bound on clock skew
12 ρ_{\max} = upper bound on clock skew

14 **Variables:**

$m[n]$ = all received messages and timestamp
16 each entry is an array $v[n]$
each entry is a queue $q[BLog]$
18 each entry is a pair $\langle s, r \rangle$

20 *native_clock* : immutable storage of the native clock
cslot : $[0, D-1]$ = current timeslot in use
22 *next* : $[0, T-1]$ = schedule of next broadcast
cT = last do-forever loop's timestamp

External functions:

26 *output*(R) : delivers record R to the upper layer
choose(S) : uniform selection of an item from the set S
28 *keys*(v) : the set of id:s that indexes v
enqueue(Q) : adds an element to the end of the queue Q
30 *dequeue*(Q) : removes the front element of the queue Q
size(Q) : size of the queue Q
32 *first*(Q) : least recently enqueued element in Q , number 0
last(Q) : most recently enqueued element in Q
34 *full*(Q) : whether queue Q is full
flush(Q) : empties the queue Q
36 *get_s*(Q) : list elements of field s in Q
get_r(Q) : list elements of field r in Q

Figure 2.1: Constants, variables and external functions for the secure and self-stabilizing native clock sampling algorithm in Fig. 2.3

operation. We note that there could be steps that read the clock and decide not to broadcast.

Since no self-stabilizing algorithm terminates (see [13]), the program of a processor consists of a do-forever loop. An iteration is said to be complete if it starts in the loop's first line and ends at the last (regardless of whether it enters conditional branches). A processor executes other parts of the program (and other programs) and activates the loop upon a time-out. We assume that

Macros and inlines:

```

40 border( $t$ ) : ( $D$ - $slot$ ) $u + t \pmod T$ 
   schedule( $t$ ) :  $slot$ - $u + t \pmod T$ 
42 leq( $x, y$ ) : ( $\exists b : 0 \leq b \leq 2 \text{ BLog } D u \wedge y \pmod T = x + b \pmod T$ )
   enq( $q, m$ ) : { while full( $q$ ) do dequeue( $q$ ); enqueue( $m$ ) }
44 G( $j$ ) : keys( $m[j].v$ )

46 expire_s( $q, t$ ): (* Expires data based on send times *)
   while size( $q$ ) > 0  $\wedge \neg$  leq(first( $q$ ). $s, t$ ) do
   dequeue( $q$ )
48 expire_r( $q, t$ ): (* Expires data based on receive times — as expire_s but with .r instead of .s *)
checkdata( $v, j$ ) :  $\wedge$  { checkdata( $m[j].v, j$ ) :  $j \in$  keys( $m[i].v$ ) }
50 checkdata( $v, j$ ) : (* Coherency test for data from processor  $j$  *)
    $\wedge$  { checklist(get_s( $v[k].q$ ), lclock( $v, j$ ))  $\wedge$  ( $j = k \vee$  checklist(get_r( $v[k].q$ ), lclock( $v, j$ ))) :  $k \in$  keys( $v$ ) }
52 checklist( $q, t$ ) : (* Checks that all elements of a list are chronologically ordered and not in the future *)
   size( $q$ ) = 0  $\vee$  (leq(first( $q$ ). $t$ )  $\wedge$  leq(last( $q$ ). $t$ )  $\wedge$  {leq( $q[b_1].q[b_2]$ ) :  $b_1 < b_2$ , { $b_1, b_2$ }  $\subseteq$  [1, size( $q$ )]}) )
54 lclock( $v, j$ ) : last( $v[j].q$ ). $s$ 
56 (* Get response-record for  $p_k$ , for  $p_j$  as the synchronizer *)
58 ts( $s, j, k$ ) : {if ( $\exists b_1^j, b_2^j, b_1^k, b_2^k$ :
    $s = m[j].v[j].q[b_1^j].s = m[k].v[j].q[b_1^k].s \wedge$ 
60  $m[k].v[k].q[b_2^k].s = m[j].v[k].q[b_2^j].s \wedge$ 
   leq( $m[j].v[j].q[b_1^j].s, m[j].v[k].q[b_2^j].r$ )  $\wedge$ 
62 leq( $m[k].v[j].q[b_1^k].r, m[k].v[k].q[b_2^k].s$ ))
   then return ( $m[k].v[j].q[b_1^k].r, m[k].v[k].q[b_2^k].s, m[j].v[k].q[b_2^j].r$ )
64 else return  $\perp$  }
```

Figure 2.2: Macros and inlines for the for the secure and self-stabilizing native clock sampling algorithm in Fig. 2.3.

every processor triggers the loop's time-out within every period of $u/2$, where $u > w + d$ is the (*operation*) *timeslot*, where $w < u/2$ is the time it takes to execute a complete iteration of the do-forever loop. Since processors execute programs other than the clock synchronization, the actual time, t , in which the timer goes off, is hard to predict. Therefore, for the sake of simplicity, we assume that time t is uniformly distributed.⁵

The *state* s_i of a processor p_i consists of the value of all the variables of the processor (including the set of all incoming communication channels, $\{queue_{j,i} | p_j \in G_i\}$). The execution of a step in the algorithm can change the state of a processor. The term *system configuration* is used for a tuple of the form (s_1, s_2, \dots) , where each s_i is the state of processor p_i (including mes-

⁵We note that a simple random scheduler can be used for the case in which time t does not follow a uniform distribution.

```

66 Do forever, every  $u/2$ 
    let  $cT = \text{read}(\text{native\_clock}) + w$ 
68 if  $\neg (\text{leq}(\text{next}-2Du, cT) \wedge \text{leq}(cT, \text{next}+u))$  then
     $\text{next} \leftarrow cT$ 
70 expire_s( $m[i].v[i].q, cT$ )
     $\forall j \in G(i) \setminus \{i\}$  do expire_r( $m[i].v[j].q, cT$ )
72 if  $\neg \text{check}()$  then
     $\forall j, k \in P$  do flush( $m[j].v[k].q$ )
74 if  $\text{leq}(\text{next}, cT) \wedge \text{leq}(cT, \text{next} + u)$  then
    let  $s = \text{first}(m[i].v[i].q).s$ 
76 output  $\langle i, s, \{ \langle j, \text{ts}(s, i, j) \rangle : j \in G(i) \setminus \{i\} \} \rangle$ 
    enq( $m[i].v[i].q, \langle cT, \perp \rangle$ )
78  $(\text{next}, \text{cslor}) \leftarrow (\text{border}(\text{next}), \text{choose}([0, D-1]))$ 
     $\text{next} \leftarrow \text{schedule}(\text{next})$ 
80 LBcast( $m[i]$ )

82 Upon LBrecv( $j, r, v$ )      ( $* i \neq j *$ )
    if checkdata( $v.j$ ) then
84 let  $s = \text{first}(m[i].v[j].q).s$ 
    output  $\langle j, s, \{ \langle k, \text{ts}(s, j, k) \rangle : k \in G(j) \setminus \{j\} \} \rangle$ 
86 enq( $m[i].v[j].q, \langle \text{last}(v[j].q).s, r \rangle$ )
     $m[j].v \leftarrow v$ 

```

Figure 2.3: Secure and self-stabilizing native clock sampling algorithm (code for $p_i \in P$).

sages in transit for p_i). We define an *execution* $E = c[0], a[0], c[1], a[1], \dots$ as an alternating sequence of system configurations $c[x]$ and steps $a[x]$, such that each configuration $c[x + 1]$ (except the initial configuration $c[0]$) is obtained from the preceding configuration $c[x]$ by the execution of the step $a[x]$. We often associate the notation of a step with its executing processor p_i using a subscript, e.g., a_i .

2.4.2 Tracing Timestamps and Communications

As stated in Section 2.2.2, we associate each *LBcast* and *LBrecv* operation with a timestamp for the moment of sending and receiving. The timestamp of an *LBcast* operation is the native time at which message m is sent, and this information is included in the sent message. When processor p_i executes the *LBrecv* operation, an event is triggered with the arguments j, t , and $\langle m \rangle$: $p_j \in G_i$ is the sending processor of message $\langle m \rangle$, which p_i receives when p_i 's native clock is t . We note that every step can be associated with at most one communication operation. Therefore it is sufficient to access the native clock

counter only once during or at the end of the operation. We denote by $C^i(a_i)$ the native clock value associated with the communication operation in step a_i , which processor p_i takes.

2.4.3 Concurrent vs. Independent Broadcasts

We say that processor $p_i \in P$ performs an *independent broadcast* in a step $a_i \in E$ if there is no processor $p_j \in \vec{G}_i$ that broadcasts in a step $a_j \in E$, such that either (1) a_j is performed after a_i and before step a_k^r that receives the message that was sent in a_i (where $p_k \in G_i$), or (2) a_i is performed after a_j and before step $a_{k'}^r$ that receives the message that was sent in a_j (where $p_{k'} \in G_j$). We say that processor $p_i \in P$ performs a *concurrent broadcast* in a step a_i if a_i is dependent (i.e., “not independent”). Concurrent broadcasts can cause message collisions, as described in Section 2.2.2.

2.4.4 Fair Executions

We say that execution E has *fair communications*, if, whenever processor p_i broadcasts ξ successive messages (successive in terms of the algorithm’s messages sent by p_i), at least one of these broadcasts is fair, i.e., not lost to noise or adversarial collisions. We note that fair communication does not imply reliable communication even for $\xi = 1$, because a message can still be lost due to non-adversarial collisions. An execution E is *fair* if the communications are fair and every correct processor, p_i , executes steps in a timely manner (by letting the loop’s timer go off in the manner that we explain above).

2.4.5 The Task

We define the system’s task by a set of executions called *legal executions* (LE) in which the task’s requirements hold. A configuration c is a *safe configuration* for an algorithm and the task of LE provided that any execution that starts in c is a legal execution (belongs to LE). An algorithm is *self-stabilizing* with

relation to the task of LE if every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

2.5 Correctness

In this section we demonstrate that the task of random broadcast scheduling is achieved by the algorithm that is presented in Fig. 2.3. Namely, with high probability, the scheduler allows the exchange of beacons and responses within a short time. The objectives of the random broadcast scheduling task are defined in Definition 2.1 and consider *broadcasting rounds*. To consider a number of broadcasting rounds from a point in time (such as the time associated with a step a), is to consider the time needed for every processor to fit in that many partitions, i.e., broadcast that many times.

Definition 2.1 (Nice executions) *Let us consider the executions of the algorithm presented in Fig. 2.3. Furthermore, let us consider a processor p_i and let Γ_i be the set of all execution prefixes, E_{Γ_i} , such that, within the first \mathcal{R} broadcasting rounds of E_{Γ_i} , (1) every processor $p_j \in G_i$ (including p_i) successfully broadcasts at least one beacon to all processors $p_k \in G_i \cap G_j$ and (2) every such processor p_j gets at least one response from all such processors p_k . We say that execution E is nice in relation to processor p_i if E has a prefix in Γ_i .*

The proof of Theorem 2.1 (Section 2.5.3, page 53) demonstrates that, when considering $\mathcal{R} = 2R$, for any processor p_i , the algorithm reaches a nice execution in relation to p_i with probability of at least $1 - 2^{-\ell+1}$, where ℓ is a predefined constant and $R = \lceil \xi \frac{\ell + \log_2(\lceil \frac{\rho_{\max}}{\rho_{\min}} \rceil + 1)n}{-\log_2(1-1/e)} \rceil$ is the expected time it takes all processors $p_j \in G_i$ (when considering the neighborhood of any processor p_i) to each broadcast at least one message that is received by all other processors in $G_i \cap G_j$.⁶

Once the system reaches a nice execution in relation to a processor p_i , and the exchange of beacons and responses occurs, the following holds. There is a

⁶ To distinguish between timestamps that should be regarded as being in the past and timestamps that should be regarded as being in the future, we require that $T > 4R$. In other words, we want to be able to consider at least 2 round-trips in the past and 2 round-trips in the future.

set, S_i , of beacon records that are in the queues of m_i and the records that were delivered to the upper layer. The set S_i includes a subset, $S'_i \subseteq S_i$, of records for beacons that were sent during the last \mathcal{R} (Definition 2.1) broadcasting rounds. In S'_i , it holds that every processor $p_j \in G_i - \{i\}$ has a beacon record, rec_j , such that every processor $p_k \in G_i \cap G_j - \{j\}$ has a beacon record, rec_k , which includes a response to rec_j . In other words, \mathcal{R} is a bound on the length of periods for which processor p_i needs to store beacon records. Moreover, with high probability, within \mathcal{R} broadcasting rounds, p_i gathers beacons from all processors $p_j \in G_i$. Furthermore, for each such beacon from a processor $p_j \in G_i$, p_i gathers responses to those beacons from all processors $p_k \in G_i \cap G_j$. For this reason, we set $BLog$ to be \mathcal{R} .

2.5.1 Scenarios in which balls are thrown into bins

We simplify the presentation of the analysis by depicting different system settings in which the message transmissions are described by a set of scenarios in which balls are thrown into bins. The sending of a message by processor p_i corresponds to a player \hat{p}_i throwing a ball. Time is discretized into timeslots that are long enough for a message to be sent and received within. The timeslots are represented by an unbounded sequence of bins, $[b_k]_{k \in \mathbb{N}}$. Transmitting a message during a timeslot corresponds to throwing a ball towards and *aiming a ball at* the corresponding bin.

Messages from processor p_i can collide with messages from up to $n - 1$ other processors if $|\vec{G}_i| = n$. Furthermore, in the worst-case scenario $|G_i| = |\vec{G}_i| = n$ for processor p_i . We want to guarantee with high probability that within G_i everyone exchanges messages. Therefore, we look at n players throwing balls into bins when analyzing the message scheduling algorithm. Our results will also hold for cases when $|G_i| < n$ and when $|\vec{G}_i| < n$, as the probability of collisions in those cases is equal to or lower than that for the worst case scenario.

Before analyzing the general system settings, we demonstrate simpler settings to acquaint the reader with the problem. Concretely, we look at the set-

tings in which the clocks of the processors are perfectly synchronized and the communication channels have no noise (or omission attacks). We ask the following question: How many bins are needed for every player to get at least one ball, that is not lost due to collisions, in a bin (Lemma 2.1 and 2.2)? We then relax the assumptions on the system settings by considering different clock offsets (Claim 2.2) and by considering different clock skews (Claim 2.3). We continue by considering noisy communication channels (and omission attacks) (Claim 2.4) and conclude the analysis by considering general system settings (Corollary 2.1).

Collisions

A message collision corresponds to two or more balls aimed at the same bin. We take the pessimistic assumption that, when balls are aimed at neighboring bins, they collide as well. This is to take non-discrete time (and later on, different clock offsets) into account. Broadcasts that “cross the borders” between timeslots are assumed to collide with messages that are broadcast in either bordering timeslot. Therefore, in the scenario in which balls are thrown into bins, two or more balls aimed at the same bin or bordering bins will bounce out, i.e., not end up in the bin.

Definition 2.2 *When aiming balls at bins in a sequence of bins, a successful ball is a ball that is aimed at a bin b . Moreover, it is required that no other ball is aimed at b or a neighboring bin of b . A neighboring bin of b is the bin directly before or directly after b . An unsuccessful ball is a ball that is not successful.*

Synchronous timeslots and communication channels that have no noise

We prove a claim that is needed for the proof of Lemma 2.1.

Claim 2.1 *For all $x \geq 2$ it holds that*

$$\left(1 - \frac{1}{x}\right)^{x-1} > \frac{1}{e}. \quad (2.1)$$

Proof: It is well known that

$$\left(1 + \frac{1}{x}\right)^x < e \quad (2.2)$$

for any $x \geq 1$. From this it follows that

$$\begin{aligned} \left(1 - \frac{1}{x}\right)^{x-1} &= \left(\frac{x-1}{x}\right)^{x-1} = \left(\frac{x}{x-1}\right)^{-(x-1)} \\ &= \left(1 + \frac{1}{x-1}\right)^{-(x-1)} = \frac{1}{\left(1 + \frac{1}{x-1}\right)^{x-1}} > \frac{1}{e} \end{aligned} \quad (2.3)$$

for $x \geq 2$. ■

Lemmas 2.1 and 2.2 consider an unbounded sequence of bins that are divided into “circular” subsequences that we call *partitions*. We simplify the presentation of the analysis by assuming that the partitions are independent. Namely, a ball that is aimed at the last bin of one partition normally counts as a collision with a ball in the first bin of the next partition. With this assumption, a ball aimed at the last bin and a ball aimed at the first bin in the same partition count as a collision instead. These assumptions do not cause a loss of generality, because the probability for balls to collide does not change. It does not change because the probability for having a certain number of balls in a bin is symmetric for all bins.

We continue by proving properties of scenarios in which balls are thrown into bins. Lemma 2.1 states the probability of a single ball being unsuccessful.

Lemma 2.1 *Let n balls be, independently and uniformly at random, aimed at partitions of $3n$ bins. For a specific ball, the probability that it is not successful is smaller than $1 - 1/e$.*

Proof: Let b be the bin that the specific ball is aimed at. For the ball to be successful, there are 3 out of the $3n$ bins that no other ball should be aimed at, b and the two neighboring bins of b . The probability that no other (specific) ball is aimed at any of these three bins is

$$1 - \frac{3}{3n}. \quad (2.4)$$

The different balls are aimed independently, so the probability that none of the other $n - 1$ balls are aimed at bin b or a neighboring bin of b is

$$\left(1 - \frac{3}{3n}\right)^{n-1} = \left(1 - \frac{1}{n}\right)^{n-1}. \quad (2.5)$$

With the help of Claim 2.1, the probability that at least one other ball is aimed at b or a neighboring bin of b is

$$1 - \left(1 - \frac{1}{n}\right)^{n-1} < 1 - \frac{1}{e}. \quad (2.6)$$

■

Lemma 2.2 states the probability of any player not having any successful balls after a number of throws.

Lemma 2.2 *Consider R independent partitions of $D = 3n$ bins. For each partition, let n players aim one ball each, uniformly and at random, at one of the bins in the partition. Let $R \geq (\ell + \log_2 n)/(-\log_2 p)$, where $p = 1 - 1/e$ is an upper bound on the probability of a specific being unsuccessful in a partition. The probability that any player gets no successful ball is smaller than $2^{-\ell}$.*

Proof: By Lemma 2.1, the probability that a specific ball is unsuccessful is upper bounded by $p = 1 - 1/e$. The probability that a player does not get any successful ball in any of R independent partitions is therefore upper bounded by p^R .

Let $X_i, i \in [1, n]$ be Bernoulli random variables with the probability of a ball being successful that is upper bounded by p^R :

$$X_i = \begin{cases} 1 & \text{if player } i \text{ gets no successful ball in } R \text{ partitions} \\ 0 & \text{if player } i \text{ gets at least one successful ball in } R \text{ partitions} \end{cases} \quad (2.7)$$

Let X be the number of players that get no successful ball in R partitions:

$$X = \sum_{i=1}^n X_i. \quad (2.8)$$

The different X_i are a finite collection of discrete random variables with finite expectations. Therefore we can use the Theorem of Linearity of Expectations [21]:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i] \leq \sum_{i=1}^n p^R = np^R. \quad (2.9)$$

The random variables assume only non-negative values. Markov's Inequality [21], $\Pr(X \geq a) \leq \mathbb{E}[X]/a$, therefore gives us

$$\Pr(X \neq 0) = \Pr(X \geq 1) \leq \frac{\mathbb{E}[X]}{1} \leq np^R \quad (2.10)$$

For $np^R \leq 2^{-\ell}$ we get that $\Pr(X \neq 0) \leq 2^{-\ell}$, which gives us

$$\begin{aligned} np^R \leq 2^{-\ell} &\Rightarrow \\ \log_2(np^R) &\leq -\ell \Rightarrow \\ \log_2(n) + R \log_2(p) &\leq -\ell \Rightarrow \\ R &\geq \frac{-\ell - \log_2 n}{\log_2 p} = \frac{\ell + \log_2 n}{-\log_2 p}. \end{aligned} \quad (2.11)$$

■

We now turn to relaxing the simplifying assumptions of synchronized clocks and communication channels with no noise. We start by considering clock offsets and skews. We then consider noisy communication channels.

Clock offsets

The clocks of the processors have different offsets, and therefore the timeslot boundaries are not aligned. We consider a scenario that is comparable to system settings in which clocks have offsets. In the scenario of balls that are thrown into bins, offsets are depicted as throwing a ball that hits the boundary between bins and perhaps hits the boundary between partitions.

Claim 2.2 considers players that have individual sequences of bins. Each sequence has its own alignment of the bin boundaries. Namely, a bin of one player may “overlap” with more than one bin of another player. Thus, the different bin

sequences that have different alignments correspond to system settings in which clocks have different offsets.

The proof of Claim 2.2 describes a variation of the scenario in which balls are thrown into bins. In the new variation, balls aimed at overlapping bins will bounce out. For example, consider two balls aimed at bin b_i^k and $b_j^{k'}$, respectively. If bins b_i^k and $b_j^{k'}$ overlap, the balls will cause each other to bounce out.

Claim 2.2 *Consider the scenario in which balls might hit the bin boundaries and take R and D as defined in Lemma 2.2. Then, we have that the probability that any player gets no successful ball is smaller than $2^{-\ell}$.*

Proof: The proof is demonstrated by the following two arguments.

Hitting the boundaries between bins. From the point of view of processor p_i , a timeslot might be the time interval $[t, t + u)$, whereas for processor p_j the timeslot interval might be different and partly belong to two different timeslots of p_i . When considering the scenario in which balls are thrown into bins, we note that a bin of one player might be seen as parts of two bins of another player.

In other words, every player, \hat{p}_i , has its own view, $[b_k^i]_{k \in \mathbb{N}}$, of the bin sequence $[b_k]_{k \in \mathbb{N}}$. The sequence $[b_k]_{k \in \mathbb{N}}$ corresponds to an ideal discretization of the real time into timeslots, whereas the sequence $[b_k^i]_{k \in \mathbb{N}}$, corresponds to a discretization of processor p_i 's native time into timeslots. We say that the bins $[b_k^i]$ and $[b_{k'}^j]$ overlap when the corresponding real time periods of $[b_k^i]$ and $[b_{k'}^j]$ overlap.

Lemma 2.2 regards balls aimed at neighboring bins as collisions. We recall the requirements that are made for ball collisions (see Section 2.5.1). These requirements say that balls aimed at neighboring bins in $[b_k]_{k \in \mathbb{N}}$ will bounce out. The proof is completed by relaxing the requirements that are made for ball collisions in $[b_k]_{k \in \mathbb{N}}$. Let us consider the scenario in which players \hat{p}_i and \hat{p}_j aim their balls at bins b_k^i and $b_{k'}^j$, respectively, such that both b_k^i and $b_{k'}^j$ overlap. The bin b_k^i can either overlap with the bins $b_{k'-1}^j$ and $b_{k'}^j$ or (exclusively) overlap with the bins $b_{k'}^j$ and $b_{k'+1}^j$. Balls aimed at any of the bins possibly overlapping with b_k^i (namely $b_{k'-1}^j$, $b_{k'}^j$ and $b_{k'+1}^j$) are regarded as colliding with the ball of

player \hat{p}_j . The same argument applies to bin $b_{k'}^j$, overlapping with bins b_{k-1}^i , b_k^i and b_{k+1}^i . In other words, the scenario of Lemma 2.2, without offset and neighboring bins leading to collision, is a superset in terms of bin overlap to the scenario in which offsets are introduced.

Hitting the boundaries between partitions. Even if the timeslot boundaries are synchronized, processor p_i might regard the time interval $[t, t + Du)$ as a partition, whereas processor p_j might regard the interval $[t, t + Du)$ as partly belonging to two different partitions. When considering the scenario in which balls are thrown into bins, this means that the players' view on which bins are part of a partition can differ.

For each bin, the probability that a specific player chooses to aim a ball at that bin is $1/D$, where D is the number of bins in the partition. Therefore the probability for a ball being successful does not depend on how other players partition the bins. ■

Clock skews

The clocks of the processors have different skews. Therefore, we consider a scenario that is comparable to system settings in which clocks have skews.

In Claim 2.3, we consider players that have individual sequences of bins. Each sequence has its own bin size. The size of player \hat{p}_i 's bins is inversely proportional to processor p_i 's clock skew, say $1/\rho_i$. We assume that the balls that are thrown by any player can fit into the bins of any other player. (Say the ball size is less than $1/\rho_{\max}$.) Thus, the different bin sizes correspond to system settings in which clocks have different skews.

Let us consider the number of balls that player \hat{p}_i may aim at bins that overlap with bins in a partition of another player. Suppose that player \hat{p}_i has bins of size $1/\rho_{\max}$ and that player \hat{p}_j has bins of size $1/\rho_{\min}$. Then player \hat{p}_i may aim up to $\hat{\rho} = \lceil \rho_{\max}/\rho_{\min} \rceil + 1$ balls in one partition of player \hat{p}_j .

Claim 2.3 *Consider the scenario with clock skews and take R and D as defined in Lemma 2.2. Let $p = 1 - 1/e$ be an upper bound on the probability of a specific ball being unsuccessful in a partition. By taking $R_{skew} = R \geq$*

$(\ell + \log_2 \hat{\rho}n)/(-\log_2 p) \in O(\ell + \log(n))$, we have that the probability that any player gets no successful ball is smaller than $2^{-\ell}$.

Proof: By taking the pessimistic assumption that all players see the others, as well as themselves, as throwing $\hat{\rho}$ balls each in every partition we have an upper bound on how many balls can interfere with each other in a partition. Thus by taking partitions of $D = 3\hat{\rho}n$ bins instead of the $3n$ bins of Lemma 2.2, and substituting n for $\hat{\rho}n$ in the R of Lemma 2.2,

$$R \geq \frac{\ell + \log_2 \hat{\rho}n}{-\log_2 p} \in O(\ell + \log(n)), \quad (2.12)$$

the guarantees of Lemma 2.2 hold. ■

Communication channels with noise

In our system settings, message loss occurs due to noise and omission attacks and not only due to the algorithm's message collisions. Recall that ξ defines the number of broadcasts required in order to guarantee at least one fair broadcast (not lost to noise or adversarial collisions; see Section 2.2.2). In the scenario in which balls are thrown into bins, this correspondingly means that at most $\xi - 1$ balls are lost to the player's trembling hand for any of its ξ consecutive throws. Omission attacks are incorporated into the ξ assumption and are thus not seen as a ball being thrown.

Claim 2.4 *Consider the communication channels with noise and take R and D as defined in Lemma 2.2. By taking $R_{noise} \geq \xi R$, we have that the probability that any player gets no successful ball is smaller than $2^{-\ell}$.*

Proof: By the system settings (Section 2.2), the noise in the communication channels is independent of collisions. We take the pessimistic approach and assume that, when a ball is lost to noise, it can still cause other balls to be unsuccessful (just as if it was not lost to noise). In order to fulfill the requirements of Lemma 2.2, we can take ξR partitions instead of R partitions. This will guarantee that each player gets at least R "fair" balls. That is, each player

gets at least R balls that are either successful or that bounce out due to collision with another ball. Thus, the asymptotic number of bins is unchanged and the guarantees of Lemma 2.2 still hold. ■

General system settings

The results gained from studying the scenario in which balls are thrown into bins are concluded by Corollary 2.1, which is demonstrated by Lemma 2.2 and claims 2.2, 2.3, and 2.4.

Corollary 2.1 *Suppose that every processor broadcasts once in every partition of D timeslots. Consider any processor p_i . The probability that every processor $p_j \in G_i$ successfully broadcasts at least one beacon to every processor $p_k \in G_i \cap G_j$ within R partitions is at least $1 - 2^{-\ell}$ when*

$$D = 3\hat{\rho}n \in O(n) \quad (2.13)$$

$$R = \left\lceil \xi \frac{\ell + \log_2(\hat{\rho}n)}{-\log_2 p} \right\rceil \in O(\ell + \log n) \quad (2.14)$$

$$\hat{\rho} = \lceil \rho_{max}/\rho_{min} \rceil + 1 \quad (2.15)$$

$$p = 1 - \frac{1}{e}. \quad (2.16)$$

Corollary 2.1 shows that, for any processor p_i , within a logarithmic number of broadcasting rounds, all processors in G_i exchange at least one beacon with their neighbors in G_i , with high probability. (See the beginning of Section 2.5.1 for the discussion on the n balls versus a processor p_i for which $|\vec{G}_i| < n$.)

2.5.2 The task of random broadcast scheduling

So far, we have analyzed a general scenario in which balls are thrown into bins. We now turn to showing that the scenario indeed depicts the implementation of the algorithm (which is presented in Fig. 2.3).

As stated earlier, when we talk about the execution of, or complete iteration of, lines 67 to 80, we do not imply that the branch in lines 75 to 80 necessarily is entered.

Definition 2.3 (Safe configurations) *Let E be a fair execution of the algorithm presented in Fig. 2.3 and $c \in E$ a configuration in which $\alpha_i = (\text{leq}(\text{next}_i - 2Du, cT_i) \wedge \text{leq}(cT_i, \text{next}_i))$ holds for every processor p_i . We say that c is safe with respect to LE .*

We show that cT_i follows the native clock of processor p_i . Namely, the value of $cT_i - w$ is in $[C^i - u, C^i]$.

Lemma 2.3 *Let E be a fair execution of the algorithm presented in Fig. 2.3, and c a configuration that is at least u after the starting configuration. Then, it holds that $(\text{leq}(C^i - u, cT_i - w) \wedge \text{leq}(cT_i - w, C^i))$ in c .*

Proof: Since E is fair, the do-forever loop's timer goes off in every period of $u/2$. Hence, within a period of u , processor p_i performs a complete iteration of the do-forever loop in an atomic step a_i .

Suppose that c immediately follows a_i . According to line 67, the value of $cT_i - w$ is the value of C^i in c . Let $t = cT_i - w = C^i$. It is easy to see that $\text{leq}(t - u, t) \wedge \text{leq}(t, t)$ in c .

Let a_i^r be an atomic step that includes the execution of lines 83 to 87 (whether entering the branch or not), follows c , and immediately precedes $c' \in E$. Let $t' = C^i$ in c' . Then, within a period of at most $u/2$, processor p_i executes step $a_i^r \in E$, which includes a complete iteration of the do-forever loop. Since the period between a_i and a_i^r is at most $u/2$, we have that $t' - t < u/2$. Therefore $\text{leq}(C^i - u, cT_i - w)$ holds in c' as $\text{leq}(C^i, cT_i - w)$ holds in c . It also follows that $\text{leq}(cT_i - w, C^i)$ holds in c' as $C^i = cT_i - w$ in c . ■

We show that when a processor p_i executes lines 75 to 80 of the algorithm presented in Fig. 2.3 it reaches a configuration in which α_i holds. This claim is used in Lemma 2.4 and Lemma 2.5.

Claim 2.5 *Let E be a fair execution of the algorithm presented in Fig. 2.3. Moreover, let $a_i \in E$ a step that includes a complete iteration of lines 67 to 80 and c the configuration that immediately follows a_i . Suppose that processor p_i executes lines 75 to 80 in a_i ; then α_i holds in c .*

Proof: Among the lines 75 to 80, only lines 78 to 79 can change the values of α_i . Let $t_1 = next_i$ immediately after line 74 and let $t_2 = next_i$ immediately after the execution of line 79. We denote by $A = t_2 - t_1$ the value that lines 78 to 79 add to $next_i$, i.e., $A = (y + D - x)u$, where $0 \leq x, y \leq D - 1$. Note that x is the value of $cslot_i$ before line 78 and y is the value of $cslot_i$ after line 78. Therefore, $A \in [u, (2D - 1)u]$.

By the claim's assertion, we have that $leq(cT_i, t_1 + u)$ holds before line 78. Since $u \leq A$, it holds that $leq(cT_i, t_1 + A)$, and therefore $leq(cT_i, t_2)$ holds.

Moreover, by the claim assertion we have that $leq(t_1, cT_i)$ holds. Since $A \leq (2D - 1)u$, it holds that $A - 2Du \leq -u$. This implies that $leq(t_1 - 2Du + A, cT_i)$. Therefore $leq(t_2 - 2Du, cT_i)$ holds. ■

We show that, starting from an arbitrary configuration, any fair execution reaches a safe configuration.

Lemma 2.4 *Let E be a fair execution of the algorithm presented in Fig. 2.3. Then, within a period of u , a safe configuration is reached.*

Proof: Let p_i be a processor for which α_i does not hold in the starting configuration of E . We show that, within the first complete iteration of lines 67 to 80, the predicate α_i holds. According to Lemma 2.3, all processors, p_i , complete at least one iteration of lines 67 to 80, within a period of u .

Let $a_i \in E$ be the first step in which processor p_i completes the first iteration. If α_i does not hold in the configuration that immediately precedes a_i , then either (1) the predicate in line 68 holds and processor p_i executes line 69 or (2) the predicate of line 74 holds at line 68.

For case (2), as $\neg(leq(t - 2Du, t) \wedge leq(t, t))$ is false for any t , immediately after the execution of line 69, the predicate $\neg(leq(next_i - 2Du, cT_i) \wedge leq(cT_i, next_i))$ does not hold. Moreover, the predicate in line 74 holds, since $leq(t, t + u)$ holds for any t .

In other words, the predicate in line 74 holds for both cases (1) and (2). Therefore, p_i executes lines 75 to 80 in a_i . By Claim 2.5, α_i holds for the configuration that immediately follows a_i . By repeating this argument for all

processors p_i , we show that a safe configuration is reached within a period of u . ■

We demonstrate the closure property of safe configurations.

Lemma 2.5 *Let E be a fair execution of the algorithm presented in Fig. 2.3 that starts in a safe configuration c , i.e. a configuration in which α_i holds for every processor p_i (Definition 2.3). Then, every configuration in E is safe with respect to LE .*

Proof: Let t_i be the value of p_i 's native clock in configuration c and $a_i \in E$ be the first step of processor p_i .

We show that α_i holds in configuration c' that immediately follows a_i . Lines 83 to 87 do not change the value of α_i . By Claim 2.5, if a_i executes lines 75 to 80 within one complete iteration, then α_i holds in c' . Therefore, we look at step a_i that includes the execution of lines 67 to 74, but does not include the execution of lines 75 to 80.

Let $t_1 = cT_i$ in c and $t_2 = c'T_i$ in c' . According to Lemma 2.3, and by the fairness of E , we have that $t_2 - t_1 \bmod T < u$. Furthermore, let $A = next_i - Du$ and $B = next_i$ in c . The values of $next_i - Du$ and $B = next_i$ do not change in c' . Since α_i is true in c , it holds that $leq(A, t_1) \wedge leq(t_1, B)$. We claim that $leq(A, t_2) \wedge leq(t_2, B)$. Since $leq(t_1, B)$ in c , we have that $leq(t_2, B + t_2 - t_1)$ while p_i executes line 74 in a_i . As a_i does not execute lines 75 to 80, the predicate in line 74 does not hold in a_i . As $leq(t_1, B)$ and $t_2 - t_1 \bmod T < u$ the predicate in line 74 does not hold iff $leq(t_2, B)$. Furthermore, we have that $leq(A, t_1)$, $leq(t_1, B)$, and $leq(t_2, B)$. As $0 < t_2 - t_1 \bmod T < u$ we have that $leq(A, t_2)$. Thus, c' is safe as α_i holds in c' . ■

2.5.3 Nice executions

We claim that the algorithm (presented in Fig. 2.3) implements nice executions with high probability. We show that, for any processor p_i , every execution (for which the safe configuration requirements hold) is a nice execution in relation to p_i with high probability.

Theorem 2.1 *Let E be a legal execution of the algorithm presented in Fig. 2.3. Then, for any processor p_i , E is nice in relation to p_i with high probability.*

Proof: Recall that in a legal execution all configurations are safe (Section 2.2). Let a_i be a step in which processor p_i broadcasts, a'_i be the first step after a_i in which processor p_i broadcasts, and a''_i be the first step after a'_i in which processor p_i broadcasts.

Let r , r' , and r'' be the values of $next_i$ between lines 78 and 79 in a_i , a'_i , and a''_i , respectively. The only changes done to $next_i$ from line 79 in a_i to lines 78 and 79 in a'_i are those two lines, which taken together change $next_i$ to $next_i + Du \pmod T$.

The period of length Du that begins at r and ends at $r' \pmod T$ is divided in D timeslots of length u . A timeslot begins at time $r + xu \pmod T$ and ends at time $r + (x + 1)u \pmod T$ for a unique integer $x \in [0, D - 1]$. The timeslot in which a'_i broadcasts is $cslot$ in c . In other words, processor p_i broadcasts within a timeframe of r to r' , which is of length Du . By the same arguments, we can show that processor p_i broadcasts within a timeframe of r' to r'' , which is of length Du . These arguments can be used to show that, after a_i , processor p_i broadcasts once per period of length Du .

Corollary 2.1 considers processor p_i , and its set G_i , which includes itself and its neighbors. The processors in \vec{G}_i broadcast once in every period of D timeslots. The timeslots are of length u , a period that each processor estimates using its native clock. Let us consider a processor p_i and R timeframes of length Du . By Corollary 2.1, the probability that all processors $p_j \in G_i$ successfully broadcast at least one beacon to all processors $p_k \in G_i \cap G_j$ is at least $1 - 2^{-\ell}$. Now, let us consider $2R$ timeframes of length Du . Consider the probability that each of the processors $p_j \in G_i$ successfully broadcasts to all processors $p_k \in G_i \cap G_j$ and get a response from all such processors p_k . By Corollary 2.1, that probability is at least $(1 - 2^{-\ell})^2 = 1 - 2^{-\ell+1} + 2^{-2\ell} > 1 - 2^{-\ell+1}$. Therefore, by Definition 2.1, for any processor p_i , E is nice in relation to p_i with high probability. ■

2.6 Performances of the algorithm

Several elements determine the precision of the clock synchronization. The clock sampling technique is one of them. Elson et al. [2] show that the reference broadcast technique can be more precise than the round-trip synchronization technique. We allow the use of both techniques. Another important precision factor is the quality of the approximation of the native clocks of neighboring nodes. Our extensive clock sample records allows for both linear regression and phase-locked looping (see Römer et al. [5]). Moreover, the clock synchronization precision improves as neighboring processors are able to sample each other's clocks more frequently. However, due to the limited energy reserves in sensor networks, careful considerations are required.

Let us consider the continuous operation mode. If the period of the clock samples is too long, the clock precision suffers, as the skews of the native clocks are not constant. Thus, an important measure is $round_i$, where $round_i$ is the time it takes a processor p_i and its neighbors in G_i to exchange beacons and responses. In other words, $round_i$ is the time it takes (1) every processor $p_j \in G_i$ (including p_i) to successfully broadcast at least one beacon to all processors $p_k \in G_i \cap G_j$ and (2) every such processor p_j to get at least one response from all such processors p_k .

Let us consider ideal system settings in which broadcasts never collide. In the worst case, $|G_i| = |\vec{G}_i| = n$. Sending n beacons and getting n responses to each of these beacons requires the communication of at least $O(n^2)$ samples. By Corollary 2.1 and Theorem 2.1, we get that $2R$ timeframes of length Du are needed. We also get that $R \in O(\log n)$ and $D \in O(n)$. The timeslot size u is needed to fit a message with $BLog = 2R$ responses to up to n processors. Hence, $u \in O(n \log n)$. Therefore $round_i \in O(n^2(\log n)^2)$. Moreover, with a probability of at least $1 - 2^{-\ell+1}$, the algorithm can secure a clock sampling period that is $O((\log n)^2)$ times the optimum.

We note that the required storage is in $O(n^2 \log n \log T)$. By Lemma 2.4, starting in an arbitrary configuration, our system stabilizes within u time, and as we have seen above $u \in O(n \log n)$.

2.6.1 Optimizations

We can use the following optimization, which is part of many existing implementations. Before accessing the communication media, a processor p_i waits for a period d and broadcasts only if there was no message transmitted during that period. Thus, processor p_i does not intercept broadcasts, from a processor $p_j \in G_i$, that it started receiving (and did not finish) before time $t - d$, where t is the time of the broadcast by p_i . In that case it aborts its message. For p_i , and for the sake of the worst-case analysis, this counts as a collision. However, for p_j it is a successful broadcast (assuming that the message is not lost to noise or to collision with another message).

2.7 Discussion

Sensor networks are particularly vulnerable to interference, whether as a result of hardware malfunction, environmental anomalies, or malicious intervention. When dealing with message collisions, message delays and noise, it is hard to separate malicious from non-malicious causes. For instance, it is hard to distinguish between a pulse-delay attack and a combination of failures, e.g., a node that suffers from a hidden terminal failure, but receives an echo of a beacon. Recent studies consider more and more implementations that take security, failures and interference into account when protecting sensor networks (e.g., [22–24], which consider multi-channel radio networks). We note that many of the existing implementations assume the existence of a fine-grained synchronized clock, which we implement.

Message scheduling is important for clock synchronization. Moradi et al. compare clock synchronization algorithms for wireless sensor networks considering precision, cost and fault tolerance [25]. They show that, without a message scheduling algorithm of some sort, the Reference Broadcast algorithm of [2] suffers heavily from collisions.

Ganerwal et al. [7] overcome the challenge of delayed beacons using the round-trip synchronization technique. With this technique the average delay of

a message from processor p_i to processor $p_j \in G_i$, and a message back from p_j to p_i , can be calculated using the send and receive times of those messages. Thus, a delay attack can be detected if the delay is larger than some known upper bound on message delay. They use the Byzantine agreement protocol [26] for a cluster of g nodes where all g nodes are within transmission range of each other. Thus, Ganeriwal et al. require $3f + 1 \leq g$. Song et al. [8] consider a different approach that uses the reference broadcasting synchronization technique. Existing statistics models refer to malicious time offsets as outliers. The statistical outlier approach is numerically stable for $2f + \epsilon \leq g$, where g is the number of neighbors and where ϵ is a safety constant (see [8]). We note that both approaches are applicable to our work. We further note that a processor $p_k \in G_j \cap G_i$ can detect delay attacks against beacons that nodes p_i and p_j have sent to each other, by the mechanisms of calculating average message delay and comparing with a known upper bound. This is possible because p_k gets send and receive times of messages back and forth between p_i and p_j .

Based on our practical assumptions, we are able to avoid the Byzantine agreement overheads and follow the approach of Song et al. [8]. We can construct a self-stabilizing version of their strategy, by using our sampling algorithm and by detecting outliers using the generalized extreme studentized deviate (GESD) algorithm [27]. Let B be the set of delivered beacon records within a period of \mathcal{R} and test the set B for outliers using the GESD algorithm.

Existing implementations of secure clock synchronization protocols [6–8, 10, 28–30] are not self-stabilizing. Thus, their specifications are not compatible with security requirements for autonomous systems. In autonomous systems, the self-stabilization design criteria are imperative for secure clock synchronization. For example, many existing implementations require initial clock synchronization prior to the first pulse-delay attack (during the protocol set up). This assumption implies that the system uses global restart for self-defense management, say, using an external intervention. We note that the adversary is capable of intercepting messages continually. Thus, the adversary can risk detection and intercept all pulses for a long period. Assume that the system detects the

adversary's location and stops it. Nevertheless, the system cannot synchronize its clocks without a global restart.

Sun et al. [31] describe a cluster-wise synchronization algorithm that is based on synchronous broadcasting rounds. The authors assume that a Byzantine agreement algorithm [26] synchronizes the clocks before the system executes the algorithm. Our algorithm is comparable with the requirements of autonomous systems and makes no assumptions on synchronous broadcasting rounds or start.

Manzo et al. [30] describe several possible attacks on an (unsecured) clock synchronization algorithm and suggest countermeasures. For single hop synchronization, the authors suggest using a randomly selected "core" of nodes to minimize the effect of captured nodes. The authors do not consider the cases in which the adversary captures nodes after the core selection. In this work, we make no assumption regarding the distribution of the captured nodes. Farrugia and Simon [29] consider a cross-network spanning tree in which the clock values propagate for global clock synchronization. However, no pulse-delay attacks are considered. Sun et al. [28] investigate how to use multiple clocks from external source nodes (e.g., base stations) to increase the resilience against an attack that captures source nodes. In this work, there are no source nodes.

In [10], the authors explain how to implement a secure clock synchronization protocol. Although the protocol is not self-stabilizing, we believe that some of their security primitives could be used in a self-stabilizing manner when implementing our self-stabilizing algorithm.

Herman and Zhang [16] present a self-stabilizing clock synchronization algorithm for sensor networks. The authors present a model for proving the correctness of synchronization algorithms and show that the converge-to-max approach is stabilizing. However, the converge-to-max approach is prone to attacks with a single captured node that introduces the maximal clock value whenever the adversary decides to attack. Thus, the adversary can at once set the clock values "far into the future", preventing the nodes from implementing a continuous time approximation function. This work is the first in the context

of self-stabilization to provide security solutions for clock synchronization in sensor networks.

2.7.1 Conclusions

Designing secure and self-stabilizing infrastructure for sensor networks narrows the gap between traditional networks and sensor networks by simplifying the design of future systems. In this work, we use system settings that consider many practical issues, and take a clean-slate approach in designing a fundamental component: a clock synchronization protocol.

The designers of sensor networks often implement clock synchronization protocols that assume the system settings of traditional networks. However, sensor networks often require fine-grained clock synchronization for which the traditional protocols are inappropriate.

Alternatively, when the designers do not assume traditional system settings, they turn to reinforcing the protocols with masking techniques. Thus, the designers assume that the adversary never violates the assumptions of the masking techniques, e.g., there are at most f captured and/or pulse-delay attacked nodes in a neighborhood at all times, for a setting where $3f + 1 \leq n$ must hold in the neighborhood. Since sensor networks reside in an unattended environment, the last assumption is unrealistic when considering long timespans.

Our design promotes self-defense capabilities once the system returns to following the original designer's assumptions. Interestingly, the self-stabilization design criteria provide an elegant way for designing secure autonomous systems.

2.7.2 Acknowledgments

This work would not have been possible without the contribution of Marina Papatriantafidou in many helpful discussions, ideas, and analysis. We wish to thank Ted Herman for many helpful discussions. Many thanks to Edna Oxman for improving the presentation.

Bibliography

- [1] Murat Demirbas, Anish Arora, Tina Nolte, and Nancy A. Lynch, “A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks.,” in *OPODIS*. 2004, vol. 3544 of *LNCS*, pp. 299–315, Springer.
- [2] Jeremy Elson, Lewis Girod, and Deborah Estrin, “Fine-grained network time synchronization using reference broadcasts,” *Operating Systems Review (ACM SIGOPS)*, vol. 36, no. SI, pp. 147–163, 2002.
- [3] Richard Karp, Jeremy Elson, Deborah Estrin, and Scott Shenker, “Optimal and global time synchronization in sensornets,” Tech. Rep., 2003.
- [4] Richard M. Karp, Jeremy Elson, Christos H. Papadimitriou, and Scott Shenker, “Global synchronization in sensornets.,” in *LATIN*. 2004, vol. 2976 of *LNCS*, pp. 609–624, Springer.
- [5] Kay Römer, Philipp Blum, and Lennart Meier, “Time synchronization and calibration in wireless sensor networks,” in *Handbook of Sensor Networks: Algorithms and Architectures*, pp. 199–237. John Wiley and Sons, Sep. 2005.
- [6] Saurabh Ganeriwal, Srdjan Capkun, Chih-Chieh Han, and Mani B. Srivastava, “Secure time synchronization service for sensor networks,” in *Proceedings of the 4th ACM workshop on Wireless security (WiSe’05)*, NYC, NY, USA, 2005, pp. 97–106, ACM Press.
- [7] Saurabh Ganeriwal, Srdjan Capkun, and Mani B. Srivastava, “Secure time synchronization in sensor networks,” *ACM Transactions on Information and Systems Security*, 2008.
- [8] Hui Song, Sencun Zhu, and Guohong Cao, “Attack-resilient time synchronization for wireless sensor networks.,” *Ad Hoc Networks*, vol. 5, no. 1, pp. 112–125, 2007.
- [9] Bruce Schneier, *Applied Cryptography*, John Wiley & Sons, 2nd edition, 1996.
- [10] Kun Sun, Peng Ning, and Cliff Wang, “Tinsersync: secure and resilient time synchronization in wireless sensor networks.,” in *ACM Conference on Computer and Communications Security*. 2006, pp. 264–277, ACM.
- [11] Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport, “Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks.,” in *OPODIS*. 2006, vol. 4305 of *LNCS*, pp. 215–229, Springer.

- [12] Edsger W. Dijkstra, “Self-stabilizing systems in spite of distributed control,” *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.
- [13] Shlomi Dolev, *Self-Stabilization*, MIT Press, March 2000.
- [14] Joffroy Beauquier and Synnöve Kekkonen-Moneta, “Fault tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors,” *International Journal of Systems Science*, vol. 28, no. 11, pp. 1177–1187, Nov. 1997.
- [15] Kay Römer, “Time synchronization in ad hoc networks,” in *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, NYC, NY, USA, 2001, pp. 173–182, ACM Press.
- [16] Ted Herman and Chen Zhang, “Best paper: Stabilizing clock synchronization for wireless sensor networks.,” in *SSS. 2006*, vol. 4280 of *LNCS*, pp. 335–349, Springer.
- [17] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer, “Tight bounds for clock synchronization,” in *28th ACM Symposium on Principles of Distributed Computing (PODC), Calgary, Canada*, August 2009, pp. 46–55.
- [18] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer, “Clock synchronization with bounded global and local skew,” in *49th Annual IEEE Symposium on Foundations of Computer Science (FOCS), Philadelphia, Pennsylvania, USA*, October 2008, pp. 509–518.
- [19] Philipp Sommer and Roger Wattenhofer, “Gradient clock synchronization in wireless sensor networks,” in *8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), San Francisco, USA*, April 2009.
- [20] N. Abramson et al., *The Aloha System*, Univ. of Hawaii, 1972.
- [21] Michael Mitzenmacher and Eli Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, New York, NY, USA, 2005.
- [22] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, Fabian Kuhn, and Calvin C. Newport, “The wireless synchronization problem,” in *PODC. 2009*, pp. 190–199, ACM.
- [23] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport, “Secure communication over radio channels,” in *PODC. 2008*, pp. 105–114, ACM.

- [24] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport, “Gossiping in a multi-channel radio network,” in *DISC. 2007*, vol. 4731 of *Lecture Notes in Computer Science*, pp. 208–222, Springer.
- [25] Farnaz Moradi and Asrin Javaheri, “Clock synchronization in sensor networks for civil security,” Technical report, Computer Science and Engineering, Chalmers University of technology, March 2009.
- [26] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease, “The byzantine generals problem.,” *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.
- [27] B. Rosner, “Percentage points for a generalized *esd* many-outlier procedure,” *Technometrics*, vol. 25, pp. 165–172, 1983.
- [28] Kun Sun, Peng Ning, and Cliff Wang, “Secure and resilient clock synchronization in wireless sensor networks,” *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 2, pp. 395–408, Feb. 2006.
- [29] Emerson Farrugia and Robert Simon, “An efficient and secure protocol for sensor network time synchronization,” *Journal of Systems and Software*, vol. 79, no. 2, pp. 147–162, 2006.
- [30] Michael Manzo, Tanya Roosta, and Shankar Sastry, “Time synchronization attacks in sensor networks,” in *Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks (SASN’05)*, NYC, NY, USA, 2005, pp. 107–116, ACM Press.
- [31] Kun Sun, Peng Ning, and Cliff Wang, “Fault-tolerant cluster-wise clock synchronization for wireless sensor networks,” *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 3, pp. 177–189, 2005.

PAPER II

Andreas Larsson, Philippas Tsigas

Self-stabilizing (k,r) -Clustering in Wireless Ad-hoc Networks with Multiple Paths

Technical Report no. 2010:06, Department of Computer Science and
Engineering, Chalmers University of Technology, Sweden, 2010.

This work will appear in the proceedings of The 31st International Conference on Distributed Computing Systems (ICDCS). June 2011, Minneapolis, Minnesota, USA. A preliminary version of this article appeared in the proceedings of 14th International Conference On Principles Of Distributed Systems (OPODIS). December 2010, Tozeur, Tunisia, LNCS 6490 pp. 79-82.

3

PAPER II: Self-stabilizing (k,r)-Clustering in Wireless Ad-hoc Networks with Multiple Paths

Wireless Ad-hoc networks are distributed systems that often reside in error-prone environments. Self-stabilization lets the system recover autonomously from an arbitrary state, making the system recover from errors and temporarily broken assumptions. Clustering nodes within ad-hoc networks can help forming backbones, facilitating routing, improving scaling, aggregating information, saving power and much more. We present the first self-stabilizing distributed (k,r)-clustering algorithm. A (k,r)-clustering assigns k cluster heads within r communication hops for all nodes in the network while trying to minimize the

total number of cluster heads. The algorithm uses synchronous communication rounds and uses multiple paths to different cluster heads for improved security, availability and fault tolerance. The algorithm assigns, when possible, at least k cluster heads to each node within $O(r)$ rounds from an arbitrary configuration. The set of cluster heads stabilizes, with high probability, to a local minimum within $O(gr \log n)$ rounds, where n is the size of the network and g is an upper bound on the number of nodes within $2r$ hops.

3.1 Introduction

Starting from an arbitrary state, self stabilizing algorithms let a system stabilize to, and stay in, a consistent state [1]. There are many reasons why a system could end up in an inconsistent state of some kind. Assumptions that algorithms rely on could temporarily be invalid. Memory content could be changed by radiation or other elements of harsh environments. Battery powered nodes could run out of batteries and new ones could be added to the network. It is often not feasible to manually configure large ad-hoc networks to recover from events like this. Self-stabilization is therefore often a desirable property of algorithms for ad-hoc networks. However, self-stabilization comes with increased costs, so a tradeoff is made. A self-stabilizing algorithm can never stop because you can not know when temporary faults occur, but it can converge to a result that holds as long as all assumptions hold. Furthermore, there are often overheads in the algorithm tied to the need to recover from arbitrary states. It can be added computations, increased size of messages or increased number of needed rounds to achieve something.

An algorithm for clustering nodes together in an ad-hoc network serves an important role. Back bones for efficient communication can be formed using cluster heads. Clusters can be used for routing messages. Cluster heads can be responsible for aggregating data, e.g. sensor readings in an ad-hoc sensor network, into reports to decrease the number of individual messages needed to rout through the network. Hierarchies of clusters on different levels can be used

for improved scaling of a large network. Nodes in a cluster could take turns doing energy costly tasks to save power over all.

Clustering is a well studied problem. Due to space constraints we point to the survey of the area with regard to wireless ad-hoc networks by Chen et al. in [2] for references to the area in general. We will focus on self-stabilization, redundancy and some security aspects. One way of clustering nodes in a network is for nodes to associate themselves with one or more cluster heads. In the (k,r) -clustering problem each node in the network should have at least k cluster heads within r communication hops away. This might not be possible for all nodes if the number of nodes within r hop from them is smaller than k . In such cases a best effort approach can be taken for getting as close to k cluster heads as possible for those nodes. The clustering should be achieved with as few cluster heads possible. To find the global minimum number of cluster heads is in general too hard, so algorithms provide an approximation. Assuming that the network allows k cluster heads for each node, the set of cluster heads forms a total (k,r) -dominating set in the network. In a *total* (k,r) -dominating set the nodes in the set also need to have k nodes in the set within r hops, in contrast to an ordinary (k,r) -dominating set in which this is only required for nodes not in the set.

There is a multitude of existing clustering algorithms for ad-hoc networks, of which a number is self-stabilizing. In [3], Johnen and Nguyen present a self-stabilizing $(1,1)$ -clustering algorithm that converges fast. Dolev and Tzachar tackle a lot of organizational problems in a self-stabilizing manner in [4]. As part of this work they present a self-stabilizing $(1,r)$ -clustering algorithm. Caron et al. present a self-stabilizing $(1,r)$ -clustering in [5] that takes weighted graphs into account.

There is a number of papers that do not have self-stabilization in mind. Fu et al. consider the $(k,1)$ -clustering problem in [6]. In [7] the full (k,r) -clustering problem is considered and both a centralized and a distributed algorithm for solving this problem are presented. Wu and Li also consider the full (k,r) -clustering in [8].

Other algorithms do not take the cluster head approach. In [9], sets of nodes that all can communicate directly with each other are grouped together without assigning any cluster heads. In this paper malicious nodes that try to disturb the protocol are also considered, but self-stabilization is not considered.

3.1.1 Our Contribution

We have constructed the first, to the best of our knowledge, self-stabilizing (k, r) -clustering algorithm for ad-hoc networks. The algorithm is based on synchronous rounds and makes sure that, within $O(r)$ rounds, all nodes have at least k cluster heads (or all nodes within r hops if a node has less than k nodes within r hops) using a deterministic scheme. A randomized scheme complements the deterministic scheme and lets the set of cluster heads stabilize to a local minimum. It stabilizes within $O(gr \log n)$ rounds with high probability, where g is a bound on the number of nodes within $2r$ hops, and n is the size of the network.

We prove quick selection of enough cluster heads. Once the system fulfills the cluster head requirements, of k cluster heads within r hops for all nodes, the requirements will continue to hold from that point on. We also prove that the set of cluster heads converges towards a local minimum. Under the extra assumption that timers are synchronized, we show an upper bound on the number of rounds it takes, with high probability, for the set of cluster heads to reach a local minimum. Furthermore, experimentally we show that without this extra assumption used in the proof the system stabilizes approximately equally fast.

Our contribution is presented as follows. In section 3.2 we introduce the system settings. Section 3.3 describes the algorithm. Section 3.4 proves the properties of the algorithm. We discuss experimental results, security and redundancy and how different system settings would affect the properties of the algorithm in Section 3.5.

3.2 System Settings

We assume a static network. Changes in the topology are seen as transient faults. We denote the set of all nodes in the network \mathcal{P} and the size of the network $n = |\mathcal{P}|$. We impose no restrictions on the network topology other than that an upper bound, g , on the number of nodes within $2r$ hops of any node is known (see below).

The set of neighbors, N_i , of a node p_i is all the nodes that can communicate directly with node p_i . In other words, a node $p_j \in N_i$ is one hop from node p_i . We assume a bidirectional connection graph, i.e. that $p_i \in N_j$ iff $p_j \in N_i$. The neighborhood, G_i^r of a node p_i is all the nodes (including itself) at most r hops away from p_i . Let $g \geq \max_j |G_j^{2r}|$ be a bound, known by the nodes, on the number of nodes within $2r$ hops.

The system is synchronous and progresses in rounds. Each round has two phases. First in the receipt phase each node p_i receives messages from all of its immediate neighbors $p_j \in N_i$. Then in the step phase each node p_i after performing the appropriate calculations broadcasts a message to all nodes $p_j \in N_i$. We assume that a broadcast by a node p_i is received reliably by all processors $p_j \in N_i$ in the receipt phase of the respective round.

3.3 Self-stabilizing (k, r) -clustering algorithm

The goal of the algorithm is, using as few cluster heads as possible, for each node p_i in the network to have a set of at least k cluster heads within its r -hop neighborhood G_i^r . This is not possible if a node p_i has $|G_i^r| < k$. Therefore, we require that $|C_i^r| \leq k_i$, where $C_i^r \subseteq G_i^r$ is the set of cluster heads in the neighborhood of p_i and $k_i = \min(k, |G_i^r|)$ is the closest number of cluster heads to k that node p_i can achieve. We do not strive for a global minimum. That is too costly. We achieve a local minimum, i.e. a set of cluster heads in which no cluster head can be removed without violating the (k, r) goal.

The basic idea is that each node p_i elects k_i nodes to be cluster heads. As all nodes elect cluster heads, the election process might lead to more than k_i nodes

Constants:

i : id of executing processor
 r : number of hops within we consider a neighborhood
 k : the number of clusterheads to elect
 g : upper bound on the number of nodes within $2r$ hop.
 $T = 8gr$: length of an escape period

Variables:

$state \in \{\text{HEAD}, \text{ESCAPING}, \text{SLAVE}\}$: The state of the node. Initially set to SLAVE.
 $timer$: Integer. Timer for escape attempts. Initially set to $T-1$.
 $estart$: Integer. The escape schedule. Initially set to 0.
 $estate \in \{\text{SLEEP}, \text{INIT}, \text{FLOOD}, \text{HOPE}\}$: State for escape attempts. Initially set to SLEEP.
 $heads$: Set of Id:s. Initially set to \emptyset .
 $S \ \& \ Z$: Sets of $\langle Id, State \rangle$ tuples. Initially set to $\{\langle i, state \rangle\}$.

External functions and macros:

$LBcast(m)$: Broadcasts message m to direct neighbors
 $LBrecv(m)$: Receives a message from direct neighbor
 $smallest(a, A)$: Returns the $\min(|A|, a)$ smallest id:s in A
 $cds(A)$: $\{\langle j, s, t \rangle \in A : t = \max_{\tau} \{\tau : \langle j, s, \tau \rangle \in A\}\}$
 $cdj(B)$: $\{\langle j, t \rangle \in B : t = \max_{\tau} \{\tau : \langle j, \tau \rangle \in B\}\}$

Figure 3.1: Constants, variables, external functions and macros for the algorithm in Fig. 3.2

being cluster heads within G_i^r . Moreover, for a node p_i that is a cluster head we might get that $\min_{p_j \in G_i^r} |C_j^r| > k$. In such a situation node p_i could possibly *escape*, i.e. drop the cluster head responsibility, without violating the condition that $|C_j^r| \geq k_j$ for any node p_j . One could imagine an algorithm that in a first phase adds cluster heads and thereafter in a second phase removes cluster heads that are not needed. To achieve self-stabilization however, we cannot rely on starting in a predefined state. Recovery from an inconsistent state might start at any time. Therefore, in our algorithm there are no phases and the mechanism for adding cluster heads runs in parallel with the mechanism for removing cluster heads and none of them ever stops. In each round each node sends out its state and forwards states of others. A cluster head node normally has the state HEAD and a non cluster head node always has state SLAVE.

If a node p_i in any round finds out that it has less than k cluster heads it selects a set of other nodes that it decides to elect as cluster heads. Node p_i then elects established cluster head nodes and any newly picked nodes by sending

```

1 on step phase:
2   if timer < 0 ∨ timer ≥ T-1
3     timer ← 0
4   else
5     timer ← timer + 1
6   S ← Z
7   heads ← {j : <j, HEAD> ∈ S}
8   /* Escaping */
9   if state in {HEAD, ESCAPING}
10    updatestate()
11    if estate = INIT ∧ state = HEAD ∧ |heads| > k
12      state ← ESCAPING
13      heads ← heads \ {i}
14    else if estate = SLEEP ∧ state = ESCAPING
15      state ← SLAVE
16    if state = SLAVE
17      estate ← SLEEP
18      estart ← 0
19    /* Add heads */
20    if |heads| < k
21      let a = k - |heads|
22      let A = {j : <j, ·> ∈ S} \ heads
23      heads ← heads ∪ smallest(a, A)
24    /* Join and send state */
25    for each j ∈ heads
26      if j ≠ i
27        forwardjoin(<j, r>)
28      else
29        state ← HEAD
30        Z ← {<i, state>}
31        sendstate(<i, state, r>)
32
33 function updateestate:
34   if timer = 0
35     estart ← uniformlyrandom({0, 1, . . . T-2r-2})
36   if timer ∈ [0, estart-1]:
37     estate ← SLEEP
38   else if timer ∈ [estart, estart]
39     estate ← INIT
40   else if timer ∈ [estart+1, estart+2r-1]
41     estate ← FLOOD
42   else if timer ∈ [estart+2r, estart+2r]
43     estate ← HOPE
44   else if timer ∈ [estart+2r+1, T-1]
45     estate ← SLEEP
46
47 function receivedstate(<j, jstate, ttl>), i ≠ j:
48   js ← jstate
49   if js = ESCAPING ∧ j ∈ heads
50     if |heads| ≤ k
51       js ← HEAD
52     else
53       heads ← heads \ {j}
54   let ss = {s : <j, s> ∈ Z} ∪ {js}
55   if HEAD ∈ ss:
56     js ← HEAD
57   else if ESCAPING ∈ ss
58     js ← ESCAPING
59   else
60     js ← SLAVE
61   Z ← {<o, s> : <o, s> ∈ Z ∧ o ≠ j} ∪ {<j, js>}
62
63   ttl ← max(1, min(r, ttl))
64   if ttl > 1:
65     forwardstate(<j, jstate, ttl-1>)
66
67 function receivedjoin(<j, ttl>):
68   ttl ← max(0, min(r, ttl))
69   if j = i ∧ estate ∉ {INIT, FLOOD}
70     state ← HEAD
71   else if ttl > 1
72     forwardjoin(<j, ttl-1>)
73
74 on LBrecv(<j, jstateset, jjoinset>):
75   for each <o, ostate, ottl> ∈ jstateset
76     if o ≠ i:
77       receivedstate(<o, ostate, ottl>)
78   for each <o, ostate, ottl> ∈ jjoinset
79     receivedjoin(<o, ottl>)
80
81 function forwardstate(tuple):
82   stateset ← stateset ∪ tuple
83
84 function forwardjoin(tuple):
85   joinset ← joinset ∪ tuple
86
87 function sendstate(tuple):
88   forwardstate(tuple)
89   stateset ← cds(stateset)
90   joinset ← cdj(joinset)
91   LBcast(<i, stateset, joinset>)
92   stateset ← ∅
93   joinset ← ∅

```

Figure 3.2: Pseudocode for the clustering algorithm.

a *join* message to them. Any node that is not a cluster head becomes a cluster head if it receives a join addressed to it.

We take a randomized approach for letting nodes try to drop their cluster head responsibility. Time is divided into periods of T rounds. A cluster head

node p_i picks uniformly at random one round out of the $T - 2r - 1$ first rounds in the period as a possible starting round, $estart_i$, for an escape attempt. If p_i has more than k cluster heads in round $estart_i$, then it will start an escape attempt. When starting an escape attempt a node sets its state to ESCAPING and keeps it that way for a number of rounds to make sure that all the nodes in G_i^r will eventually know that it tries to escape. A node $p_j \in G_i^r$ that would get fewer than k cluster heads if p_i would stop being a cluster head can veto against the escape attempt. This is done by recording the state of p_i as HEAD and thus continuing to send joins addressed to it. If p_j , on the other hand, has more than k cluster heads it would not need to veto. Thus, by accepting the state of p_i as ESCAPING, p_j will not send any join to p_i . After a number of rounds all nodes $G_i \setminus \{i\}$ will have had the opportunity to veto the escape attempt. If none of them objected, at that point p_i will get no joins and can set its state to SLAVE.

If an escape attempt by p_i does not overlap in time with another escape attempt it will succeed if and only if $\min_{p_j \in G_i^r} |C_j^r| > k$. If there are overlaps by other escape attempts, the escape attempt by p_i might fail even in cases where $\min_{p_j \in G_i^r} |C_j^r| > k$. The random escape attempt schedule therefore aims to minimize the risk of overlapping attempts.

The pseudocode for the algorithm is described in Fig. 3.2 with accompanying constants, variables, external functions and macros in Fig 3.1. In the step phase of each round lines 1-31 are executed. The code for the receipt phase can be found in lines 47-72. To have only one message for each node sent per round, all forwarding and sending of messages in lines 1-72 use the functions in lines 74-93 that collect everything that is to be sent until the end of the step phase where one message is sent out.

3.4 Correctness

In Section 3.4.1 we will show that within $O(r)$ rounds we will have $|C_i^r| \geq k_i$ for any node p_i . First we show that this holds while temporarily disregarding the escaping mechanism, and then that it holds for the general case in Theorem 3.1.

In Section 3.4.2 we will show that a cluster head node p_i can become slave if it is not needed and if it tries to escape undisturbed by other nodes in G_i^{2r} . We continue to show that the set of nodes converges, with high probability, to a local minimum within $O(gr \log n)$ rounds under the assumption that the timers of all nodes in the network are synchronized in Theorem 3.2.

Finally we present the message complexity in Theorem 3.3.

Definition 3.1 *If all assumptions about the network hold and all nodes follow the protocol throughout the entire round s then round s is called a legal round.*

Definition 3.2 *For a node p_i to be a cluster head is equivalent to $state_i \in \{HEAD, ESCAPING\}$. For a node p_i to be a slave is equivalent to $state_i = SLAVE$. For a node p_j , we define C_j^r as the set of cluster heads in G_j^r . Furthermore, we define H_x to be the set of cluster heads in the network in a round x .*

Definition 3.3 *A node initiates an escape attempt in round s if lines 12-13 are executed in round s . In other words in round s node p_i has $state_i = HEAD$ at line 1, $|heads_i| > k$ after executing line 7 and then line 10 sets $estate_i$ to INIT. Thereafter, the condition holds at line 11 and lines 12-13 are executed.*

3.4.1 Getting enough cluster heads

We begin by showing that nodes get to learn their neighborhood, G_i .

Lemma 3.1 *Assume that round s and all following rounds are legal. For any node p_i , $\{p_j : \langle p_j, \cdot \rangle \in S_i\} = G_i^r$ holds in the step phase of round $s + r$ and throughout rounds $s + r + 1 + t$ for any non-negative t .*

Proof: In every round any node p_j broadcasts its id and state (line 31) with tll set to r . The tll is a *time to live* value that denotes how many more hops a message should be forwarded and is decreased by one every time the associated message is received. When tll reaches 1 the message is not forwarded any more. Therefore, during the following r rounds the id and the state is forwarded r hops away (lines 63-65). Consider a node $p_j \in G_i^r$ ($i \neq j$) that is \hat{r} hops away from

p_i . At round $s+r+t$ node p_i gets the id and state message that originated from node p_j at round $s+r-\hat{r}+t$. As t is non-negative and $\hat{r} \leq r$ we know that p_j sent a message with its state and id at round $s+r-\hat{r}+t \geq s$. For node p_i itself: (1) it adds itself to Z during each round (line 30), and (2) the only other line that could change Z is line 61 and is not executed in case $j = i$ and (3) S is set to Z at the beginning of the step phase. Therefore $G_i^r \subseteq \{p_j : \langle p_j, \cdot \rangle \in S_i\}$ in the step phase of round $s+r$ and thereafter.

A message with id and state of a node $p_j \notin G_i^r$ that is being received by some node p_i in round s could potentially lead to an id in S_i that is not in G_i^r . However such a message can not be sent out in round s with a *t*tl greater than $t-1$ (lines 63-65) and Z is cleared from nodes $p_j \neq p_i$ in every round in line 30. Therefore p_j could reach p_i in rounds $s+1$ to $s+r-1$, but not as late as $s+r+t$ for a non-negative t . Therefore $G_i^r \supseteq \{p_j : \langle p_j, \cdot \rangle \in S_i\}$ in the step phase of round $s+r$ and thereafter. ■

We continue with showing nodes within r hops get to know the state of a node that stays in that state.

Lemma 3.2 *If a node p_i has the same state σ in rounds s to $s+r-1$, then any node $p_j \in G_i^r \setminus \{p_i\}$ will receive the state σ and only state σ for p_i in round $s+r$.*

Proof: Node p_i sends out its state with a *t*tl of r in each round (line 31). Nodes that receive this state message with a *t*tl greater than 1 will forward the state with a *t*tl of one less (lines 64-65). Thus a message from p_i originating in round $s-t$ (for a positive t) can possibly be received by nodes in G_i^r in the rounds $s-t+1$ to $s+r-t$, but not in round $s+r$ as that would need an original *t*tl of $r+t$. Furthermore a state σ' sent in round $s+r-1+t$ (for a positive t) can be received earliest in round $s+r+t$. Thus only states sent in rounds s to $s+r-1$ can be received by a node $p_j \in G_i^r$ in round $s+r$.

Now consider any node $p_j \in G_i^r \setminus \{p_i\}$. Let $\hat{r} \in [1, r]$ be the smallest number of hops between p_i and p_j . By the Lemma statement node p_i sends out state σ in round $s+r-\hat{r}$. That message is forwarded one step each round and \hat{r} rounds later in round $s+r$ it reaches node p_j . ■

We now look how the addition of cluster heads work while temporarily disregarding the escaping mechanism.

Lemma 3.3 *Let round s and all following rounds be legal. Assume that the state of a node can never be ESCAPING, estate is always SLEEP and that lines 8-18 are not going to be executed. With these assumption after round $s + 2r + t$ for a non-negative t any node p_i will have k_i cluster heads within r hops.*

Proof: The limiting assumptions leave only one way for the state to change, namely by the execution of line 70 where state is set to HEAD.

From Lemma 3.1 we know that in round $s+r$, at the latest, node p_i will have all nodes in G_i^r in S_i . We also know that $|G_i^r| \geq k_i$. Let's look at round $s + r$. At line 20, $heads_i$ might already contain nodes. We have the one case where $|heads_i| \geq k \geq k_i$ already and one case where $|heads_i| < k$. In the second case lines 21-23 will be executed. Out of the set A of nodes in G_i^r that are not in $heads_i$, the smallest $\min(|A|, k - |heads_i|)$ nodes will be added to $heads_i$ in line 23. Thus after execution of line 23 $heads_i$ will contain $\min(|G_i^r|, k) = k_i$ nodes and at line 25 $|heads_i| \geq k_i$.

For each node $p_j \in heads_i$ either a join message with ttl r is sent out (at line 27, when $j \neq i$) or the state is set to HEAD directly (at line 29, when $j = i$). For the nodes $p_j \neq p_i$ the join messages are forwarded (line 72) to all nodes in G_i^r within r hops in r rounds (in a similar fashion as forwarded state as discussed in the proof of Lemma 3.1).

Each node $p_j \in heads_i$ thus gets a join addressed to itself at the latest in round $s + 2r$ and it will become a cluster head by setting its state to HEAD at line 70. Thus after round $s + 2r$ any node p_i will have k_i cluster heads within r hops. ■

Now we consider the full escape mechanisms and show that a node that get joins become a cluster head.

Lemma 3.4 *Consider a node p_i that receives a join during the receipt phase of the legal round z that follows the legal round $z - 1$. Then node p_i is a cluster*

head at the end of round z . Furthermore, if node p_i is a cluster at the end of round $z - 1$ then it is a cluster head throughout the entire round z .

Proof: Let σ be $state_i$ and e be $estate_i$ at the reception of a join from any node in a legal round z which follows a legal round $z - 1$. We begin by showing that the only thing that can happen with $state_i$ during the receipt phase of round z is for it to either change to HEAD or to stay HEAD or ESCAPING. We have four different cases for different e and σ .

Case 1 $e \in \{\text{INIT}, \text{FLOOD}\} \wedge \sigma = \text{SLAVE}$: This cannot happen as (1) node p_i in the previous round (the legal round $z - 1$) could not have $state_i = \text{SLAVE}$ without having executed line 17 that sets $estate_i$ to SLEEP and (2) there is no way for $estate_i$ to change during the receipt phase of a round.

Case 2 $e \in \{\text{INIT}, \text{FLOOD}\} \wedge \sigma = \text{HEAD}$: No change to $state_i$, that remains HEAD.

Case 3 $e \in \{\text{INIT}, \text{FLOOD}\} \wedge \sigma = \text{ESCAPING}$: No change to $state_i$, that remains ESCAPING.

Case 4 $e \notin \{\text{INIT}, \text{FLOOD}\}$: Here $state_i$ is set to HEAD.

Furthermore, the only way for $state_i$ to be ESCAPING at the start of the step phase of round z is if $e \in \{\text{INIT}, \text{FLOOD}\}$. In that case $estate_i$ must have been set to FLOOD after line 10 in round $z - 1$ from which it follows that $estate_i \in \{\text{FLOOD}, \text{HOPE}\}$ after execution of line 10 in round z . Thus, the condition in line 14 does not hold in round z and line 15, the only line that can set $state_i$ to SLAVE, is not executed. Therefore, node p_i is a cluster head at the end of round z and if it were a cluster head at the beginning of round z it was so throughout the round. ■

Lemma 3.5 *Let s and all following rounds be legal rounds and assume a node p_j wants a node $p_i \in G_j^r$ to be cluster head as soon as it knows about it and is never willing to let it escape. In other words (1) if $p_i \notin heads_j$ after line 7 the condition in line 21 would always hold and $p_i \in A$ after executing line 22 and (2) the condition in line 50 would always hold.*

Then node p_i will be a cluster head after round $y \leq s + 2r$ and throughout all following rounds.

Proof: Let \hat{r} be the number of hops between p_i and p_j and let round x be the first round $\geq s$ in which node p_j receives a state from p_i . We know that $s \leq x \leq s + \hat{r}$. Furthermore, let y be the round in which p_i gets the join from p_j that was sent in round x . We know that $y = x + \hat{r}$ and thus $s + 1 \leq y \leq s + 2\hat{r}$ and thus both round $x - 1$ and x are legal. According to Lemma 3.4, p_i will be a cluster head at the end of round y .

According to the assumptions, $p_i \in heads_j$ at line 25 in every round $\geq x$ and thus p_j sends a join to p_i in every such round. This means that p_i will receive a join in every round $\geq y$, and thus, by Lemma 3.4, be a cluster head in the step phase of round y and throughout the following rounds. ■

Now we can show that within $2r + 1$ legal rounds from an arbitrary configuration all nodes p_i have at least k_i cluster heads and that the set of cluster heads in the network can only stay the same or shrink from that point on.

Theorem 3.1 *Let round s and all following rounds be legal. Then any node p_j will have k_j cluster heads within r hops in the step phase of round $s + 2r$ and throughout any following rounds.*

Moreover, a node that is not in H_x in a round $x \geq s + 2r$ can not be in H_{x+t} for a non-negative t and consequently $|H_{x+t}| \leq |H_x|$.

Proof: From Lemma 3.3 we have seen that as long as the escape mechanism does not allow nodes to change its state to SLAVE after being a cluster head, any node p_j will have k_j cluster heads within r hops in the step phase of round $s + 2r$ and throughout any following rounds.

Furthermore, from Lemma 3.5 and its proof we have seen that as long as a node p_j wants to have node p_i as a cluster head p_i will remain a cluster head. Now we will look in what situations p_j does not want p_i as a cluster head even though it did at some earlier point in time.

If $|heads_j| < k$ at line 20 in a round s , then node p_j finds up to $k - |heads_j|$ nodes in $\{p_i : < p_i, \cdot > \in S_j\} \setminus heads_j$ and sends a join to them in a round s . Assume $p_i \in G_j^r$ is one of the newly picked nodes in A after executing line 22 in round s . We call this set A in round s for \hat{A} . Node p_i does not get the join until round $s + \hat{r}$ where \hat{r} is the number of hops between nodes p_i and p_j .

As we saw in the proof of Lemma 3.4, if $state_i = \text{ESCAPING} \wedge estate_i \in \{\text{INIT}, \text{FLOOD}\}$ does not hold in round $s + \hat{r}$ then p_i will send out HEAD. That will reach p_j in round $s + 2\hat{r}$ and consequently $p_i \in heads_j$ in round $s + 2\hat{r}$. If $state_i = \text{ESCAPING}$ and $estate_i \in \{\text{INIT}, \text{FLOOD}\}$ in round $s + \hat{r}$, node p_i will not send out HEAD in that round and p_j might not get HEAD from p_i in round $s + 2\hat{r}$. If p_j got HEAD from some other node $p_l \in G_j^r$ in a round $x \in [s + 1, s + 2\hat{r}]$ node p_j might not want p_i as a cluster head any more. Node p_j will not send join to p_i in round x if (1) $|heads_j| \geq k$ at line 20 or (2) p_i has received HEAD from enough nodes not in \hat{A} so that p_i is not among the smallest nodes picked out in line 22 in round $s + 2\hat{r}$. On the other hand if none of these cases hold p_j will continue to send joins to p_i and by Lemma 3.5 node p_j will remain a cluster head.

The second way for a node $p_i \in G_j^r$ to be SLAVE even though it earlier were in $heads_j$ is to escape using the escape mechanism. In other words in some round z node p_i initiates an escape attempt. When receiving different states for a node, HEAD takes precedence over ESCAPING that takes precedence over SLAVE (lines 55-60). This combined with Lemma 3.2 means that in some round $y \in [z + 1, z + r]$ node p_i get the state ESCAPING from p_i and that in the previous round $y - 1$ p_j got HEAD from p_j .

Node p_j will have $p_i \in heads_j$ after executing line 7 in round $y - 1$ as p_j receives HEAD from p_i in round $y - 1$. Thus $p_i \in heads_j$ at line 47 in round y when p_j receives ESCAPING from p_i . If $|heads_j| \leq k$ at that point then p_j will interpret the state as HEAD for all purposes other than forwarding the message (lines 50-51). Thus $p_i \in heads_j$ after executing line 7 in round y as well, and p_j will send a join. By Lemma 3.5 and its proof, node p_i will remain cluster head in that case. If on the other hand $|heads_j| > k$ for node p_j at line 47 in round y then p_j removes p_i from heads and will consequently not send any join in round y . When p_j gets ESCAPING from p_i in the coming rounds $y + 1, y + 2, \dots$, then p_i will not be in $heads_j$ and thus no joins will be sent in those rounds either. If node p_j receives ESCAPING for more than one node $p_l \in heads_j$ in rounds $y, y + 1, \dots$ then p_j will let them go in first come first served fashion. When node p_j decides to let a node p_l go it is immediately

removed from $heads_j$ in line 53. Thus, node p_i will not let so many nodes go that $|heads_j| \geq k$ would not be fulfilled (if $k_j < k$ no node is ever allowed to go).

Finally, a node $p_i \notin G_j^r$ might be in $heads_i$ in a round $z \in [s, s + r - 1]$ but by Lemma 3.1 such a node is not in S_j in round $s + r$ and thus node p_i will pick some other node instead of such a p_j to send join to in round $s + r$ if not earlier.

So any node p_j will have k_j cluster heads within r hops in the step phase of round $s + 2r$ and throughout any following rounds. Therefore in any of these rounds no node will fulfill the condition in line 20. Hence, no node p_i that is not a cluster head at the beginning of the state phase of round $s + 2r$ can be picked by any node p_j in line 22. Therefore no such node p_i can become a cluster head in the state phase of round of round $s + 2r$ or thereafter.

Thus a node that is not in set of cluster heads in the entire network at round x , H_x , for a round $x \geq s + 2r$ can never be in H_{x+t} for a non-negative t . Moreover, $|H_{x+t}| \leq |H_x|$ for any $x \geq s + 2r$ and any non-negative t . ■

3.4.2 Convergence to a local minimum

In this section we show that the set of cluster heads converges to a local minimum.

Lemma 3.6 *Consider a round s for which all rounds from $s - 2r - 1$ and forward are legal. Assume that node p_i initiates an escape attempt in round s and assume that in rounds $[s, s + r]$ all nodes $p_j \in G_i^r$ have $|C_j^r| > k$. If no other node $p_l \in G_i^{2r}$ than node p_i initiates an escape attempt in any round $\in [s - 2r - 1, s + r - 1]$ then node p_i will set $state_i$ to SLAVE in round $s + 2r + 1$ and have $state_i = SLAVE$ throughout any round $s + 2r + 1 + t$ for a positive t .*

Proof: Assume that a node p_l initiates an escape attempt in round x . In round $x + 2r$ node p_l will set $estate_l$ to HOPE. In all rounds in $[x, x + 2r]$ node p_l will send out $state_l = ESCAPING$. If node p_l gets a join to itself in the receipt phase of round $x + 2r + 1$ it sets $state_i$ to HEAD in line 70. Otherwise p_l sets

$state_l$ to SLAVE in line 15 in round $x + 2r + 1$. Let σ be the $state_i$ that is sent out by p_l in round $x + 2r + 1$. We know that $\sigma \neq \text{ESCAPING}$. We assume that node p_l does not initiate any more escape attempts in the time span we are looking at. Therefore node p_l sends out σ in the rounds in $[x + 2r + 1, x + 3r]$. By Lemma 3.2, in round $x + 3r + 1$ all nodes $p_{j'} \in G_l^r \setminus \{p_l\}$ receives σ and only σ for p_l in the receipt phase. Therefore, either all nodes $p_j \in G_l^r$ (including p_l) have $p_l \in heads_j$ (if $\sigma = \text{HEAD}$) or none of them have $p_l \in heads_j$ (if $\sigma = \text{SLAVE}$) after executing line 7 in the step phase of round $x + 3r + 1$. This continues to hold in the receive phase of round $x + 3r + 2$. Thus in round $x + 3r + 1 + t$, for a positive t , no node $p_j \in G_l^r$ can have $p_l \in C_j^r$ without having $p_l \in heads_j$.

Now if node p_i initiates an escape attempt in round s , by Lemma 3.2, all nodes $p_j \in G_i^r$ will receive ESCAPING and only ESCAPING for node p_i in round $s + r$. As we saw above no node p_l initiating an escape attempt in a round $\leq s - 2r - 2$ can in round $s + r$ be in C_j^r , for a node $p_j \in G_l^r$, without being in $heads_j$. By the Lemma assumptions, no node $p_l \in G_i^{2r}$ makes an escape attempt in a round in $[s - 2r - 1, s + r - 1]$. In addition, consider a node p'_l that initiates an escape attempt in a round $s + r - 1 + t$ for a positive t . A node $p_j \in G_i^r$ can only receive ESCAPING from that escape attempt in rounds $\geq s + r + t$. Therefore a node $p_j \in G_i^r$ will in round $s + r$ receive ESCAPING for node p_i but not for any other node.

In rounds $[s, s + r]$ all nodes $p_j \in G_i^r \setminus \{p_i\}$ have $|C_j^r| > k$. Therefore, when p_j receives ESCAPING for p_i in round $s + r$ either (1) $p_i \notin heads_j$ because p_j received ESCAPING and had $|heads_j| > k$ in some round in $[s + 1, s + r - 1]$ or (2) $p_i \in heads_j$ and $|heads_j| > k$ in which case p_j removes p_i from $heads_j$ at line 53. Thus in the step phase of rounds in $[s + r, s + 2r]$ no node p_j sends a join to p_i . Therefore, in the round $s + 2r + 1$ no join is received by p_i and therefore p_i sets $state_i$ to SLAVE in round $s + 2r + 1$. There is no round $\geq s$ in which p_i sends out HEAD and, by Theorem 3.1, no node will need to add new nodes as cluster heads in any round $\geq s$. Hence, node p_i will have $state_i = \text{SLAVE}$ in the step phase of round $s + 2r + 1$ and throughout any round $s + 2r + 1 + t$ for a positive t . ■

Definition 3.4 We say that the timers of the nodes in the network are synchronized if $\text{timer}_i = \text{timer}_j$ for all pair of nodes $p_i, p_j \in \mathcal{P}$ for all legal rounds.

Lemma 3.7 Let round s and all following rounds be legal. Furthermore, let $g = \max_j |G_j^{2r}|$ be a bound on the number of nodes within $2r$ hops. Consider a node p_i that is a cluster head in any round $\geq s$. Assume that $|C_j^r| > k$ holds for all nodes $p_j \in G_i^r$ from round $s + 2r$ and as long as p_i remains a cluster head. If the timers of all nodes in the network are synchronized and $T = 8gr$, node p_i will be SLAVE in any round $s + 2r + 8(\beta + 1)gr - 2 + t$ for a non-negative t with probability at least $1 - 2^{-\beta}$.

Proof: From Theorem 3.1 we know that from round $s + 2r$ nodes can only go from being cluster heads to being slaves. Consider a cluster head node p_i . Let x_i^0 be the first round $\geq s + 2r$ in which $\text{timer}_i = 0$ at line 8. As long as node p_i remain a cluster head it will execute line 35 every round $x_i^t = x_i^0 + tT$, for a non-negative t and a given T , and schedule an escape attempt in the period $\Pi_i^t = [x_i + tT, x_i + (t + 1)T - 1]$. Node p_i picks one of the first $T - 2r - 1$ rounds in the period, uniformly at random and independently from any other random choice, to initiate an escape attempt in. Thus the probability that node p_i initiates an escape attempt in any given round is $\leq 1/(T - 2r - 1)$.

Now consider a period Π_i^t in which p_i initiates an escape attempt. Let D_i^t be the set of rounds $[x_i^t - 2r - 1, x_i^t + r - 1]$. The number of nodes that could be cluster heads in G_i^{2r} is bounded by g . If $F_{i,l}^t$ is the event that a node $p_l \in G_i^{2r}$ initiates an escape attempt in any round in D_i^t then $P[F_{i,l}^t] \leq (3r + 1)/(T - 2r - 1) =: \rho$. Let A_i^t be the event that none of the nodes in G_i^{2r} initiate an escape attempt in a round in D_i^t . We say that A_i^t is the event that node p_i gets

an *uninterfered escape attempt* in period Π_i^t . Then we get

$$P[A_i^t] \geq (1 - \rho)^{g-1} = [\mu := \frac{1}{\rho}] \quad (3.1)$$

$$= \left(\left(1 - \frac{1}{\mu} \right)^{\mu-1} \right)^{(g-1)/(\mu-1)} \quad (3.2)$$

$$> \left(\frac{1}{e} \right)^{(g-1)/(\mu-1)} = \exp \left(-\frac{g-1}{\mu-1} \right) \quad (3.3)$$

$$= \exp \left(-\frac{g-1}{\frac{T-2r-1}{3r+1} - 1} \right) \quad (3.4)$$

We want at least probability $1/2$ for event A and get

$$\frac{1}{2} \leq \exp \left(-\frac{g-1}{\frac{T-2r-1}{3r+1} - 1} \right) \quad (3.5)$$

$$\Rightarrow \frac{1}{2} \leq \exp \left(-\frac{(g-1)(3r+1)}{T-5r-2} \right) \quad (3.6)$$

$$\Rightarrow -\ln 2 \leq -\frac{(g-1)(3r+1)}{T-5r-2} \quad (3.7)$$

$$\Rightarrow T-5r-2 \geq \frac{-(g-1)(3r+1)}{-\ln 2} \quad (3.8)$$

$$\Rightarrow T \geq \frac{(g-1)(3r+1)}{\ln 2} + 5r + 2 =: \hat{T}. \quad (3.9)$$

We know that $1 \leq r$ and thus we get

$$\hat{T} = \frac{1}{\ln 2}(3gr - 3r + g - 1) + 5r + 2 \quad (3.10)$$

$$< 2(3gr - 3r + g - 1) + 5r + 2 \quad (3.11)$$

$$= 6gr - 6r + 2g - 2 + 5r + 2 \quad (3.12)$$

$$= 6gr + 2g - r \quad (3.13)$$

$$< 6gr + 2g \quad (3.14)$$

$$< [r \geq 1] < 8gr. \quad (3.15)$$

Therefore we can set

$$T = 8gr > \hat{T} \quad (3.16)$$

and get $P[A_i^t] > 1/2$.

According to the Lemma assumption the timers of all nodes in the network are synchronized. Thus we have a global $x^t = x_i^t$ and $\Pi^t = \Pi_i^t$ holding for all nodes $p_i \in \mathcal{P}$. Consider a period starting in round z . The earliest in a period a node p_i can initiate an escape attempt is in round z when $estart_i = 0$. The latest a node p_j could initiate an escape attempt in the period starting in $z - T$ is in round $(z - T) + (T - 2r - 2) = z - 2r - 2$. Thus by Lemma 3.6 an escape attempt initiated in an earlier period cannot affect an escape attempt in this period. The latest in a period a node p_i can initiate an escape attempt is in round $z + T - 2r - 2$ when $estart_i = T - 2r - 2$. However $z + T - 2r - 2 + r - 1 < T$ and therefore, by Lemma 3.6, no escape attempt in a later period could affect this period. This together with the fact that the random choices in different executions of the line 35 are all mutually independent would make what happens in different rounds mutually independent. However if a node p_i becomes SLAVE in a round t it is not doing an escape attempt in round $t + 1$ which only increases the probability for A_j^{t+1} for another node p_j . Therefore, by assuming independence the calculated lower bound on the probability of an undisturbed escape attempt gets worse.

Consider the β periods Π^0 to $\Pi^{\beta-1}$ and let $A_i = \bigcup_{t=0}^{\beta-1} A_i^t$. Thus with the assumption of period independence that gives us a worse bound we get

$$P[A_i] = P\left[\bigcup_{t=0}^{\beta-1} A_i^t\right] = 1 - P\left[\bigcap_{t=0}^{\beta-1} \bar{A}_i^t\right] \quad (3.17)$$

$$= 1 - \prod_{t=0}^{\beta-1} P[\bar{A}_i^t] > 1 - \prod_{t=0}^{\beta-1} \frac{1}{2} = 1 - 2^{-\beta}. \quad (3.18)$$

The latest period Π^0 could start is $s + 2r + T - 1$ in which case $\Pi^{\beta-1}$ ends in round $s + 2r + T - 1 + \beta T - 1 = s + 2r + 8(\beta + 1)gr - 2$. ■

Assuming that the timers of all nodes in the network are synchronized, we show that with at least probability $1 - 2^{-\alpha}$ the set of cluster heads in the network stabilizes to a local minimum within $O((\alpha + \log n)gr)$ rounds.

Theorem 3.2 *Let round s and all following rounds be legal. Consider round $f = s + 2r + 8(\alpha + \log n + 1)gr - 2$, where n is the number of nodes in the network. Assume that the timers of all nodes in the network are synchronized.*

Then, with at least probability $1 - 2^{-\alpha}$, in round f there will be no cluster head node p_i in the network for which $\min_{p_j \in G_i} |C_j^r| > k$ holds and $H_{f+t} = H_f$ holds for any positive t .

Proof: We use the notations Π^t , x^t and A_i and the concept of uninterfered escape attempts from the proof of Lemma 3.7.

Let $\beta = \alpha + \log n$, where $n = |\mathcal{P}|$. Let A be the event all nodes in the network get at least one uninterfered escape attempt in the periods Π^0 to $\Pi^{\beta-1}$. We get that

$$P[A] = 1 - P[\bar{A}] = 1 - P\left[\bigcup_{p_i \in \mathcal{P}} \bar{A}_i\right] \geq [\text{Boole's inequality}] \quad (3.19)$$

$$\geq 1 - \sum_{p_i \in \mathcal{P}} P[\bar{A}_i] \geq [\text{Lemma 3.7}] \geq 1 - \sum_{p_i \in \mathcal{P}} 2^{-\beta} \quad (3.20)$$

$$= 1 - n2^{-\beta} = 1 - 2^{\log n - \beta} = 1 - 2^{-\alpha}. \quad (3.21)$$

Thus by the proof of Lemma 3.7 all nodes in the network gets an uninterfered escape attempt with at least probability $1 - 2^{-\alpha}$ by the round $f = s + 2r + 8(\alpha + \log n + 1)gr - 2$. Together with Lemma 3.7 This concludes that with high probability all nodes p_i for which $|C_i^r| > k$ holds at their uninterfered escape attempt will have set $state_i$ to SLAVE by round f . From this follows that at round f there is no node for which $\min_{p_j \in G_i} |C_j^r| > k$ holds. Hence, by Lemma 3.1, no node p_i that is cluster head in round f can ever set $state_i$ to SLAVE in a round $\geq f$ and $H_{f+t} = H_f$ holds for any positive t . ■

At last we take a look at message complexity.

Theorem 3.3 *Let round s and all following rounds be legal. Then the size of the message sent by a node p_i in any round $\geq s + r$ is in $O(|G_i| \cdot \log n)$.*

Proof: By Lemma 3.1 we have that for any node p_i , $\{p_j : \langle p_j, \cdot \rangle \in S_i\} = G_i^r$ holds in the step phase of round $s+r$ and throughout rounds $s+r+1+t$ for

any non-negative t . Following the proof steps of that Lemma, we can conclude that the only nodes represented in $stateset_i$ and $joinset_i$ are the ones in G_i^r .

The only message a node p_i transmits in a round, is transmitted in line 91. Before that, $stateset_i$ is shrunk in line 89 so that, for every possible pair of node id j and state s , only the maximum TTL is kept. Similarly, $joinset_i$ is shrunk in line 90, so that for every possible node id j , only the maximum TTL is kept.

Each $stateset$ entry contains a node id which is encoded in $\log n$ bits, one of three possible states that is encoded in 2 bits and a TTL value that is encoded in $\log r$ bits. Each $joinset$ entry contains a node id and a TTL value. Thus the number of bits transmitted per node in G_i^r is in $O(\log n + \log r)$ which is in $O(\log n)$ as $r < n$. Therefore, the size of the message sent by a node p_i in any round $\geq s + r$ is in $O(|G_i| \cdot \log n)$. ■

3.5 Discussion

To see if the timers really need to be synchronized, we performed simulations of the algorithm for various settings of k and r and for different network densities. From this we can conclude that when $T = 8gr$ we get a much faster convergence than the upper bound we have proved. Setting T even lower decrease the convergence time even further. A representative picture of the general results can be seen in Fig. 3.3. The experiments show that when all $timer_j$ are independently and uniformly distributed in $[0, T - 1]$ at beginning of the experiment the convergence time is not far from what it is in the case with synchronized timers. We also see that if the nodes starts up with random information in their variables the convergence time is faster than for an initialized start where each node p_i does not know G_i and sets itself as cluster head in the first round. To conclude we can with good margin use the result of convergence within $O(gr \log n)$ rounds from Theorem 3.2 for the unsynchronized setting.

We performed some preliminary experiments regarding the global optimum, the local minima our algorithm finds and the worst possible local minimum for random graphs. For different choices for k and r our algorithm got on

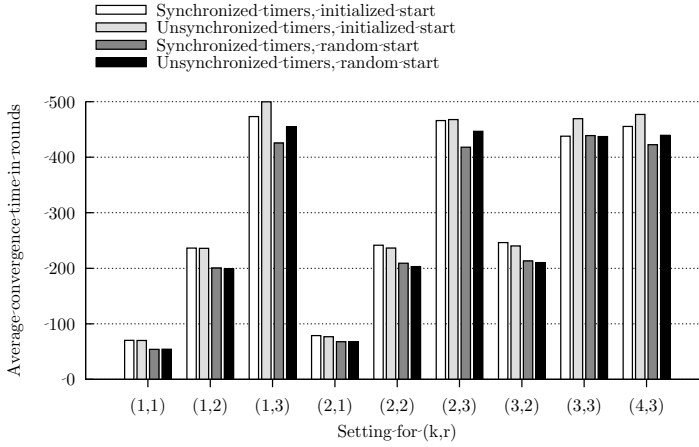


Figure 3.3: Simulation results of the algorithm when $T = gr/2$ indicating that synchronization of timers is not needed.

average 1.10-1.18 times the number of cluster heads compared to the optimum. The worst possible local minima contained on average 1.23-1.60 the number of cluster heads compared to the optimum.

The flooding of messages makes sure that if there exist multiple paths of at most length r between a node p_i and a node p_j then joins and state updates will traverse all possible paths. This can give us higher fault tolerance if there are communication disturbances on some links (i.e. between some immediate neighbors) and also higher availability for nodes to reach their cluster heads.

The multiple paths can also give us higher security if some nodes in the network can be compromised. If there is at least one path of at most r hops between a node p_i and a node p_j that is not passing through any compromised nodes then the flooding makes sure that node p_i and p_j gets to know about each other. Moreover, if p_j wants p_i to be cluster head then the compromised nodes cannot stop that. If nodes add information to the messages about the paths they have taken during message forwarding then the nodes get to know about the multiple paths. With this knowledge they can in an application layer use as

diverse paths as possible to communicate with their cluster heads. Thus even if a compromised node is on the path to one cluster head and drops messages or do other malicious behavior there can be other cluster heads for where there is no compromised nodes on the chosen paths.

Consider a compromised node p_c that can lie and not follow protocol. First assume that p_c cannot introduce node id:s that does not exist (Sybil attacks, [10]) or node id:s for nodes that are not within G_c^r (wormhole attacks, [11]) and that p_c cannot do denial of service attacks. Then p_c can make any or all nodes within G_c^r become and stay cluster heads by sending joins to them. Consider a node p_i that is a cluster head and has a path to a node p_j of length $\leq r$ hops that does not pass through p_c . In this situation p_c can not give the false impression that p_i is not a cluster head as HEAD takes precedence over ESCAPING that takes precedence over SLAVE at message receipt. If p_c on the other hand is in a bottle neck between nodes without any other paths between them then it can lie about a node p_l being a cluster heads and refuse to forward any joins to p_l . Now if we assume that p_c is not restricted in what id:s it can include in false messages it can convince a node p_l that nodes not in G_l^r are cluster heads. In the worst case it can eventually make p_l rely exclusively on non-existent cluster heads with paths that all go through p_c . In any case the influence by a compromised node p_c is contained within G_c^r as the maximum *ttl* of a message is r and is enforced at message receipt.

The flooding of messages might make the algorithm too expensive in some sensor networks with limited battery power. If that is the case the algorithm might be run on an overlay network for which the flooding becomes much cheaper. For instance a self-stabilizing spanning tree algorithm like the one in [12] might be used to set up this overlay network. This on the other hand effectively removes all the pros of having multiple paths so it is a trade off between redundant paths and message costs.

3.5.1 Conclusions

We have presented the first self-stabilizing (k, r) -clustering algorithm for ad-hoc networks. A deterministic mechanism guarantees that all nodes, if possible for the given topology, have k cluster heads within r hops. A randomized mechanism lets the set of cluster heads stabilize to a local minimum. We have shown, under the extra assumption of synchronized timers, that the set of cluster heads converges, with high probability, to a local minimum within $O(gr \log n)$ rounds, where g is an upper bound on number of nodes within $2r$ hops, and n is the size of the network. With simulations we have shown that even without this extra assumption the system converges much faster than the proven bounds and that g does not have to be known very accurately. We have also discussed how the algorithm can help us with fault tolerance and security and that the algorithm can be run on an overlay network, e.g. a spanning tree, if message costs needs to be reduced.

Bibliography

- [1] Shlomi Dolev, *Self-Stabilization*, MIT Press, March 2000.
- [2] Yuanzhu Peter Chen, Arthur L. Liestman, and Jiangchuan Liu, *Clustering Algorithms for Ad Hoc Wireless Networks*, vol. 2, chapter 7, pp. 154–164, Nova Science Publishers, 2004.
- [3] Colette Johnen and Le Huy Nguyen, “Robust self-stabilizing weight-based clustering algorithm,” *Theor. Comput. Sci.*, vol. 410, no. 6-7, pp. 581–594, 2009.
- [4] Shlomi Dolev and Nir Tzachar, “Empire of colonies: Self-stabilizing and self-organizing distributed algorithm,” *Theor. Comput. Sci.*, vol. 410, no. 6-7, pp. 514–532, 2009.
- [5] Eddy Caron, Ajoy Kumar Datta, Benjamin Depardon, and Lawrence L. Larmore, “A self-stabilizing k -clustering algorithm using an arbitrary metric,” in *Euro-Par*, 2009, pp. 602–614.
- [6] Yongsheng Fu, Xinyu Wang, and Shanping Li, “Construction k -dominating set with multiple relaying technique in wireless mobile ad hoc networks,” in *CMC '09: Proceedings of the 2009 WRI International Conference on Communications*

- and Mobile Computing*, Washington, DC, USA, 2009, pp. 42–46, IEEE Computer Society.
- [7] Marco Aurélio Spohn and J. J. Garcia-Luna-Aceves, “Bounded-distance multi-clusterhead formation in wireless ad hoc networks,” *Ad Hoc Netw.*, vol. 5, no. 4, pp. 504–530, 2007.
- [8] Yiwei Wu and Yingshu Li, “Construction algorithms for k-connected m-dominating sets in wireless sensor networks,” in *MobiHoc '08: Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, New York, NY, USA, 2008, pp. 83–90, ACM.
- [9] Kun Sun, Pai Peng, Peng Ning, and Cliff Wang, “Secure distributed cluster formation in wireless sensor networks,” in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, Washington, DC, USA, 2006, pp. 131–140, IEEE Computer Society.
- [10] James Newsome, Elaine Shi, Dawn Song, and Adrian Perrig, “The sybil attack in sensor networks: analysis & defenses,” in *IPSN '04: Proceedings of the 3rd international symposium on Information processing in sensor networks*, New York, NY, USA, 2004, pp. 259–268, ACM.
- [11] Yih chun Hu, Adrian Perrig, and David B. Johnson, “Wormhole detection in wireless ad hoc networks,” Tech. Rep., Rice University, Department of Computer Science, 2002.
- [12] Sudhanshu Aggarwal and Shay Kutten, “Time optimal self-stabilizing spanning tree algorithms,” in *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, London, UK, 1993, pp. 400–410, Springer-Verlag.