THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

# Security and Self-stabilization in Sensor Network Services

ANDREAS LARSSON

*Division of Networks and Systems*
*Department of Computer Science and Engineering*
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012

**Security and Self-stabilization in Sensor Network Services**
*Andreas Larsson*
ISBN: 978-91-7385-717-8

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3398
ISSN: 0346-718X

Technical report 84D
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 2012

# Security and Self-stabilization in Sensor Network Services

Andreas Larsson
*Division of Networks and Systems, Chalmers University of Technology*

## ABSTRACT

Wireless sensor networks consist of small sensor nodes that monitor their environment and that together can cover vast geographical regions. It is a promising field with many possible applications in different application areas. Typically the nodes do not have any preexisting information about the network topology when deployed. Instead, they gather information about their local topology by exchanging information with the other nodes in their vicinity, using wireless communication. Using this information they can organize themselves according to the needs of the applications of the network. Sensor nodes are often very limited in computing power, memory and battery life. They have traffic patterns different from those of many other types of networks. New algorithms are often needed to suit these conditions. For networks that consist of a very large number of nodes, algorithms have to scale well.

Security and fault tolerance are of high importance for many sensor network applications. A sensor network application needs to remain functioning even when nodes fail 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 in 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 a 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 clock synchronization algorithm for sensor networks that is resilient against attacks from outside as well as by compromised nodes from inside. Sensor nodes also need to organize their own network. A common way is to cluster the nodes together into groups. They are used by many applications and other fundamental services. We present a self-stabilizing algorithm for clustering. It uses redundant paths to be resilient against captured nodes in the network. It assumes perfect message transfers and lock step synchronization of the nodes. In addition, we present a clustering algorithm, that is a further development of that work, that can handle unreliable communication media and unsynchronized nodes, assuming a limit on clock rate differences.

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

# Preface

This thesis is based on the work contained in the following publications:

- ▷ Jaap-Henk Hoepman, **Andreas Larsson**, Elad M. Schiller, and Philippas Tsigas. “Secure and self-stabilizing clock synchronization in sensor networks.” In *Proceedings of the 9th International Symposium on Self Stabilization, Safety, And Security of Distributed Systems (SSS 2007)*. volume 4838 of *Lecture Notes in Computer Science*, pages 340–356, Springer, 2007.
- ▷ Jaap-Henk Hoepman, **Andreas Larsson**, Elad M. Schiller, and Philippas Tsigas. “Secure and self-stabilizing clock synchronization in sensor networks.” *Theoretical Computer Science*, volume 412, number 40, pages 5631–5647, 2011.
- ▷ **Andreas Larsson** and Philippas Tsigas. “Self-stabilizing (k,r)-clustering in wireless ad-hoc networks with multiple paths.” In *Proceedings of the 14th International Conference On Principles Of Distributed Systems (OPODIS 2010)*, volume 6490 of *Lecture Notes in Computer Science*, pages 79–82, Springer, 2010.
- ▷ **Andreas Larsson** and Philippas Tsigas. “A self-stabilizing (k,r)-clustering algorithm with multiple paths for wireless ad-hoc networks.” In *Proceedings of the 31st International Conference on*

*Distributed Computing Systems (ICDCS 2011)*, pages 353–362, IEEE Computer Society, 2011.

▷ **Andreas Larsson** and Philippas Tsigas. “Self-stabilizing (k,r)-clustering in clock rate-limited systems.” In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2012)*, volume ?? of *Lecture Notes in Computer Science*, pages ??–??, Springer, 2012 (To appear).

*To Malin, Elvira, Anne-Marie and Leif-Göran*

# Acknowledgments

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 Sébastien Tixeuil from Pierre and Marie Curie University as my opponent. 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 Oliver Theel from the University of Oldenburg for being the discussion leader for my licentiate thesis.

I thank Daniel Cederman for discussions, support and fun times. I want to give my appreciation to the rest of the colleagues in the Distributed Computing and Systems group: Bapi Chatterjee, Farnaz Moradi, Georgios Georgiadis, Magnus Almgren, Marina Papatriantafilou, Nhan Nguyen Dang, Olaf Landsiedel, Valentin Tudor, Yiannis Nikolakopoulos 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 colleagues, outside of the Distributed Computing and Systems group, that are involved or previously were involved in the Security Arena project and in the SysSec project: Anna Gryszkiewicz, Erland Jonsson, Fang Chen, Laleh Pirzadeh, Morten Fjeld, Staffan Björk, Tomas Olovsson, Ulf Larson and Vilhelm Verendel.

I thank the 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 thank the Seventh Framework Program for funding the SysSec project. I thank all the other members of the project and the expert group members. I also would like to thank all the master students I have had the joy of working with and in particular Afshan Samani, Carlos Aleixandre Tudó and Farinaz Ghasemi. 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 friends, the Chalmers study friends 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, for complementing me in the right areas and sharing my values in the right areas, and for always being there for me. I thank my daughter Elvira for her love, her endless energy and curiosity, and for just being awesome. 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 all discussions and for always feeling welcome in his home. I 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, 2012

# Contents

# List of Figures

# Part I

# INTRODUCTION

# 1 Introduction

## 1.1 Introduction

### 1.1.1 Sensor 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 a 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 their vicinity, and they establish an infrastructure.

There are many application areas for sensor networks. The possibilities span areas as civil security, health care, agriculture, research, environmental, commercial and military applications [1, 2]. There are many parameters 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 are 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 losing coverage of the monitored area or losing 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 according to needs and can change over time [3]. The possibility of rapid deployment can be of high value for many areas like medical, civil security and military. One example is rapid monitoring of 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 communication is usually the most expensive activity of a sensor node. A MICAz node in receive mode uses around 20 mA [5], which would empty 1000 mAh 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 many 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 Requirements

Security is critical for many applications of sensor networks. Some concrete examples of applications obviously needing security include border protection, trespassing and burglar alarm systems, surveillance systems, systems dealing with industrial secrets, and law enforcement and military applications in general. However, just as for other kinds of networks and systems, security is important for a much wider set of applications. There are gains in attacking many different kind of systems for different purposes. An entity that wants to attack the network is called an *adversary*.

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 aim 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 monitor restricted areas might have active adversaries 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.

Sensor networks are deployed in areas that are to be monitored. This usually implies that they are physically available to an adversary. Furthermore, to 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 traditional security algorithms inappropriate for use in sensor networks [7]. 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 a base station, or between nodes close to each other. Another possibility is that someone with a smart device can query the network dynamically. Thus, it temporarily takes the base station role at some place in the network topology to collect data after which it leaves the network. In any case the traffic does not flow 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 adversary 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 addressed. It also means that security protocols that are used in other networks, e.g., the Internet, are often unsuitable 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 adversary can use compromised nodes to attack the network [8]. The adversary could place her own sensor nodes in the area to disturb or infiltrate the network. The adversary 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 replace a captured node after extracting secret information, like cryptographic keys. Malicious nodes like this inside the network, *compromised nodes*, are a challenge to deal with and this is an important area for research. Compromised nodes can do a lot of damage to the network. They can use and share 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 rarely something that can be added on top of insecure systems to be able to withstand attacks. Security needs to be part of most protocols and algorithms in the system. Otherwise the adversary can chose to direct the attention to the unsecured parts. Therefore, it is important to have secure algorithms for all the basic services that are needed in sensor networks.

This is just a short introduction. In the following sections of the chapter we are going to look at attacks towards sensor networks in general and look at cryptography, key management, authentication, localization, clock synchronization, clustering, routing, aggregation and self-stabilization in more detail. More information on other security challenges can be found in [9], [10], [11] and [12].

## 1.2 Attacks in general

### 1.2.1 The Adversary

An adversary is an entity that attempts to break the security of a system. The purpose may be to extract secret information, to gain unauthorized access to the network or to cause harm to the network. Here, we give a brief overview of the adversary and different general attacks against sensor networks. Additional details can be found in [13].

We can distinguish between a *passive* and an *active* adversary:

- A passive adversary only monitors the communication link and listens to every piece of information that passes through. The adversary uses this information offline to try to break confidentiality to gain unauthorized information.

- An active adversary can use all the techniques available to a passive adversary. She can also interfere with the operations of the network by tampering with nodes, sending messages, causing collisions, jamming communications and performing other active attacks. This has the potential to cause much greater harm to the network as it may in turn cause other changes to the network. Here integrity and availability can be attacked in addition to confidentiality.

We can also distinguish between a mote-class adversary and a laptop-class adversary:

- A mote-class adversary has access to one or a few nodes with capabilities similar to the nodes that are deployed in the network.

- A laptop-class adversary has access to a much more powerful device than the sensor nodes, e.g., a laptop. This allows for a larger set of attack techniques.

Finally, we can distinguish between an insider and an outsider adversary:

- An insider adversary is able to compromise or capture nodes of the network or insert new nodes of her own into the network. Once this is done she can attack the network using these nodes.
- An outsider adversary has no such access to nodes inside the network.

### 1.2.2 Physical Layer Attacks

#### Jamming

Jamming is a physical layer attack in which the adversary transmits signals over the wireless medium to prevent other nodes from communicating because of the signal to noise ratio being too low [14].

#### Tampering

The adversary gains physical access to the nodes where they are deployed. This allows for extracting information, e.g., cryptographic keys, or even reprogramming them and redeploying them. Such reprogrammed *compromised* nodes can be used in insider attacks [15].

#### Sensor Manipulation

The sensing hardware itself might also be spoofed or attacked. Possibilities range from distant manipulations, e.g., by laser pointers, to local manipulations, e.g., chemical sprays.

### 1.2.3 Data-link Layer Attacks

#### Collisions

In collision attacks the adversary sends messages that collides with specific messages, instead of constantly jamming the medium. The adversary figures out when a message is being sent, either from knowing details about the protocols the sensor nodes are running or simply by listening to the communication medium to hear transmissions that are being started. Then, at the same time as this message is being sent, she sends a message of her own, causing a collision preventing other nodes from receiving the message.

#### Exhaustion Attacks

The batteries of sensor nodes can be exhausted if the network faces continuous collisions and back-off in MAC protocols, potentially resulting in degradation of availability.

### 1.2.4 Network Layer Attacks

#### Selective Forwarding

Malicious nodes can refuse to forward some or all messages that are supposed to be forwarded by it to other nodes. This can break many protocols or result in delays and bandwidth degradation in the network.

#### Sinkholes

In a sinkhole attack a compromised node sends out incorrect routing information to erroneously convince other nodes that it is a good node to route through to, e.g., towards a base station [16]. This allows for larger impact for selective forwarding attacks or to tamper with forwarded messages.

#### Sybil Attacks

A Sybil attack is when a malicious node creates its own multiple identities and presents them to other nodes in the network [17–19]. This can give the malicious node a larger influence in many different protocols, e.g., with voting or redundancy, than it would have just using its own identity.

#### Hello Floods

A laptop-class adversary broadcasts messages with powerful signals reaching a large portion of the network. Being regarded as a neighbor of many nodes it can gain undue influence, especially in routing protocols [16].

#### Wormhole Attacks

Two nodes in different regions of the sensor network can launch a wormhole attack if they share have a low latency link, separate from the normal communication channels. In this attack one of them relays messages from its neighborhood to the other one that in turn replays these messages in its neighborhood. This can lead nodes to get an incorrect view of the network topology and fool services that relies on topology knowledge.

## 1.3 Cryptography, Key Management and Authentication

A set of different attempts to implement secure communication specifically for wireless sensor networks appears in the literature. Solutions such as TinySec [20], SenSec [21], MiniSec [22], and TinyECC [23] are all designed to run under TinyOS [24], a widely used operating system for sensor nodes. ContikiSec [25] presents a system designed for the Contiki operating system [26].

### 1.3.1 Security Properties

Security properties that should be provided by a secure network layer for wireless sensor networks are briefly described below. After that, individual paper contributions are discussed.

#### Confidentiality

Confidentiality is a basic property of any secure communication system. Confidentiality guarantees that information is kept secret from unauthorized parties. The typical way to achieve confidentiality is by using symmetric key cryptography for encrypting the information with a shared secret key. Symmetric key algorithms are often divided into stream ciphers and block ciphers. In the case of block ciphers, a mode of operation is needed to achieve semantic security (see below).

#### Semantic Security

Semantic security guarantees that a passive adversary cannot extract partial information about the plaintext by observing the ciphertext [22]. Block ciphers do not hide data patterns since identical plaintext blocks are encrypted into identical ciphertext blocks. Thus, a special mode of operation and an initialization vector (IV) are often used and are needed to provide some randomization. Initialization vectors have the same length as the block and are typically added in clear to the ciphertext.

#### Integrity

Integrity guarantees that the packet has not been modified during the transmission. It is typically achieved by including a message integrity code (MIC) or a checksum in each packet. The MIC is computed by calling a cryptographic hash function. By comparing the current MIC with the one stated in the packet, malicious altering or accidental transmission errors can be detected. Checksums are designed to detect only accidental transmission errors.

#### Authenticity

Data authenticity guarantees that legitimate parties should be able to detect when a message is sent by unauthorized parties and reject it. One common way to achieve authenticity is by including a message authentication code (MAC) in each packet. The MAC of a packet is computed using a shared secret key, which could be the same key used to encrypt the plaintext. In such a scheme, anyone that knows this shared secret key can issue a MAC for a message. In contrast, public key authentication algorithms can provide authentication for which anyone that knows the public key can authenticate that a message is from the one entity holding the corresponding private key.

### 1.3.2 Symmetric Key Cryptography

In recent years, the increased need of security in wireless sensor networks has prompted research efforts to develop and provide security modules for these platforms. These efforts go from simple stream ciphers to public key cryptography architectures.

SPINS [27], presented in 2002, is the first security architecture designed for wireless sensor networks. It is optimized for resource-constrained environments and it is composed of two secure building blocks: SNEP and μTesla. SPINS offers data confidentiality, two-party data authentication, and data freshness. However, SNEP was unfortunately neither fully specified nor fully implemented [20].

In 2004, TinySec [20] was presented as the first fully implemented link layer security suite for wireless sensor networks. It is written in the nesC language and is incorporated in the official TinyOS release. TinySec provides confidentiality, message authentication, integrity, and semantic security. The default block cipher in TinySec is Skipjack, and the selected mode of operation is CBC–CS. Skipjack has an 80-bit key length, which is expected to make the cipher insecure in the near future [28]. In order to generate a MAC, it uses Cipher Block Chaining Message Authentication Code (CBC–MAC), which has security deficiencies [29]. It provides semantic security with an 8-byte initialization vector, but adds only a 2-byte counter overhead per packet. TinySec adds less than 10% energy, latency, and bandwidth overhead.

SenSec [21] is another cryptographic layer, presented in 2005. It is inspired by TinySec, and also provides confidentiality, access control, integrity, and semantic security. It uses a variant of Skipjack as the block cipher, called Skipjack-X. In addition, SenSec provides a resilient keying mechanism.

MiniSec [22] is a secure sensor network communication architecture designed to run under TinyOS. It offers confidentiality, authentication, and replay protection. MiniSec has two operating modes, one tailored for single-source communications, and the other tailored for multi-source broadcast communication. The authors of MiniSec chose Skipjack as the block cipher, but they do not evaluate other block ciphers as part of their design. The mode of operation selected in MiniSec is the OCB shared key encryption mechanism, which simultaneously provides authenticity and confidentiality.

TinyECC [23] is a configurable library for elliptic curve cryptography operations for sensor nodes. It was released in 2008 and targets TinyOS. Compared with the other attempts to implement public key cryptography in wireless sensor networks, TinyECC provides a set of optimization switches that allow it to be configured with different resource consumption levels. In TinyECC, the energy consumption of the cryptographic operations is on the order of millijoules, whereas using symmetric key cryptography is on the order of microjoules [30].

### 1.3.3 Key Management

No cryptographic algorithms can of course be used without having the nodes share keys in some way, regardless if it is secret keys for symmetric cryptography or public keys for public key cryptography. There are many different approaches to share keys in a secure manner.

#### Key Predistribution

In key predistribution solutions the nodes are being loaded with keys before deployment and with these keys the nodes can setup communications and possibly generate new keys. In regards to the risk of having nodes compromised, more sophisticated solutions are needed than merely having one master key shared by all nodes. However, considering the other end of the spectrum, it is not generally feasible for all pair of nodes to share a unique key. That takes up far too much storage space.

In [31], Eschenauer and Gligor present a random predistribution scheme that starts out by drawing a number of keys randomly for each node before deployment from a pool of keys. After deployment nodes discover what keys they share with neighboring nodes. They can then set up secure communications using those shared keys. With properly set parameters the chance of a node sharing at least one key with a certain neighbor is high.

To increase the resiliency against compromised nodes in the network Chan et al. propose in [32] a method in which it is not enough to just share one pregenerated key but a certain number of pregenerated keys. Other methods set a threshold on the number of compromised nodes that can be tolerated. These include [33], [34] and [35].

Various methods, e.g., [36], [37], [38], [39] and [40], aim to reduce the overhead of key predistribution by taking into account roughly which areas different nodes will be deployed in and predistribute keys accordingly to reduce the number of needed keys for nodes to keep track of.

The SecLEACH protocol in [41] adapts the idea in [31] to set up secure communications for the changing clusters generated by the cluster algorithm in [42].

The previous methods are all probabilistic in the sense that there were no guarantees that a certain pair of nodes would share keys with each other. Chan and Perrig presents a method in [43] in which nodes deterministically set up $\sqrt{n}$ different keys per node using other nodes as trusted intermediaries. Here $n$ is the size of the network and each key is a pairwise key shared by only two nodes.

### Other Mechanisms

Zhu et al. reason in [44] that in many systems it takes a longer time for an adversary to compromise nodes than for nodes to set up keys between themselves. They use a global predistributed key together with unique node identifiers to set up pairwise keys with direct neighbors and to set up a cluster key for a cluster of nodes. This predistributed key is erased to limit the effect of compromised nodes. They also present an efficient way for the base station to share pairwise keys with each node in the network and discuss how to update a global network key in case of node compromise.

Anderson et al. present a technique in [45] in which keys are generated and transmitted in clear text. Assuming that an eavesdropping adversary cannot eavesdrop everywhere at once, not all keys will be known to an adversary. Nodes then take help from other nodes to reinforce the security of keys so that a key that might be known by the adversary gets updated to a key that is not known even if the adversary listens in on the update messages. In [46], Cvrcek and Svenda verify results from [45] and introduces a variant of the key reinforcement scheme. Miller and Vaidya also exchange keys in the clear in [47], but use multiple channels to make it hard for an eavesdropping adversary to get hold of more than a few of the keys that are being broadcast in its vicinity.

In [48], Oliveira et al. set up keys in clustered heterogeneous networks between nodes and their cluster heads. They use a hybrid approach by partly using predistributed keys and partly setting up new keys between nodes.

More details on key management in wireless sensor networks can be found in the survey by Camtepe and Yener in [49] and the review by Zhang and Varadharajan [50].

### 1.3.4 Authentication

Authentication is a keystone for secure protocols. Public key based authentication schemes are very powerful, but may be too expensive for sensor networks.

The SNEP protocol in [27], the LEAP protocol in [44], the TinySec protocol in [20] and an AES-based protocol in [51] provide node to node authentication without resorting to public key cryptography. In [52], an algorithm is presented that is specifically aimed for ZigBee networks that has already been organized into clusters.

#### Broadcast Authentication

Broadcasting is important for many sensor network services. Thus there is a need for authenticating broadcasts in an efficient manner.

In [27], Perrig et al. also introduce the µTESLA algorithm for authenticating broadcasts. The basic idea is as follows. A chain of keys is created in reverse. A key in the chain is generated by using a one-way hash function on the next key of the chain. Time is divided into timeslots and one key is assigned for each timeslot. The creator of the keys can in one timeslot send a message with a MAC calculated by a key in the chain. In a later timeslot it can reveal the later key in the chain, which only the creator of the key chain can do. In that way it authenticates that it sent the message. The last key of the chain, that is not the hash of some other key, needs to be distributed and authenticated separately, which requires predistribution. In [53], Liu and Ning reduce the setup requirements and increase the robustness of µTESLA. In [54], Liu and Ning introduce multi-level key chains to allow for better scaling. Liu et al. add revocation possibilities to µTESLA in [55] and using basic µTESLA as a building block allows for better scaling with reduced storage needs and better resiliency against denial of service attacks. Luk et al. present in [56] the RPT protocol, based on µTESLA, that is specially suited for authenticating broadcasts that are sent at regular times. They also present the LEA protocol that is aimed for broadcasts with low entropy. They discuss different properties of broadcast authentication and what protocols to use depending on the underlying needs of a system.

### User Authentication

Separate from the authentication problem, where nodes authenticate themselves to each other, is the user authentication problem, where a user of the network is being authenticated by the nodes in the network. Just as for node to node authentication, different methods are based on tools like public key cryptography, symmetric key cryptography and one way hash functions. The user that is being authenticated can often be assumed to be much more powerful in terms of processing power, memory, storage, etc.

For node to node authentication in a static network there might be no need for any node to be able to authenticate any other node. In contrast, for user authentication, it might be required for any node in the network to be able to authenticate any user. Additional challenges arise when there is a need for privacy for the users. For details on this topic, we refer the reader to papers such as [57], [58], [59], [60], [61] and [62].

## 1.4 Localization

### 1.4.1 The Importance of Localization

Localization is the service providing information about where sensor nodes are located. This is needed to identify where different events happened, both by knowing the location of the nodes sensing the event and, using multiple cooperating sensors, where the event itself took place. Geographic location information is also needed for other services like geographic routing, geographic information querying, geographic key distribution, location-based authentication and checking geographic network coverage. It is also useful if the nodes themselves need to be found, e.g., for repairs or battery changes, or to find resources tagged by sensor nodes.

### 1.4.2 Localization Techniques

The easiest method to localize sensor nodes is to use GPS. However, this can be unfeasible due to several reasons: (1) it makes the nodes more expensive, (2) it drains batteries much quicker, and (3) it makes the nodes larger. Also, GPS does not work properly in all environments such as indoors, between tall buildings, etc.

There are two basic categories of localization algorithms. The first one is based on so called infrastructure-based techniques in which there are some entities called beacons, possibly a subset of the sensor nodes themselves, that are equipped with GPS or know their location by some other means. With the help of these beacons the location of the regular nodes in the network can be calculated. The second category include autonomous techniques in which no such special hardware or infrastructures are available. Another characteristic is if a protocol is range-dependent or range-independent, i.e., whether there is a need to calculate distances between nodes.

The usual way to measure the location of a node is to collect data from nodes in the neighborhood and use this information to calculate the node's location. The information needed include distances and/or angles to other nodes together with their respective locations. Distances can be calculated using signal strength or receive time of signals. Finally, the location can be calculated using techniques like triangulation, trilateration or multilateration.

### 1.4.3 Attacks Against Localization

Attacks include beacon nodes reporting false locations in beacon messages, ordinary nodes reporting false locations for location verification techniques, misrepresenting distances, e.g., by sending with a different transmission power in signal strength base techniques or using delay attacks (see Section 1.5.3) to misrepresent signal propagation times. Impersonation, wormhole attacks and Sybil attacks can also be used to fool nodes to calculate incorrect locations [63].

### 1.4.4 Secure Localization

The SeRLoc protocol in [64] is a range-independent protocol in which the nodes of the network are divided into two sets. One set of nodes have omnidirectional antennas and the other set, the locators, are equipped with directional antennas. The locators send out different beacons in different directions that contain the position of the locator and the broadcasting angle of the antenna. The normal nodes use these beacons to calculate their position. As the non-locator nodes do not participate actively in the protocol, the locators or their messages would be the points that adversaries are most interested in manipulating to attack the protocol. The nodes and locators share a symmetric key that is used to encrypt the location information. The beacons are authenticated by a one-way hash chain. The protocol defends against a set of compromised nodes and wormhole attacks.

In [65], Zeng et al. improve the Monte Carlo based localization technique for mobile sensor networks described in [66]. They add authentication, filter out inconsistent values and add a new sampling method to be used in case of detected attacks.

Chen et al. present three localization techniques in [67], that use detection mechanisms to detect and disregard nodes with malicious behavior. The detection mechanisms look for nodes that send multiple messages when they should only send one, pair of nodes that claim to be further away from each other than possible given that both were heard by the same node, and nodes that do not act consistently with other nodes. Furthermore, nodes that act consistently with already detected misbehaving nodes are also deemed misbehaving. In [68], Chen et al. present a wormhole localization algorithm based on distance inconsistencies and inconsistencies where nodes receive their own messages or the same message multiple times. The algorithm can not deal with packet loss though, but is further refined in [69] where packet loss is taken care of.

Iqbal and Murshed use trilateration in [70] on all possible subsets of size three of the neighboring beacon nodes to find out the area that data from most triplets produce. Thus, many malicious beacons need to collude to sway the result of a node as long as fair number of honest beacons are in range of that node. Simulations compare the algorithm favorably with the EARMMSE algorithm in [71].

Algorithms that use received signal strength to calculate distances for use in localization calculations are vulnerable to attacks that tamper with received signal strength, e.g., by placing absorbing or reflecting materials in the area. In [72], Li suggests that algorithms should instead be implemented using signal strength differences to be resilient against such attacks.

In [73], Jadliwala et al. investigate under which conditions location errors can be bounded in a setting with captured beacon nodes. They show a lower bound on the number of captured nodes and describe a class of algorithms that can bound the location error. They also present and evaluate three algorithms that are in this class.

In [74], Mi et al. present a technique for secure localization (together with location-based key distribution) in networks that are manually deployed with a GPS equipped master node. They defend against wormhole attacks, restrict impact of insider nodes and propose using motion sensors as a backup if the GPS module becomes unusable, possibly due to an attack.

In [75], Wozniak et al. investigate the robustness using least median squares in a multi-hop distance vector technique and present modifications that need to be made in order to withstand attacks.

Above we have described major recent results, but more details can be found in the surveys [63], [76], [77] and [78] that exclusively look at the topic of secure localization.

## 1.5 Clock Synchronization

### 1.5.1 The Importance of Clock Synchronization

Many wireless sensor network applications and protocols need a shared view of time. Examples include localization schemes, pinpointing and tracking events, scheduling of a shared radio medium, e.g., using Time Division Multiple Access (TDMA), detecting duplicate events. For some applications the precision needs to be very high. Therefore, clock synchronization protocols are crucial for wireless sensor networks. Broadly speaking, existing clock synchronization protocols for more general networks are too expensive for sensor networks because of the nature of the hardware and the limited resources that sensor nodes have.

### 1.5.2 Clock Synchronization Techniques

Elson et al. present the reference broadcast synchronization technique in [79], in which beacon nodes are broadcast wirelessly. Due to the wireless medium different recipients will receive the beacon at more or less the same time, thus having a common event to relate to. All recipients of the beacon sample the clock when they receive it, and by comparing their clock samples they can approximate offsets between their respective clocks.

Another technique for approximate clock offsets is the round-trip synchronization technique used by Ganeriwal et al. in the TPSN protocol described in [80]. A message is sent from node $A$ to node $B$ and another message back from $B$ to $A$. By sampling the clocks at send and receive of the two messages, the clock offset can be approximated, given that the delays for the two messages are close to equal. The delay can also be approximated from this information, given that the clock rates are approximately equal. This can be useful, especially when a long delay can be a sign of an attack.

A third technique that Maroti et al. use in the FTSP protocol in [81] is to have a clock source and then using a hierarchy to flood the time from the source outwards, with nodes synchronizing their time to the closest node higher up in the hierarchy that they received the time from.

The clocks of the nodes can be synchronized using the approximations of clock offsets gained by the above techniques. Elson et al. [79] use linear regression to deal with differences in clock rates. Their basic algorithm synchronizes a cluster. Overlapping clusters with shared gateway nodes can be used to convert timestamps among clusters. Karp et al. [82] input clock samples for beacon receive times into an iterative algorithm, based on resistance networks, to converge to an estimated global time. Römer et al. [83] 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.

### 1.5.3 Attacks Against Clock Synchronization

One threat from insider nodes is that they can send out incorrect timestamps used at various points in many of the common clock synchronization techniques. Another threat is that the malicious nodes in some cases can be placed, possibly due to deliberate manipulation of protocol, in important positions in hierarchies used in global synchronization techniques.

A different threat is the so called delay attack (also known as the pulse-delay attack) described in [84]. An adversary can receive (at least part of) a message, jam the medium for a set of nodes before they receive the entire message, and then replay the message slightly later. This requires no inside nodes in the network or any cryptographic keys, but the jamming must happen at a precise moment in time. This attack can also be performed, without the time requirement for the jamming, by two collaborating insider nodes. The first jams the network in a small area and the second, outside this area, receives the message normally. Then the second node sends out the jammed beacon at a later time or forwards it to the first node to send out at a later time.

Additional details on attacks against clock synchronization in wireless sensor networks can be found in [85].

### 1.5.4 Secure Clock Synchronization Techniques

Song et al. present in [86] ways to detect bad timestamp values from insider nodes using a roundtrip synchronization approach and two methods to filter out such outlier values. The first uses the generalized extreme student deviate algorithm and the other uses a time transformation technique to filter out timestamps that have too large offset values.

Sun et al. present in [87] two related schemes to withstand attacks from insider nodes. One divides nodes into levels depending on their distance to a clock source node by comparing pairwise clock differences in a chain between the nodes and the source. The other uses a diffusion scheme that allows for any pair of nodes to compare clock differences with each other. The authors also show how to use several source nodes for this second scheme. The first scheme is more efficient and provides better precision, whereas the second provides better coverage. The algorithms are vulnerable to delay attacks though.

Sun et al. present in [88] a two-phase algorithm where one phase uses a roundtrip synchronization technique to give a basic pairwise synchronization between nodes. They present a way to both timestamp and add authentication to messages on the fly while transmitting to be able to timestamp as close as possible to the actual transmission. Phase two adapts the µTesla solution from [27] to get local broadcast authentication (which needs the loose synchronization from phase one) and achieves global synchronization. Key chains of rapidly expiring keys defend against delay attacks.

Sanchez synchronizes nodes both pairwise and, in a clustered network, clusterwise in [89], using the round-trip synchronization technique. They take duty cycling into account so that nodes can be sleeping between synchronization rounds and their technique defends against some nodes in the network being compromised.

Ganeriwal et al. present a family of clock synchronization algorithms in [84] and [90]. They are based on the roundtrip synchronization technique in [80] and filter out over-delayed message exchanges to fend against delay attacks and compromised nodes. They present both single and multi-hop pairwise synchronization techniques as well as group synchronization techniques, where some can deal with insider attacks from compromised nodes and some can not. Byzantine agreement is used to get a group synchronization algorithm that withstands insider attacks in the group synchronization.

Hoepman et al. present in [91] a secure clock synchronization algorithm with a randomized clock sampling algorithm at the core. The algorithm is resilient against both delay attacks and attacks from insider nodes. Moreover, the algorithm is self-stabilizing. The clock sampling allows a combination of getting the precision of the reference broadcast technique were many nodes have common points with the ability of the roundtrip synchronization technique to detect spurious delays.

Hu et al. [92] consider under-water sensor networks where nodes communicate using acoustic means and may be following streaming water. Nodes deployed at different depths move at different speeds. In this setting the propagation delay is variable and far from negligible and must be taken into account. They propose a method that synchronizes clocks vertically, between nodes at different depths. They consider insider attacks from compromised nodes and use various statistical methods to detect and defend against such attacks.

Li et al. build up a hierarchy under a base station in [93] and use overhearing to get verification that nodes do not send out incorrect data. Hu et al. use an FTSP style flooding protocol in [94] and use a system of predicting future clock values to detect attacks from insider nodes. Roosta et al. propose in [95] a set of attack countermeasures for the FTSP protocol and present results from their testbed implementation. Chen and Leneutre propose a method using one-way hash chains in [96] to ensure authenticity and integrity of synchronization beacons. Rasmussen et al. show in [97] methods to protect against attacks towards localization and clock synchronization protocols with the help of external navigation stations. Farrugia and Simon use a cross-network spanning tree in which the clock values propagate for global clock synchronization in [98]. They use passive overhearing to let some nodes synchronize without the need of active participation. They defend against replay and worm-hole attacks. Du et al. discuss in [99] how to take advantage of high-end nodes with GPS to improve efficiency for secure clock synchronization. Secure clock synchronization in wireless sensor networks is also discussed in [100].

## 1.6 Clustering

### 1.6.1 The Importance of Clustering

Clustering nodes together into groups is an important low level service for wireless sensor networks. Sensor networks, like other ad-hoc networks, need to organize themselves after deployment. Clustering sets up a structure, e.g., 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.

### 1.6.2 Clustering Techniques

One way of clustering nodes in a network is to have nodes associating 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 $r$ hops is smaller than $k$. In such cases a best effort approach can be taken for getting as close to $k$ cluster heads as possible. Assuming that the network allows $k$ cluster heads for each node, the set of cluster heads forms a (k,r)-dominating set in the network. If the cluster heads need to have $k$ cluster heads as well, it forms a *total* (k,r)-dominating set, in contrast to an ordinary (k,r)-dominating set in which this is only required for nodes not in the set. The clustering should be achieved with as few cluster heads as possible. Finding the global minimum number of cluster heads is in general NP complete, so algorithms usually provide an approximation instead. Many algorithms are limited to providing (1,1)-clustering and some provide (1,r)-clustering, (k,1)-clustering or other subsets of (k,r)-clustering.

Some clustering algorithms provide a number of cluster heads but do not make sure that a certain node has a number of cluster heads within some certain radius, but instead use random approaches to get a good statistical coverage.

Another way of providing clusters is for nodes to assign themselves to different clusters without any nodes being assigned as cluster heads. Often these clusters are based on cliques, sets of nodes that forms a complete graph.

A general overview of clustering in wireless sensor networks can be found in [101] by Abbasi and Younis. A survey on clustering wireless ad-hoc networks in general can be found in [102].

### 1.6.3 Attacks against clustering algorithms

As for other services, an adversary can disturb protocols from the outside, e.g., by jamming the network, causing collisions, inserting false messages and replaying possibly altered messages. Apart from defending against such outside attacks, it is important to take attacks by malicious insider nodes into account.

By not following protocol, malicious nodes can make sure to be cluster heads whenever they want in protocols where nodes declare that they are cluster heads with a certain probability. Thus they can gain an undue influence in the network and from there have a better platform to launch attacks against other protocols that is running on top of the clustering service. Instead of assigning cluster heads, other algorithms form clusters of nodes by agreeing upon group membership. For such algorithms, a malicious node can send conflicting information to other nodes so that they cannot agree on which nodes are part of which groups. For multi-hop clustering a malicious node can forward false information on which nodes are cluster heads and which are not.

### 1.6.4 Secure Clustering Algorithms

In [103], Sun et al. present a secure clustering algorithm that divides the network into disjoint cliques, sets of nodes that all can communicate directly with each other and where each node belongs to exactly one clique (possibly by itself). No cluster heads are assigned. The algorithm takes compromised nodes into account. The use of signed messages allows for nodes to be able to prove misbehavior of malicious nodes to be able to remove them from consideration.

The SLEACH algorithm that Wang et al. present in [104] is based on the LEACH algorithm in [42]. Time is divided into rounds and in each round nodes become cluster heads with a certain probability. To make sure that no node can become cluster head too often, or for outsider nodes to be able to join the protocol, extensive key exchanges are done with a base station.

Banerjee et al. present in [105] the GS-LEACH protocol. It is another secured version of the LEACH protocol. It is based on key distribution that is done in grids with nodes within the grids taking turns being cluster heads.

Wang and Cho in [106] look at secure clustering from a secure election point of view and present a scheme based on signal strength to defend against attacks that try to split an agreement of election results.

### 1.6.5 Self-stabilizing Clustering Algorithms

There is a multitude of existing clustering algorithms for ad-hoc networks of which a number are self-stabilizing. Johnen and Nguyen present a self-stabilizing (1,1)-clustering algorithm that converges fast in [107]. Dolev and Tzachar tackle a lot of organizational problems in a self-stabilizing manner in [108]. 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 [109] that takes weighted graphs into account. Larsson and Tsigas present a self-stabilizing (k,r)-clustering algorithm in [110] and [111].

## 1.7 Routing

### 1.7.1 The Importance of Routing

Unless the user of the network moves around in the area the network is deployed in and collects data directly from the nodes, information needs to be sent through the network. Therefore the nodes need to solve the routing problem, i.e., how to forward messages through the network when a message needs to travel from some node to another. At times there is only a need for information to flow between each sensor node and the base station. Therefore some algorithms only take care of routing to and from a base station.

### 1.7.2 Attacks Against Routing Protocols

We present an overview of different attacks that can be used to interfere with routing protocols below. Many of the attack techniques are being used against many other types of protocols, but some, like sinkhole attacks, are specifically aimed against routing protocols. For further details we refer the reader to [16].

#### Wormhole Attacks

The idea of the wormhole attack is to tunnel messages via a low latency link between two compromised nodes and replay them in different parts of the network. This can disrupt routing protocols as other nodes will get an incorrect view of the network topology. If one of the compromised nodes is close to the base station, the other compromised node can launch a sinkhole attack (see description below).

#### Sybil Attacks

By presenting multiple identities to the other nodes of the network a node can increase its chances of being included in many communication paths in the network. Other nodes will not realize that these identities in fact belongs to one physical node.

### Clone Attacks

This attack is a relative of the Sybil attack where a node acts using multiple existing identities. Keys or other credentials from different captured nodes are being used by different compromised nodes in many different places in the network to maximize the possible damage. By being located in different regions no legitimate nodes can directly hear different traffic sources using the same credentials. Therefore, by having many compromised nodes presenting themselves as many legitimate nodes each, they can gain a large influence in the network.

### Selective Forwarding

A simple form of the selective forwarding attack is for a compromised node to act like a “black hole” by refusing to forward any messages. However, in many protocols this results in the node being regarded as dead and thereafter being excluded from consideration. A more effective attack can be to forward certain messages and drop others to disturb the routing protocol itself or another protocol running on top of the routing protocol.

### Hello Flood Attacks

Many protocols, including routing protocols, exchange some form of so called hello messages, where they present themselves to their neighbors. A laptop-class-adversary, generating a much more powerful signals than the normal nodes, can convince many nodes that the laptop is their neighbor and use this fact to get into a position were many nodes include the laptop in their routes.

### Sinkhole Attacks

Sinkhole attacks are performed by a compromised node by making itself an attractive choice for routing. The goal of this attack is to direct a lot of traffic to a particular area of the network. This position can be used to launch other attacks such as selective forwarding attacks.

### Routing Loop Attacks

The idea behind the routing loop attacks is to create loops in how messages are being routed. The result is that a message are being constantly forwarded around in this loop, draining batteries of nodes involved in the loop and preventing the message from reaching its destination.

### Using False Information

A compromised node can send out false information about its battery levels, its distance to a base station or its location or other metrics that are used to decide how to route. This can make it seem more attractive from other nodes' point of view than it really is, resulting in that the compromised node becomes part of many routing paths after which it can launch selective forwarding or other attacks.

### Base Station Impersonation

In routing algorithms where the goal is to forward messages towards the base station, a simple attack against an unsecured routing protocol can be claiming to be a base station. In protocols that have many possible base stations it might also be possible for a node to insert itself into the lists of available base station without impersonating any existing base stations.

## 1.7.3 Secure Routing Algorithms

Lee and Choi present the SeRINS algorithm in [112] that uses multiple paths to be resilient against attacks by compromised nodes. The algorithm defends against both selective forwarding attacks and injection of false routing data.

The SHEER algorithm by Ibriq and Mahgoub is presented in [113]. It sets up a hierarchy and uses probabilistic transmissions with the aim to preserve energy. It adapts to changes of battery in the network. It does not cope with malicious insider nodes.

Yin and Madria present their SecRout, also known as ESecRout, in [114] which is extended in [115] with more experiments and analysis. It is an algorithm for routing query results from nodes towards the sink. They aim to stop message tampering and selective forwarding attacks by using blacklisting.

Du et al. present the TTSR routing algorithm in [116] that, in a heterogeneous setting, takes advantage of high performance nodes scattered throughout the network together with more limited nodes. It defends against spoofed routing information and selective forwarding, sinkhole, wormhole and hello flood attacks.

The SeRWA algorithm in [117] uses wormhole detection to find routes in the presence of wormhole attacks. It is based on overhearing together with authentication of messages to detect when a node that is supposed to forward a message drops it or tampers with it. Such detected malicious nodes can be excluded and routed around.

In [118], Deng et al. present the hierarchical multiple path routing algorithm INSENS. Here the nodes send their neighbor information to the base station, that in turn chooses the multiple paths for routing. Kumar and Jena use the same basic mechanism for their SCMRP algorithm in [119], but they build up a clustered hierarchy to be more energy efficient. The base station is responsible for the cluster formation process.

For more details on secure hierarchical routing, we point the reader to the survey in [120].

### Geographic Protocols

These protocols assume that the nodes know their locations and use the geographical location knowledge to decide what routes messages should be forwarded along.

In [121], Du et al. present the SCR algorithm, together with a key management scheme. The geographic coordinate system is divided into a grid, or cells. They choose redundant paths for sending a message and forward messages by choosing cells rather than individual nodes. They defend against attacks such as sinkhole, Sybil, wormhole, selective forwarding, hello flood and clone attacks.

Wood et al. present a family of secure routing protocols in [122] with varying levels of security and varying amounts of state that needs to be stored and kept up to date. The weakest provides probabilistic defenses but does not need to keep any state. And stronger ones provides more security guarantees but requires to keep more state information.

The ATSR geographic routing algorithm is presented in [123] and uses a distributed trust model to defend against attacks. It detects and excludes nodes that do not forward messages correctly or that do not execute the trust protocol correctly. It also takes remaining battery levels into account when making routing decisions to prolong the network lifetime.

## 1.8 Aggregation

### 1.8.1 The Importance of Aggregation

Often information from the sensor nodes in the network is gathered at a base station (or by some other entity querying the network). The battery constraints of many wireless sensor networks make it very important to limit communications. Instead of having every sensor reading being sent from every node all the way to the base station data aggregation can be used to produce reports from data gathered by many nodes.

### 1.8.2 Aggregation Techniques

There are several different aggregation techniques. One family of methods forms a tree rooted in the base station and has parents aggregate data from themselves and their children. Another family has cluster heads appointed by running a clustering algorithm (see Chapter 1.6) and has these cluster heads take the role as special aggregator nodes. Aggregation schemes can also be classified as single aggregator or multiple aggregator schemes. In the former, aggregation happens once for each piece of data and the report is transferred to the base station. In the latter, aggregation happens multiple times on the way.

More details on general aggregation in wireless sensor networks can be found in [124].

### 1.8.3 Secure Aggregation Algorithms

Hu and Evans introduce in [125] an aggregation method that is resilient against malicious outsider nodes in the network and against a single compromised key.

Deng et al. present in [126] methods both for nodes to authenticate themselves towards an aggregator, and for an aggregator to authenticate itself toward nodes it aggregates data for.

There are various methods, [127], [128], [129], [130] and [131], with the common denominator that an aggregator needs some form of certificate from the node it aggregates for.

Data injection attacks are done by compromised insider nodes that inject false data to skew the aggregated value [132, 133]. Algorithms for which the largest possible influence done by data injection attacks is proportional to the number of compromised nodes are said to achieve *optimal security*. The algorithms in [133], [134] and [135] all achieve optimal security. However the amount of communication required for a single node might be $O(\log n)$ and they require two round-trip communication rounds between the base station and the nodes of the network. Miyaji and Omote present in [136] an algorithm that achieves optimal security with an $O(1)$ communication load per node and only one round-trip communication round by assuming a weaker adversary model in which the adversary cannot compromise keys of both a node and its parent node.

#### Aggregating Encrypted Data

Some aggregation algorithms use homomorphic encryption techniques. Such techniques aggregate encrypted data without the need of decryption. In this way data from one node can be kept secret from other nodes, but still be aggregated. Let $D$ be a decrypting function and $E$ the corresponding encrypting function. The cryptographic algorithm is additively homomorphic if $D(E(a) + E(b)) =$

$a + b$, for any $a$ and $b$. In the same way it is multiplicatively homomorphic if $D(E(a) \cdot E(b)) = a \cdot b$.

Castelluccia et al. present in [137] how to use an additive homomorphic encryption scheme to let nodes keep their data private while still being able to efficiently calculate functions over the data from different nodes. They support calculating sums, mean variances and standard deviations. Parent nodes in a tree can aggregate encrypted data from their children without any decryption. Moreover, the method defends against outside tampering of any data with an authentication scheme. However, there is no prevention to avoid bad values from a node inside the network.

In [138], Huang et al. present a single aggregator scheme for keeping sensor data private. It provides an encryption method that lets an aggregator evaluate if two of its children provide the same data without revealing the value itself. In [139], Ozdemir and Xiao present an algorithm that allows for aggregation of data encrypted with different encryption keys in different regions. Bahi et al. achieve homomorphic encryption using elliptic cryptography in [140]. Other algorithms involving homomorphic encryption include [141], [142], [143] and [144].

#### Further Reading on Secure Aggregation

More details on secure data aggregation for wireless sensor networks can be found in the surveys [145], [146], [147], and [148].

## 1.9 Self-stabilization

Self-stabilizing algorithms [149–151] 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 memory, 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 [152].

In the sensor network setting assumptions about the system could eventually be violated when an adversary, far more powerful than the limited sensor nodes, starts disturbing the sensor network. An example is a temporary denial of service attack that disturbs communications to a level where assumptions about message throughput are violated. It can be hard to anticipate all possible states the network could end up in after an attack. Large numbers of nodes could get compromised and send incorrect information, nodes could be physically attacked in different ways or the adversary 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 adversary has been chased away or more nodes have been added to the network.

As an example, the secure and self-stabilizing clock synchronization algorithm presented in [153] and [91] assumes that there is an upper bound on the fraction of sent messages from each node that are being lost due to malicious collisions or attacks. The underlying assumption is that an adversary wishes to remain undetected and therefore does not jam or produce collisions for all messages of a node. In a situation where this bound assumption does not hold, e.g., if the adversary attacks more messages than that, the algorithm cannot guarantee to deliver the specified level of service. In this case it cannot guarantee to share a complete set of timestamps between neighboring nodes with high probability within a certain time span. When message delivery assumptions once again hold, e.g., after the adversary is detected and chased off, the algorithm can, due to the self-stabilizing property, quickly recover and deliver the promised level of service.

## 1.10 Our Research 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 take security into account does not take compromised nodes into account.

Our approach is to provide high level networking protocols for sensor networks and/or ad hoc networks that are self-stabilizing and that takes security into account. 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.

We have looked at two fundamental network services, 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 nodes to take turns doing energy intensive tasks and for aggregating data from many nodes to reduce the amount of data that needs to be sent through the network. It can also be used for forming communication backbones, for routing and for building hierarchies that allow for better scaling.

## 1.11 Contributions

### 1.11.1 Paper I: Secure and Self-stabilizing Clock Synchronization in Sensor Networks

As we have seen above, accurate clock synchronization is imperative for many applications in sensor networks. 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 capture nodes and intercepts messages that it later replays – a so called *pulse delay attack*. Our algorithm guarantees automatic recovery after failures from an arbitrary state. 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 $O(n \log n)$).

### 1.11.2 Paper II: A Self-stabilizing (k,r)-clustering Algorithm with Multiple Paths for Wireless Ad-hoc Networks

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.

In the second paper, we present the first distributed 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 algorithm assumes that all communication is reliable.

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.

### 1.11.3 Paper III: Self-stabilizing (k,r)-clustering in Clock Rate-limited Systems

In the third paper we develop the algorithm from the second paper further to make it more general. That algorithm assumes perfect message transfers and lock step synchronization of the nodes. With regards to message loss, in this article we only assume that out of a certain number of messages, at least one is transmitted successfully. Furthermore, we only assume a limit on differences in the clock rate of the local clocks of nodes with no synchronization between the nodes.

In addition to the more general system settings, a veto mechanism against nodes leaving the role as a cluster head is introduced to speed up convergence.

## 1.12 Conclusions & Future Work

In this thesis, we have presented three self-stabilizing high level networking protocols that takes security into account in some way. One secure algorithm for clock synchronization and two for clustering with redundancy that can improve security. Both services are 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.

To allow for secure and self-stabilizing applications to be widely deployed in large scale networks many fundamental network services are needed. More research needs to be done to provide algorithms for these services that are both self-stabilizing and take security into account. Especially important is to defend against attacks from nodes within the network. Another interesting direction is to combine different protocols together into a secure and fault tolerant package for increased efficiency and ease of use.

There is 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 want to investigate ways to choose among possible paths to retain as much redundancy as possible without relying on flooding when building on top of the (k,r)-clustering algorithms.

## 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(L),” `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, 2009.

[5] “MICAz data sheet,” `www.openautomation.net/uploadsproductos/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] E. Shi and A. Perrig, “Designing secure sensor networks,” *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 38 – 43, dec. 2004.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Adrian Perrig, John Stankovic, and David Wagner, “Security in wireless sensor networks,” *Commun. ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[13] Tanya Roosta, Shiuhpyng Shieh, and Shankar Sastry, “Taxonomy of security attacks in sensor networks and countermeasures,” in *The First IEEE International*

*Conference on System Integration and Reliability Improvements*, Hanoi, 2006, pp. 13–15.

[14] Murat Çakiroğlu and Ahmet Turan Özcerit, “Jamming detection mechanisms for wireless sensor networks,” in *Proceedings of the 3rd international conference on Scalable information systems*, ICST, Brussels, Belgium, Belgium, 2008, InfoScale ’08, pp. 4:1–4:8, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[15] Mauro Conti, Roberto Di Pietro, Luigi Vincenzo Mancini, and Alessandro Mei, “Emergent properties: detection of the node-capture attack in mobile wireless sensor networks,” in *Proceedings of the first ACM conference on Wireless network security*, New York, NY, USA, 2008, WiSec ’08, pp. 214–219, ACM.

[16] Chris Karlof and David Wagner, “Secure routing in wireless sensor networks: Attacks and countermeasures,” in *First IEEE International Workshop on Sensor Network Protocols and Applications*, 2003, pp. 113–127.

[17] John R. Douceur, “The sybil attack,” in *1st International Workshop on Peer-to-Peer Systems*, 2002, IPTPS’02, pp. 251–260.

[18] James Newsome, Elaine Shi, Dawn Song, and Adrian Perrig, “The sybil attack in sensor networks: analysis & defenses,” in *Proceedings of the 3rd international symposium on Information processing in sensor networks*, New York, NY, USA, 2004, IPSN ’04, pp. 259–268, ACM.

[19] Wassim Znaidi, Marine Minier, and Jean-Philippe Babau, “An ontology for attacks in wireless sensor networks,” Research Report RR-6704, INRIA, 2008.

[20] Chris Karlof, Naveen Sastry, and David Wagner, “Tinysec: a link layer security architecture for wireless sensor networks,” in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA, 2004, SenSys ’04, pp. 162–175, ACM.

[21] Tieyan Li, Hongjun Wu, Xinkai Wang, and Feng Bao, “SenSec: Sensor security framework for TinyOS,” Tech. Rep., Institute for Infocomm Research, Singapore, 2005.

[22] Mark Luk, Ghita Mezzour, Adrian Perrig, and Virgil Gligor, “MiniSec: A secure sensor network communication architecture,” in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. 2007, IPSN ’07, pp. 479–488, ACM Press.

[23] An Liu and Peng Ning, "TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks," in *Proceedings of the 7th international conference on Information processing in sensor networks*, Washington, DC, USA, 2008, IPSN '08, pp. 245–256, IEEE Computer Society.

[24] "TinyOS," `http://www.tinyos.net`.

[25] Lander Casado and Philippas Tsigas, "Contikisec: A secure network layer for wireless sensor networks under the contiki operating system," in *Proceedings of the 14th Nordic Conference on Secure IT Systems: Identity and Privacy in the Internet Age*, Berlin, Heidelberg, 2009, NordSec '09, pp. 133–147, Springer-Verlag.

[26] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, Washington, DC, USA, 2004, LCN '04, pp. 455–462, IEEE Computer Society.

[27] Adrian Perrig, Robert Szewczyk, J.D̃. Tygar, Victor Wen, and David E. Culler, "Spins: security protocols for sensor networks," *Wirel. Netw.*, vol. 8, no. 5, pp. 521–534, September 2002.

[28] Devesh Jinwala, Dhiren Patel, and Kankar Dasgupta, "Optimizing the block cipher and modes of operations overhead at the link layer security framework in the wireless sensor networks," in *Proceedings of the 4th International Conference on Information Systems Security*, Berlin, Heidelberg, 2008, ICISS '08, pp. 258–272, Springer-Verlag.

[29] Morris Dworkin, *NIST Special Publication 800-38B, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*, National Institute of Standards and Technology, Computer Security Division, 2005.

[30] Chih-Chun Chang, S. Muftic, and D.J. Nagel, "Measurement of energy costs of security in wireless sensor nodes," in *ICCCN 2007: Proceedings of 16th International Conference on Computer Communications and Networks*, aug. 2007, pp. 95–102.

[31] Laurent Eschenauer and Virgil D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM conference on Computer and communications security*, New York, NY, USA, 2002, CCS '02, pp. 41–47, ACM.

[32] Haowen Chan, Adrian Perrig, and Dawn Song, “Random key predistribution schemes for sensor networks,” in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2003, SP ’03, pp. 197–, IEEE Computer Society.

[33] Donggang Liu, Peng Ning, and Rongfang Li, “Establishing pairwise keys in distributed sensor networks,” New York, NY, USA, February 2005, vol. 8, pp. 41–77, ACM.

[34] Wenliang Du, Jing Deng, Yunghsiang S. Han, Pramod K. Varshney, Jonathan Katz, and Aram Khalili, “A pairwise key predistribution scheme for wireless sensor networks,” *ACM Trans. Inf. Syst. Secur.*, vol. 8, pp. 228–258, May 2005.

[35] F. Delgosha and F. Fekri, “Threshold key-establishment in distributed sensor networks using a multivariate scheme,” in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, Barcelona, Spain, April 2006.

[36] Donggang Liu and Peng Ning, “Location-based pairwise key establishments for static sensor networks,” in *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, New York, NY, USA, 2003, SASN ’03, pp. 72–82, ACM.

[37] Dijiang Huang, Manish Mehta, Deep Medhi, and Lein Harn, “Location-aware key management scheme for wireless sensor networks,” in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, New York, NY, USA, 2004, SASN ’04, pp. 29–42, ACM.

[38] W. Du, J. Deng, Y.S. Han, S. Chen, and P.K. Varshney, “A key management scheme for wireless sensor networks using deployment knowledge,” in *23:rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*. IEEE, 2004, vol. 1.

[39] Hao Yang, Fan Ye, Yuan Yuan, Songwu Lu, and William Arbaugh, “Toward resilient security in wireless sensor networks,” in *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, New York, NY, USA, 2005, MobiHoc ’05, pp. 34–45, ACM.

[40] Zhen Yu and Yong Guan, “A key management scheme using deployment knowledge for wireless sensor networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, pp. 1411–1425, October 2008.

[41] L.B. Oliveira, H.C. Wong, M. Bern, R. Dahab, and A.A.F. Loureiro, "Secleach - a random key distribution solution for securing clustered sensor networks," in *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, july 2006, pp. 145 –154.

[42] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Washington, DC, USA, 2000, HICSS '00, pp. 8020–, IEEE Computer Society.

[43] Haowen Chan, "Pike: Peer intermediaries for key establishment in sensor networks," in *Proceedings of IEEE Infocom*, 2005, pp. 524–535.

[44] Sencun Zhu, Sanjeev Setia, and Sushil Jajodia, "Leap: efficient security mechanisms for large-scale distributed sensor networks," in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, NYC, NY, USA, 2003, pp. 62–72, ACM Press.

[45] Ross Anderson, Haowen Chan, and Adrian Perrig, "Key infection: Smart trust for smart dust," *Proceedings of the 13:th IEEE International Conference on Network Protocols*, vol. 0, pp. 206–215, 2004.

[46] Daniel Cvrcek and Petr Svenda, "Smart dust security – key infection revisited," *Electron. Notes Theor. Comput. Sci.*, vol. 157, pp. 11–25, May 2006.

[47] Matthew J. Miller and Nitin H. Vaidya, "Leveraging channel diversity for key establishment in wireless sensor networks," in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, Barcelona, Spain, April 2006, pp. 1–12.

[48] Leonardo B. Oliveira, Hao Chi Wong, Antonio A. F. Loureiro, and Ricardo Dahab, "On the design of secure protocols for hierarchical sensor networks," *Int. J. Secur. Netw.*, vol. 2, no. 3/4, pp. 216–227, 2007.

[49] Seyit A. Camtepe and Bülent Yener, "Key distribution mechanisms for wireless sensor networks: a survey," Tech. Rep., Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 2005.

[50] Junqi Zhang and Vijay Varadharajan, "Review: Wireless sensor network key management survey and taxonomy," *J. Netw. Comput. Appl.*, vol. 33, pp. 63–75, March 2010.

[51] Sherin M. Youssef, A. Baith Mohamed, and Mark A. Mikhail, "An enhanced security architecture for wireless sensor network," in *Proceedings of the 8th WSEAS international conference on Data networks, communications, computers*, Stevens Point, Wisconsin, USA, 2009, pp. 216–224, World Scientific and Engineering Academy and Society (WSEAS).

[52] Wei Chen, Xiaoshuan Zhang, Dong Tian, and Zetian Fu, "An identity-based authentication protocol for clustered zigbee network," in *Proceedings of the Advanced intelligent computing theories and applications, and 6th international conference on Intelligent computing*, Berlin, Heidelberg, 2010, ICIC'10, pp. 503–510, Springer-Verlag.

[53] Donggang Liu and Peng Ning, "Efficient distribution of key chain commitments for broadcast authentication in distributed sensor networks," in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2003)*, San Diego, California, USA, 2003, The Internet Society.

[54] Donggang Liu and Peng Ning, "Multilevel µtesla: Broadcast authentication for distributed sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 800–836, November 2004.

[55] D. Liu, P. Ning, S. Zhu, and S. Jajodia, "Practical broadcast authentication in sensor networks," in *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*, july 2005, pp. 118 – 129.

[56] Mark Luk, Adrian Perrig, and Bram Whillock, "Seven cardinal properties of sensor network broadcast authentication," in *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, New York, NY, USA, 2006, SASN '06, pp. 147–156, ACM.

[57] Zinaida Benenson, Nils Gedicke, and Ossi Raivio, "Realizing robust user authentication in sensor networks," in *Proceedings of Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, 2005.

[58] Canming Jiang, Bao Li, and Haixia Xu, "An efficient scheme for user authentication in wireless sensor networks," in *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 01*, Washington, DC, USA, 2007, AINAW '07, pp. 438–442, IEEE Computer Society.

[59] Kirk H. M. Wong, Yuan Zheng, Jiannong Cao, and Shengwei Wang, "A dynamic user authentication scheme for wireless sensor networks," in *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06) - Volume 01*, Washington, DC, USA, 2006, pp. 244–251, IEEE Computer Society.

[60] H.R. Tseng, R.H. Jan, and W. Yang, "An improved dynamic user authentication scheme for wireless sensor networks," in *IEEE Global Telecommunications Conference (GLOBECOM'07)*. IEEE, 2007, pp. 986–990.

[61] L.C. Ko, "A novel dynamic user authentication scheme for wireless sensor networks," in *Proceedings of IEEE International Symposium on Wireless Communication Systems (ISWCS '08).*, Reykjavik, Iceland, October 2008.

[62] Binod Vaidya, Joel J. Rodrigues, and Jong Hyuk Park, "User authentication schemes with pseudonymity for ubiquitous sensor network in ngn," *Int. J. Commun. Syst.*, vol. 23, pp. 1201–1222, September 2010.

[63] Waleed Ammar, Ahmed ElDawy, and Moustafa Youssef, "Secure localization in wireless sensor networks: A survey," *CoRR*, vol. abs/1004.3164, 2010.

[64] Loukas Lazos and Radha Poovendran, "Serloc: Robust localization for wireless sensor networks," *TOSN*, vol. 1, no. 1, pp. 73–100, 2005.

[65] Yingpei Zeng, Jiannong Cao, Jue Hong, Shigeng Zhang, and Li Xie, "Secmcl: A secure monte carlo localization algorithm for mobile sensor networks," in *Mobile Adhoc and Sensor Systems, 2009. MASS '09. IEEE 6th International Conference on*, oct. 2009, pp. 1054 –1059.

[66] Lingxuan Hu and David Evans, "Localization for mobile sensor networks," in *Proceedings of the 10th annual international conference on Mobile computing and networking*, New York, NY, USA, 2004, MobiCom '04, pp. 45–57, ACM.

[67] Honglong Chen, Wei Lou, and Zhi Wang, "A novel secure localization approach in wireless sensor networks," *EURASIP J. Wirel. Commun. Netw.*, vol. 2010, pp. 12:1–12:12, February 2010.

[68] Honglong Chen, Wei Lou, and Zhi Wang, "Conflicting-set-based wormhole attack resistant localization in wireless sensor networks," in *Proceedings of the 6th International Conference on Ubiquitous Intelligence and Computing*, Berlin, Heidelberg, 2009, UIC '09, pp. 296–309, Springer-Verlag.

[69] Honglong Chen, Wei Lou, Xice Sun, and Zhi Wang, "A secure localization approach against wormhole attacks using distance consistency," *EURASIP J. Wirel. Commun. Netw.*, vol. 2010, pp. 8:1–8:11, April 2010.

[70] Anindya Iqbal and M. Manzur Murshed, "Attack-resistant sensor localization under realistic wireless signal fading," in *Proceedings of the 2010 IEEE Wireless Communications and Networking Conference, WCNC 2010*, Sydney, Australia, April 2010, pp. 1–6, IEEE.

[71] Donggang Liu, Peng Ning, An Liu, Cliff Wang, and Wenliang Kevin Du, "Attack-resistant location estimation in wireless sensor networks," *ACM Trans. Inf. Syst. Secur.*, vol. 11, pp. 22:1–22:39, July 2008.

[72] Xiaoyan Li, "Designing localization algorithms robust to signal strength attacks," in *Proceedings of the Seventh Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON 2010*, Boston, Massachusetts, USA, June 2010, pp. 1–3, IEEE.

[73] M. Jadliwala, Sheng Zhong, S.J. Upadhyaya, Chunming Qiao, and J.-P. Hubaux, "Secure distance-based localization in the presence of cheating beacon nodes," *Mobile Computing, IEEE Transactions on*, vol. 9, no. 6, pp. 810 –823, june 2010.

[74] Qi Mi, John A. Stankovic, and Radu Stoleru, "Secure walking gps: a secure localization and key distribution scheme for wireless sensor networks," in *Proceedings of the Third ACM Conference on Wireless Network Security, WISEC 2010*, Hoboken, New Jersey, USA, March 2010, pp. 163–168, ACM.

[75] Sander Wozniak, Tobias Gerlach, and Guenter Schaefer, "Optimization-based secure multi-hop localization in wireless ad hoc networks," in *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*, Norbert Luttenberger and Hagen Peters, Eds., Dagstuhl, Germany, 2011, vol. 17 of *OpenAccess Series in Informatics (OASIcs)*, pp. 182–187, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[76] A. Boukerche, H.A.B. Oliveira, E.F. Nakamura, and A.A.F. Loureiro, "Secure localization algorithms for wireless sensor networks," *Communications Magazine, IEEE*, vol. 46, no. 4, pp. 96 –101, April 2008.

[77] Avinash Srinivasan and Jie Wu, *A Survey on Secure Localization in Wireless Sensor Networks*, 2007.

[78] Yingpei Zeng, Jiannong Cao, Jue Hong, Shigeng Zhang, and Li Xie, "Secure localization and location verification in wireless sensor networks: a survey," *The Journal of Supercomputing*, pp. 1–17, 2010, 10.1007/s11227-010-0501-4.

[79] 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.

[80] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava, "Timing-sync protocol for sensor networks," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, NYC, NY, USA, 2003, pp. 138–149, ACM Press.

[81] Miklos Maroti, Branislav Kusy, Gyula Simon, and Akos Ledeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys'04)*, NYC, NY, USA, 2004, pp. 39–49, ACM Press.

[82] Richard M. Karp, Jeremy Elson, Christos H. Papadimitriou, and Scott Shenker, "Global synchronization in sensornets," in *Proceedings of the 6th Latin American Symposium on Theoretical Informatics, LATIN'04*, Buenos Aires, Argentina, 2004, vol. 2976 of *LNCS*, pp. 609–624, Springer.

[83] 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, September 2005.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] Kun Sun, Peng Ning, and Cliff Wang, "Tinysersync: secure and resilient time synchronization in wireless sensor networks," in *Proceedings of the 13th ACM conference on Computer and communications security*, New York, NY, USA, 2006, CCS '06, pp. 264–277, ACM.

[89] David Sanchez, "Secure, accurate and precise time synchronization for wireless sensor networks," in *Proceedings of the 3rd ACM workshop on QoS and security for wireless and mobile networks*, New York, NY, USA, 2007, Q2SWinet '07, pp. 105–112, ACM.

[90] Saurabh Ganeriwal, Srdjan Capkun, and Mani B. Srivastava, "Secure time synchronization in sensor networks," *ACM Transactions on Information and Systems Security*, 2008.

[91] Jaap-Henk Hoepman, Andreas Larsson, Elad M. Schiller, and Philippas Tsigas, "Secure and self-stabilizing clock synchronization in sensor networks," *Theoretical Computer Science*, vol. 412, no. 40, pp. 5631–5647, 2011, Available online 16 April 2010.

[92] Fei Hu, Steve Wilson, and Yang Xiao, "Correlation-based security in time synchronization of sensor networks.," in *WCNC'08*, 2008, pp. 2525–2530.

[93] Hui Li, Yanfei Zheng, Mi Wen, and Kefei Chen, "A secure time synchronization protocol for sensor network," in *Proceedings of the 2007 international conference on Emerging technologies in knowledge discovery and data mining*, Berlin, Heidelberg, 2007, PAKDD'07, pp. 515–526, Springer-Verlag.

[94] Xin Hu, Taejoon Park, and K.G. Shin, "Attack-tolerant time-synchronization in wireless sensor networks," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, april 2008, pp. 41 –45.

[95] T. Roosta, Wei-Chieh Liao, Wei-Chung Teng, and S. Sastry, "Testbed implementation of a secure flooding time synchronization protocol," in *Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE*, 31 2008-april 3 2008, pp. 3157 –3162.

[96] Lin Chen and Jean Leneutre, "Toward secure and scalable time synchronization in ad hoc networks," *Comput. Commun.*, vol. 30, pp. 2453–2467, September 2007.

[97] Kasper Bonne Rasmussen, Srdjan Capkun, and Mario Cagalj, "Secnav: secure broadcast localization and time synchronization in wireless networks," in *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, New York, NY, USA, 2007, MobiCom '07, pp. 310–313, ACM.

[98] 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.

[99] Xiaojiang Du, M. Guizani, Yang Xiao, and Hsiao-Hwa Chen, "Secure and efficient time synchronization in heterogeneous sensor networks," *Vehicular Technology, IEEE Transactions on*, vol. 57, no. 4, pp. 2387 –2394, july 2008.

[100] A. Boukerche and D. Turgut, "Secure time synchronization protocols for wireless sensor networks," *Wireless Communications, IEEE*, vol. 14, no. 5, pp. 64 –69, october 2007.

[101] Ameer Ahmed Abbasi and Mohamed Younis, "A survey on clustering algorithms for wireless sensor networks," *Comput. Commun.*, vol. 30, no. 14-15, pp. 2826–2841, 2007.

[102] 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.

[103] 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.

[104] X. Wang, L. Yang, and K. Chen, "Sleach: Secure low-energy adaptive clustering hierarchy protocol for wireless sensor networks," *Wuhan University Journal of Natural Sciences*, vol. 10, no. 1, pp. 127–131, 2005, Cited By (since 1996): 3.

[105] P. Banerjee, D. Jacobson, and S.N. Lahiri, "Security and performance analysis of a secure clustering protocol for sensor networks," in *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, july 2007, pp. 145 –152.

[106] Gicheol Wang and Gihwan Cho, "Secure cluster head sensor elections using signal strength estimation and ordered transmissions," *Sensors*, vol. 9, no. 6, pp. 4709–4727, 2009.

[107] 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.

[108] 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.

[109] 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.

[110] Andreas Larsson and Philippas Tsigas, “Self-stabilizing (k,r)-clustering in wireless ad-hoc networks with multiple paths,” in *Proceedings of the 14th International Conference On Principles Of Distributed Systems (OPODIS 2010)*, Tozeur, Tunisia, December 2010, vol. 6490 of *Lecture Notes in Computer Science*, pp. 79–82, Springer.

[111] Andreas Larsson and Philippas Tsigas, “A self-stabilizing (k,r)-clustering algorithm with multiple paths for wireless ad-hoc networks,” in *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS 2011)*, Minneapolis, Minnesota, USA, June 2011, pp. 353–362, IEEE Computer Society.

[112] Suk-Bok Lee and Yoon-Hwa Choi, “A secure alternate path routing in sensor networks,” *Comput. Commun.*, vol. 30, pp. 153–165, December 2006.

[113] Jamil Ibriq and Imad Mahgoub, “A secure hierarchical routing protocol for wireless sensor networks,” in *Communication systems, 2006. ICCS 2006. 10th IEEE Singapore International Conference on*, oct. 2006, pp. 1 –6.

[114] Jian Yin and Sanjay Madria, “Secrout: A secure routing protocol for sensor networks,” in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 01*, Washington, DC, USA, 2006, AINA ’06, pp. 393–398, IEEE Computer Society.

[115] Jian Yin and Sanjay K. Madria, “Esecrout: An energy efficient secure routing for sensor networks,” *Int. J. Distrib. Sen. Netw.*, vol. 4, no. 2, pp. 67–82, 2008.

[116] Xiaojiang Du, M. Guizani, Yang Xiao, and Hsiao-Hwa Chen, “Two tier secure routing protocol for heterogeneous sensor networks,” *Wireless Communications, IEEE Transactions on*, vol. 6, no. 9, pp. 3395 –3401, september 2007.

[117] Sanjay Madria and Jian Yin, “Serwa: A secure routing protocol against wormhole attacks in sensor networks,” *Ad Hoc Netw.*, vol. 7, pp. 1051–1063, August 2009.

[118] Jing Deng, Richard Han, and Shivakant Mishra, "Insens: Intrusion-tolerant routing for wireless sensor networks," *Comput. Commun.*, vol. 29, pp. 216–230, January 2006.

[119] S. Kumar and S. Jena, "Scmrp: Secure cluster based multipath routing protocol for wireless sensor networks," in *Wireless Communication and Sensor Networks (WCSN), 2010 Sixth International Conference on*, dec. 2010, pp. 1 –6.

[120] Suraj Sharma and Sanjay Kumar Jena, "A survey on secure hierarchical routing protocols in wireless sensor networks," in *Proceedings of the 2011 International Conference on Communication, Computing &#38; Security*, New York, NY, USA, 2011, ICCCS '11, pp. 146–151, ACM.

[121] Xiaojiang Du, Yang Xiao, Hsiao-Hwa Chen, and Qishi Wu, "Secure cell relay routing protocol for sensor networks: Research articles," *Wirel. Commun. Mob. Comput.*, vol. 6, pp. 375–391, May 2006.

[122] Anthony D. Wood, Lei Fang, John A. Stankovic, and Tian He, "Sigf: a family of configurable, secure routing protocols for wireless sensor networks," in *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, New York, NY, USA, 2006, SASN '06, pp. 35–48, ACM.

[123] Mariano García-Otero, Theodore Zahariadis, Federico Álvarez, Helen C. Leligou, Adrián Población-Hernández, Panagiotis Karkazis, and Francisco J. Casajús-Quirós, "Secure geographic routing in ad hoc and wireless sensor networks," *EURASIP J. Wirel. Commun. Netw.*, vol. 2010, pp. 10:1–10:12, January 2010.

[124] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi, "In-network aggregation techniques for wireless sensor networks: a survey," *Wireless Communications, IEEE*, vol. 14, no. 2, pp. 70–87, April 2007.

[125] Lingxuan Hu and David Evans, "Secure aggregation for wireless networks," in *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, Washington, DC, USA, 2003, SAINT-W '03, pp. 384–, IEEE Computer Society.

[126] Jing Deng, Richard Han, and Shivakant Mishra, "Security support for in-network processing in wireless sensor networks," in *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, New York, NY, USA, 2003, SASN '03, pp. 83–93, ACM.

[127] Wenliang Du, Jing Deng, Yunghsiang S. Han, and Pramod Varshney, "A witness-based approach for data fusion assurance in wireless sensor networks," in *In Proceedings of the IEEE Global Telecommunications Conference*, 2003, pp. 1435–1439.

[128] Bartosz Przydatek, Dawn Song, and Adrian Perrig, "Sia: secure information aggregation in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA, 2003, SenSys '03, pp. 255–265, ACM.

[129] Sencun Zhu, Sanjeev Setia, Sushil Jajodia, and Peng Ning, "An interleaved hop-by-hop authentication scheme for filtering false data injection in sensor networks," in *IEEE Symposium on Security and Privacy (IEEE-SP'04)*, 2004.

[130] Harald Vogt, "Exploring message authentication in sensor networks," in *Proceedings of European Workshop on Security of Ad Hoc and Sensor Networks (ESAS)*. 2004, Springer-Verlag.

[131] Zhen Yu and Yong Guan, "A dynamic en-route scheme for filtering false data injection in wireless sensor networks," in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2005, SenSys '05, pp. 294–295, ACM.

[132] David Wagner, "Resilient aggregation in sensor networks," in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, New York, NY, USA, 2004, SASN '04, pp. 78–87, ACM.

[133] Haowen Chan, Adrian Perrig, and Dawn Song, "Secure hierarchical in-network aggregation in sensor networks," in *Proceedings of the 13th ACM conference on Computer and communications security*, New York, NY, USA, 2006, CCS '06, pp. 278–287, ACM.

[134] Keith B. Frikken and Joseph A. Dougherty, IV, "An efficient integrity-preserving scheme for hierarchical sensor aggregation," in *Proceedings of the first ACM conference on Wireless network security*, New York, NY, USA, 2008, WiSec '08, pp. 68–76, ACM.

[135] Mark Manulis and Jörg Schwenk, "Security model and framework for information aggregation in sensor networks," *ACM Trans. Sen. Netw.*, vol. 5, pp. 13:1–13:28, April 2009.

[136] Atsuko Miyaji and Kazumasa Omote, "Efficient and optimally secure in-network aggregation in wireless sensor networks," in *Proceedings of the 11th international conference on Information security applications*, Berlin, Heidelberg, 2011, WISA'10, pp. 135–149, Springer-Verlag.

[137] Claude Castelluccia, Aldar C-F. Chan, Einar Mykletun, and Gene Tsudik, "Efficient and provably secure aggregation of encrypted data in wireless sensor networks," *ACM Trans. Sen. Netw.*, vol. 5, pp. 20:1–20:36, June 2009.

[138] Shih-I Huang, Shiuhpyng Shieh, and J. D. Tygar, "Secure encrypted-data aggregation for wireless sensor networks," *Wirel. Netw.*, vol. 16, pp. 915–927, May 2010.

[139] Suat Ozdemir and Yang Xiao, "Integrity protecting hierarchical concealed data aggregation for wireless sensor networks," *Computer Networks*, vol. 55, no. 8, pp. 1735 – 1746, 2011.

[140] Jacques M. Bahi, Christophe Guyeux, and Abdallah Makhoul, "Efficient and robust secure aggregation of encrypted data in sensor networks," in *Proceedings of the 2010 Fourth International Conference on Sensor Technologies and Applications*, Washington, DC, USA, 2010, SENSORCOMM '10, pp. 472–477, IEEE Computer Society.

[141] C. Castelluccia, E. Mykletun, and G. Tsudik, "Efficient aggregation of encrypted data in wireless sensor networks," in *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*, july 2005, pp. 109 – 117.

[142] D. Westhoff, J. Girao, and M. Acharya, "Concealed data aggregation for reverse multicast traffic in sensor networks: Encryption, key distribution, and routing adaptation," *Mobile Computing, IEEE Transactions on*, vol. 5, no. 10, pp. 1417 –1431, oct. 2006.

[143] I. Rodhe and C. Rohner, "n-LDA: n-Layers data aggregation in sensor networks," in *Distributed Computing Systems Workshops, 2008. ICDCS '08. 28th International Conference on*, june 2008, pp. 400 –405.

[144] Rabindra Bista, Kyoung-Jin Jo, and Jae-Woo Chang, "A new approach to secure aggregation of private data in wireless sensor networks," in *Proceedings of the 2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, Washington, DC, USA, 2009, DASC '09, pp. 394–399, IEEE Computer Society.

[145] Yingpeng Sang, Hong Shen, Yasushi Inoguchi, Yasuo Tan, and Naixue Xiong, "Secure data aggregation in wireless sensor networks: A survey," *Parallel and Distributed Computing Applications and Technologies, International Conference on*, vol. 0, pp. 315–320, December 2006.

[146] Alessandro Sorniotti, Laurent Gomez, Konrad Wrona, and Lorenzo Odorico, "Secure and trusted in-network data processing in wireless sensor networks: a survey," *Journal of Information Assurance and Security*, vol. 2, no. 3, September 2007.

[147] Hani Alzaid, Ernest Foo, and Juan Gonzalez Nieto, "Secure data aggregation in wireless sensor network: a survey," in *Proceedings of the sixth Australasian conference on Information security - Volume 81*, Darlinghurst, Australia, Australia, 2008, AISC '08, pp. 93–105, Australian Computer Society, Inc.

[148] Suat Ozdemir and Yang Xiao, "Secure data aggregation in wireless sensor networks: A comprehensive overview," *Computer Networks*, vol. 53, no. 12, pp. 2022 – 2037, 2009.

[149] Edsger W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[150] Shlomi Dolev, *Self-Stabilization*, MIT Press, March 2000.

[151] 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.

[152] 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.

[153] Jaap-Henk Hoepman, Andreas Larsson, Elad M. Schiller, and Philippas Tsigas, "Secure and self-stabilizing clock synchronization in sensor networks," in *Prooceedings of the 9th International Symposium on Self Stabilization, Safety, And Security of Distributed Systems (SSS 2007)*. 2007, vol. 4838 of *Lecture Notes in Computer Science*, pp. 340 – 356, Springer-Verlag.

# 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*, volume 412, number 40, pages 5631–5647, 2011.

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

This work extends and improves work that appeared in:
Jaap-Henk Hoepman, Andreas Larsson, Elad M. Schiller, and Philippas Tsigas. "Secure and self-stabilizing clock synchronization in sensor networks." In *Prooceedings of the 9th International Symposium on Self Stabilization, Safety, And Security of Distributed Systems (SSS 2007)*. volume 4838 of *Lecture Notes in Computer Science*, pages 340–356, Springer, 2007.

# 2 Paper I: Secure and Self-stabilizing Clock Synchronization in Sensor Networks

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 capture 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]

[1]Lenzen 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.

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, $\rho_i$, is the first derivative of the clock value with respect to real time. Thus $\rho_i = \lim_{\tau \to 0}(C^i(t+\tau) - C^i(t))/\tau$. 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 $p_i$ by $\overrightarrow{G_i} \subseteq \{p_j \in P \mid 2R_{lb} \geq |L_i - L_j|\}$. 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.

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

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 $n \geq |\overrightarrow{G_i}|$ 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 $G_i = \overrightarrow{G_i}$ 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 $\textsf{\textit{omission}}(m_i)$ (which is associated with a particular message, $m_i$, sent by processor $p_i$) immediately after $\textsf{\textit{LBcast}}_i(m_i)$. 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 $\textsf{\textit{omission}}(m_i)$ 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 $\xi$.

The environment selects messages to remove due to ambient noise, but is limited by $\xi$ 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 $\xi$, 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 $\xi$ (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 $\xi$ 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 $\xi$ (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 $\beta$ broadcasts of the algorithm, where $\beta$ 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 $\alpha$ messages and the adversary can intercept at most $\beta$ of the algorithm's messages. Our system settings are comparable to the assumptions made by Gilbert et al. [11] on the ratio of $\beta/\alpha$. However, in contrast to the unknown $\beta$, 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 $\overrightarrow{G_i}$. 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 $\xi$ 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, $\ell$, 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

$\langle j, t_1, \{\langle k, \langle t_2, t_3, t_4\rangle\rangle\}\rangle$, where $\{\langle k, \langle t_2, t_3, t_4\rangle\rangle\}$ 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 $\langle t_s^i, \langle t_s^{j_1}, t_r^{j_1}\rangle, \langle t_s^{j_2}, t_r^{j_2}\rangle, \ldots\rangle$, 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 $\langle q_i, q_{j_1}, q_{j_2}, \ldots\rangle$, 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.

[3] We note that $BLog$ depends on the safety parameter, $\ell$, 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 $\rho_{\min}$ and $\rho_{\max}$ (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

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

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

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

External functions:
output($R$) : delivers record R to the upper layer
choose($S$) : uniform selection of an item from the set S
keys($v$) : the set of id:s that indexes v
enqueue($Q$) : adds an element to the end of the queue Q
dequeue($Q$) : removes the front element of the queue Q
size($Q$) : size of the queue Q
first($Q$) : least recently enqueued element in Q, number 0
last($Q$) : most recently enqueued element in Q
full($Q$) : whether queue Q is full
flush($Q$) : empties the queue Q
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:
border($t$) : $(D\text{-}cslot)u + t \mod T$
schedule($t$) : $cslot \cdot u + t \mod T$
leq($x, y$) : $(\exists\, b : 0 \leq b \leq 2\, BLog\, D\, u \wedge y \mod T = x + b \mod T)$
enq($q, m$) : {while full($q$) do dequeue($q$); enqueue($m$) }
G($j$) : keys($m[j].v$)

expire_s($q$,$t$): (* Expires data based on send times *)
  while size($q$) $> 0 \wedge \neg$ leq(first($q$).$s$, $t$) do
    dequeue($q$)
expire_r($q$,$t$): (* Expires data based on receive times − as expire_s but with .r instead of .s *)
check() : $\wedge$ {checkdata($m[j].v, j$) : $j \in$ keys($m[i].v$)}
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($v$)}
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($q$)$]$})
lclock($v$,$j$) : last($v[j].q$).$s$

(* Get response-record for $p_k$, for $p_j$ as the synchronizer *)
ts($s, j, k$) : {if ($\exists\, b_1^j, b_2^j, b_1^k, b_2^k$:
    $s = m[j].v[j].q[b_1^j].s = m[k].v[j].q[b_1^k].s \wedge$
    $m[k].v[k].q[b_2^k].s = m[j].v[k].q[b_2^j].s \wedge$
    leq($m[j].v[j].q[b_1^j].s, m[j].v[k].q[b_2^j].r$) $\wedge$
    leq($m[k].v[j].q[b_1^k].r, m[k].v[k].q[b_2^k].s$))
then return $\langle m[k].v[j].q[b_1^k].r, m[k].v[k].q[b_2^k].s, m[j].v[k].q[b_2^j].r \rangle$
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 $(s_1, s_2, \ldots)$, where each $s_i$ is the state of processor $p_i$ (including mes-

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

```
**Do forever, every** $u/2$
  **let** $cT = \mathsf{read}(native\_clock) + w$
  **if** $\neg\,(\mathsf{leq}(next\text{-}2Du, cT) \wedge \mathsf{leq}(cT, next{+}u))$ **then**
    $next \leftarrow cT$
  $\mathsf{expire\_s}(m[i].v[i].q, cT)$
  $\forall\, j \in \mathsf{G}(i)\setminus\{i\}$ **do** $\mathsf{expire\_r}(m[i].v[j].q, cT)$
  **if** $\neg\, \mathsf{check}()$ **then**
    $\forall\, j,k \in P$ **do** $\mathsf{flush}(m[j].v[k].q)$
  **if** $\mathsf{leq}(next, cT) \wedge \mathsf{leq}(cT, next + u)$ **then**
    **let** $s = \mathsf{first}(m[i].v[i].q).s$
    output $\langle i, s, \{\langle j, \mathsf{ts}(s, i, j)\rangle : j \in \mathsf{G}(i)\setminus\{i\}\}\rangle$
    $\mathsf{enq}(m[i].v[i].q, \langle cT, \perp\rangle)$
    $(next, cslot) \leftarrow (\mathsf{border}(next), \mathsf{choose}([0, D\text{-}1]))$
    $next \leftarrow \mathsf{schedule}(next)$
    $\mathsf{LBcast}(m[i])$

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

**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* $E = c[0], a[0], c[1], a[1], \ldots$ 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 $p_j \in \overrightarrow{G_i}$ 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 $\xi$ 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 $\Gamma_i$ be the set of all execution prefixes, $E_{\Gamma_i}$, such that, within the first $\mathcal{R}$ broadcasting rounds of $E_{\Gamma_i}$, (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 $\Gamma_i$.*

The proof of Theorem 2.1 (Section 2.5.3, page 93) 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 $\ell$ is a predefined constant and $R = \lceil \xi \frac{\ell + \log_2((\lceil \rho_{\max}/\rho_{\min} \rceil + 1)n)}{-\log_2(1-1/e)} \rceil$ 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 $T > 4\mathcal{R}$. 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 $|\overrightarrow{G_i}| = n$. Furthermore, in the worst-case scenario $|G_i| = |\overrightarrow{G_i}| = n$ 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 $|\overrightarrow{G_i}| < n$, 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 settings 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*

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

**Proof:** It is well known that

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

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

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

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

$$1 - \frac{3}{3n}. \tag{2.4}$$

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

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

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

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

■

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 $p^R$:

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

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

$$X=\sum_{i=1}^{n} X_i. \tag{2.8}$$

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]:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] \leq \sum_{i=1}^{n} p^R = np^R. \tag{2.9}$$

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

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

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

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

■

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 $b^j_{k'}$ overlapping with bins $b^i_{k-1}$, $b^i_k$ and $b^i_{k+1}$. 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,

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

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 $\xi$ 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 $\xi$ consecutive throws. Omission attacks are incorporated into the $\xi$ 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 $\xi R$ 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*

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

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

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

$$p = 1 - \frac{1}{e}. \tag{2.16}$$

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 $|\overrightarrow{G_i}| < n$.)

### 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 $\alpha_i = (\textsf{leq}(next_i - 2Du, cT_i) \wedge \textsf{leq}(cT_i, next_i)$ 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 $(\textsf{leq}(C^i - u, cT_i - w) \wedge \textsf{leq}(cT_i - w, C^i))$ 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 $\textsf{leq}(t - u, t) \wedge \textsf{leq}(t, t)$ 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 $a'_i \in E$, which includes a complete iteration of the do-forever loop. Since the period between $a_i$ and $a'_i$ is at most $u/2$, we have that $t' - t < u/2$. Therefore $\textsf{leq}(C^i - u, cT_i - w)$ holds in $c'$ as $\textsf{leq}(C^i, cT_i - w)$ holds in $c$. It also follows that $\textsf{leq}(cT_i - w, C^i))$ 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 $\alpha_i$ 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 $\alpha_i$ holds in $c$.*

**Proof:** Among the lines 75 to 80, only lines 78 to 79 can change the values of $\alpha_i$. 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 $\textsf{\textit{leq}}(cT_i, t_1 + u)$ holds before line 78. Since $u \leq A$, it holds that $\textsf{\textit{leq}}(cT_i, t_1 + A)$, and therefore $\textsf{\textit{leq}}(cT_i, t_2)$ holds.

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

We show that, starting from an arbitrary configuration, any fair execution researches 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 $\alpha_i$ does not hold in the starting configuration of $E$. We show that, within the first complete iteration of lines 67 to 80, the predicate $\alpha_i$ 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 $\alpha_i$ 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 $\neg(\textsf{\textit{leq}}(t - 2Du, t) \land \textsf{\textit{leq}}(t, t))$ is false for any $t$, immediately after the execution of line 69, the predicate $\neg(\textsf{\textit{leq}}(next_i - 2Du, cT_i) \land \textsf{\textit{leq}}(cT_i, next_i))$ does not hold. Moreover, the predicate in line 74 holds, since $\textsf{\textit{leq}}(t, t + u)$ 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, $\alpha_i$ 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 $\alpha_i$ 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 $\alpha_i$ holds in configuration $c'$ that immediately follows $a_i$. Lines 83 to 87 do not change the value of $\alpha_i$. By Claim 2.5, if $a_i$ executes lines 75 to 80 within one complete iteration, then $\alpha_i$ 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 $t_2 = cT_i$ in $c'$. According to Lemma 2.3, and by the fairness of $E$, we have that $t_2 - t_i \bmod T < u$. 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 $\alpha_i$ is true in $c$, it holds that $\mathit{leq}(A, t_1) \wedge \mathit{leq}(t_1, B)$. We claim that $\mathit{leq}(A, t_2) \wedge \mathit{leq}(t_2, B)$. Since $\mathit{leq}(t_1, B)$ in $c$, we have that $\mathit{leq}(t_2, B + t_2 - t_1)$ 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 $\mathit{leq}(t_1, B)$ and $t_2 - t_1 \bmod T < u$ the predicate in line 74 does not hold iff $\mathit{leq}(t_2, B)$. Furthermore, we have that $\mathit{leq}(A, t_1)$, $\mathit{leq}(t_1, B)$, and $\mathit{leq}(t_2, B)$. As $0 < t_2 - t_1 \bmod T < u$ we have that $\mathit{leq}(A, t_2)$. Thus, $c'$ is safe as $\alpha_i$ 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 $next_i + Du \mod T$.

The period of length $Du$ that begins at $r$ and ends at $r' \mod T$ is divided in $D$ timeslots of length $u$. A timeslot begins at time $r + xu \mod T$ and ends at time $r + (x+1)u \mod T$ 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 $\overrightarrow{G_i}$ 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, $|G_i| = |\overrightarrow{G_i}| = n$. 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.

Ganeriwal 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 $\epsilon$ is a safety constant (see [8]). We note that both approaches are applicable to our work. We further note that a processor $p_k \in G_j \cap G_j$ 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 Papatriantafilou 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, "Tinysersync: 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

## A self-stabilizing (k,r)-clustering algorithm with multiple paths for wireless ad-hoc networks

In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS 2011)*,
Minneapolis, Minnesota, USA, June 2011.

This work extends and improves work that appeared in:
Andreas Larsson and Philippas Tsigas. “Self-stabilizing (k,r)-clustering in wireless ad-hoc networks with multiple paths.” In *Proceedings of the 14th International Conference On Principles Of Distributed Systems (OPODIS 2010)*, volume 6490 of *Lecture Notes in Computer Science*, pages 79–82, Springer, 2010.

3

# Paper II: A Self-stabilizing (k,r)-clustering Algorithm with Multiple Paths for Wireless Ad-hoc Networks

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, for references to the area in general, we point to the survey of the area with regard to wireless ad-hoc networks by Chen, Liestam and Liu in [2] and the survey by Abbasi and Younis in [3] for wireless sensor networks. 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, algorithms provide an approximation. The (1,r)-clustering problem, a subset of the (k,r)-clustering problem, can be formulated as a classical set cover problem. This was shown to be NP complete in [4]. 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. Johnen and Nguyen present a self-stabilizing (1,1)-clustering algorithm that converges fast in [5]. Dolev and Tzachar tackle a lot of organizational problems in a self-stabilizing manner in [6]. As part of this work they present a self-stabilizing (1,r)-clustering algorithm. Caron, Datta, Depardon and Larmore present a self-stabilizing (1,r)-clustering in [7] that takes weighted graphs into account.

There is a number of papers that do not have self-stabilization in mind. Fu, Wang and Li consider the (k,1)-clustering problem in [8]. In [9] 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 [10].

Other algorithms do not take the cluster head approach. In [11], 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. We also present experimental results on how the algorithm copes with changes to the topology and on how our results compares with global optima.

Some initial ideas that lead to these results was previously published in [12]. It is a brief announcement with few technical details and without any proofs or experimental results.

### 3.1.2 Document Structure

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 affects 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. In our proofs for convergence times of our algorithm we use an assumption of synchronized timers. Synchronized timers is *not* an assumption of the algorithm itself and is *not* needed for the algorithm to work correctly. Furthermore, we demonstrate experimentally that it does not significantly affect convergence times either.

## 3.3 Self-stabilizing Algorithm for $(k, r)$-clustering

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.

```
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 $\in$ {HEAD, ESCAPING, 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 $\in$ {SLEEP, INIT, FLOOD, HOPE} :
    State for escape attempts. Initially set to SLEEP.
  heads : Set of Id:s. Initially set to $\emptyset$.
  S & Z: Sets of $<$ Id,State$>$ tuples.
    Initially set to $\{< i,state >\}$.

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($A$): $\{< j,s,t > \in A : t = \max_\tau \{\tau : < j,s,\tau > \in A\}\}$
  cdj($B$): $\{< j,t > \in B : t = \max_\tau \{\tau : < j,\tau > \in B\}\}$
```

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

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 of the algorithm is for cluster heads to constantly broadcast the fact that they are cluster heads and for all nodes to constantly broadcast a list of nodes they consider to be cluster heads. This list of cluster heads consists both of nodes that are known to be cluster heads and, additionally, nodes that are elected to become cluster heads. The content of the broadcasts are forwarded $r$ hops, but in an aggregated form to keep message sizes down. The election process might establish too many cluster heads. Therefore, there is a mechanism for cluster heads to drop their cluster head roles, to *escape*, eventually establishing a local minimum of cluster heads forming a total (k,r)-dominating set (or, if not possible given the topology, fulfilling $|C_j^r| \geq k_j$ for any node $p_j$). The choice of which nodes to pick when electing cluster heads is based on node ID in order to limit the number of unneeded cluster heads that are elected when new cluster heads are needed.

```
on step phase:
  if $timer < 0 \lor timer \geq T$-1
    $timer \leftarrow 0$
  else
    $timer \leftarrow timer + 1$
  $S \leftarrow Z$
  $heads \leftarrow \{j : <j, \text{HEAD}> \in S\}$
  /* Escaping */
  if state in {HEAD, ESCAPING}
    updateestate()
    if $estate = \text{INIT} \land state = \text{HEAD} \land |heads| > k$
      $state \leftarrow$ ESCAPING
      $heads \leftarrow heads \setminus \{i\}$
    else if $estate = \text{SLEEP} \land state = \text{ESCAPING}$
      $state \leftarrow$ SLAVE
  if $state = \text{SLAVE}$
    $estate \leftarrow$ SLEEP
    $estart \leftarrow 0$
  /* Add heads */
  if $|heads| < k$
    let $a = k - |heads|$
    let $A = \{j : <j, \cdot> \in S\} \setminus heads$
    $heads \leftarrow heads \cup \text{smallest}(a, A)$
  /* Join and send state */
  for each $j \in heads$
    if $j \neq i$
      forwardjoin($<j, r>$)
    else
      $state \leftarrow$ HEAD
  $Z \leftarrow \{<i, state>\}$
  sendstate($<i, state, r>$)

function updateestate:
  if $timer = 0$
    $estart \leftarrow uniformlyrandom(\{0, 1, \ldots T\text{-}2r\text{-}2\})$
  if $timer \in [0, estart\text{-}1]$:
    $estate \leftarrow$ SLEEP
  else if $timer \in [estart, estart]$
    $estate \leftarrow$ INIT
  else if $timer \in [estart+1, estart+2r\text{-}1]$
    $estate \leftarrow$ FLOOD
  else if $timer \in [estart+2r, estart+2r]$
    $estate \leftarrow$ HOPE
  else if $timer \in [estart+2r+1, T\text{-}1]$
    $estate \leftarrow$ SLEEP

function receivedstate($<j, jstate, ttl>$), $i \neq j$:
  $js \leftarrow jstate$
  if $js = \text{ESCAPING} \land j \in heads$
    if $|heads| \leq k$
      $js \leftarrow$ HEAD
    else
      $heads \leftarrow heads \setminus \{j\}$
  let $ss = \{s : <j, s> \in Z\} \cup \{js\}$
  if HEAD $\in ss$:
    $js \leftarrow$ HEAD
  else if ESCAPING $\in ss$
    $js \leftarrow$ ESCAPING
  else
    $js \leftarrow$ SLAVE
  $Z \leftarrow \{<o, s> : <o, s> \in Z \land o \neq j\} \cup \{<j, js>\}$

  $ttl \leftarrow \max(1, \min(r, ttl))$
  if $ttl > 1$:
    forwardstate($<j, jstate, ttl\text{-}1>$)

function receivedjoin($<j, ttl>$):
  $ttl \leftarrow \max(0, \min(r, ttl))$
  if $j = i \land estate \notin \{\text{INIT, FLOOD}\}$
    $state \leftarrow$ HEAD
  else if $ttl > 1$
    forwardjoin($<j, ttl\text{-}1>$)

on LBrecv($<j, jstateset, jjoinset>$):
  for each $<o, ostate, ottl> \in jstateset$
    if $o \neq i$:
      receivedstate($<o, ostate, ottl>$)
  for each $<o, ostate, ottl> \in jjoinset$
    receivedjoin($<o, ottl>$)

function forwardstate(tuple):
  $stateset \leftarrow stateset \cup tuple$

function forwardjoin(tuple):
  $joinset \leftarrow joinset \cup tuple$

function sendstate(tuple):
  forwardstate(tuple)
  $stateset \leftarrow \text{cds}(stateset)$
  $joinset \leftarrow \text{cdj}(joinset)$
  LBcast($<i, stateset, joinset>$)
  $stateset \leftarrow \emptyset$
  $joinset \leftarrow \emptyset$
```

**Figure 3.2:** *Pseudocode for the self-stabilizing clustering algorithm.*

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 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 it 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 in section 3.4.3.

**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

In this section we build up a case showing that the algorithm will elect enough cluster heads. We show that nodes get to know their neighborhood (Lemma 3.1), that they get to know the state of nodes in their neighborhoods (Lemma 3.2), that cluster heads are elected (Lemmas 3.3, 3.4 and 3.5). Finally, in Theorem 3.1, we show that within a $O(r)$ rounds each node in the network have enough cluster heads within $r$ hops (topology allowing).

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

**Lemma 3.1** *Assume that round $s$ and all following rounds are legal. For any node $p_i$, $\{p_j : < p_j, \cdot > \in S_i\} = G_i^r$ 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 $ttl$ set to $r$. The $ttl$ 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 $ttl$ 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 $G_i^r \subseteq \{p_j : < p_j, \cdot > \in S_i\}$ 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 $ttl$ 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 $G_i^r \supseteq \{p_j : < p_j, \cdot > \in S_i\}$ in the step phase of round $s + r$ and thereafter. ■

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

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

**Proof:** Node $p_i$ sends out its state with a $ttl$ of $r$ in each round (line 31). Nodes that receive this state message with a $ttl$ greater than $1$ will forward the state with a $ttl$ 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 $ttl$ of $r+t$. Furthermore a state $\sigma'$ 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 $\sigma$ 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. In this setting we will show that within a finite number of rounds we will have $|C_i^r| \geq k_i$ for any node $p_i$. Later on we will lift this restriction and show that $|C_i^r| \geq k_i$ will still hold even when regarding the more general case.

**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 a $ttl$ of $r$ 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 receive 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 $\sigma$ 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 $\sigma$.

*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. ■

In the following Lemma we show that a node that is continuously wanted as a cluster head eventually becomes one.

**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 $\widehat{A}$. 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 $state_i =$ 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 $\widehat{A}$ 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 $y + 1, y + 2, \ldots$, 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 $y, y + 1, \ldots$ 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. We show that a cluster head node that is not needed can escape the cluster head responsibility if not interfered by other escape attempts (Lemma 3.6). We show that an unneeded cluster head node escapes within $O(gr)$ rounds with high probability under assumptions of synchronized timers (Lemma 3.7). Finally, in Theorem 3.2, we show that with high probability the entire network reaches a local minimum within $O(gr \log n)$ rounds. We begin by looking at the escape of an uninterfered node.

**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 $\sigma$ be the $state_i$ that is sent out by $p_l$ in round $x+2r+1$. We know that $\sigma \neq$ 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 $\sigma$ 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 $\sigma$ and only $\sigma$ 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 $\sigma =$ HEAD) or none of them have $p_l \in heads_j$ (if $\sigma =$ 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 $state_i =$ 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 $timer_i = timer_j$ for all pair of nodes $p_i, p_j \in \mathcal{P}$ for all legal rounds.*

Under the added assumption of synchronized timers we show that an unneeded cluster head node either escapes within $O(gr)$ rounds, unless it becomes needed due to other escaped cluster heads.

**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 $timer_i = 0$ 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 $\Pi_i^t$ 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 $p_l \in G^{2r}$ 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 $\Pi_i^t$. Then we get

$$\begin{aligned}
P[A_i^t] &\geq (1-\rho)^{g-1} = [\mu := \frac{1}{\rho}] \\
&= \left(\left(1 - \frac{1}{\mu}\right)^{\mu-1}\right)^{(g-1)/(\mu-1)} \\
&> \left(\frac{1}{e}\right)^{(g-1)/(\mu-1)} = \exp\left(-\frac{g-1}{\mu-1}\right) \\
&= \exp\left(-\frac{g-1}{\frac{T-2r-1}{3r+1} - 1}\right). \qquad (3.1)
\end{aligned}$$

We can simplify this, using the fact that $r \geq 1$, to get that for $T = 8gr$ we have $P[A_i^t] > 1/2$.

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 $\beta$ periods $\Pi^0$ 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

$$\begin{aligned} P[A_i] =& P\left[\bigcup_{t=0}^{\beta-1} A_i^t\right] = 1 - P\left[\bigcap_{t=0}^{\beta-1} \bar{A}_i^t\right] = 1 - \prod_{t=0}^{\beta-1} P[A_i] \\ >& 1 - \prod_{t=0}^{\beta-1} \frac{1}{2} = 1 - 2^{-\beta}. \end{aligned} \tag{3.2}$$

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

From Theorem 3.1 we got that all nodes $p_i$ have at least $k_i$ cluster heads within $r$ hops in $2r$ rounds after an arbitrary configuration.

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 $\Pi^t$, $x^t$ 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 $\Pi^0$ to $\Pi^{\beta-1}$.

We get that

$$\begin{aligned}P[A] =& 1 - P[\bar{A}] = 1 - P[\bigcup_{p_i \in \mathcal{P}} \bar{A}_i] \geq \text{[Boole's inequality]}\\ \geq& 1 - \sum_{p_i \in \mathcal{P}} P[\bar{A}_i] \geq \text{[Lemma 3.7]} \geq 1 - \sum_{p_i \in \mathcal{P}} 2^{-\beta}\\ =& 1 - n2^{-\beta} = 1 - 2^{\log n - \beta} = 1 - 2^{-\alpha}. \end{aligned} \quad (3.3)$$

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 $t$. ■

## 3.4.3 Message Complexity

We now show the message complexity for the algorithm.

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

**Proof:** By Lemma 3.1 we have that for any node $p_i$, $\{p_j : < p_j, \cdot > \in S_i\} = G_i^r$ 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 $O(\log n + \log r)$. Therefore, the size, in bits, of the message sent by a node $p_i$ in any round $\geq s + r$ is in $O(|G_i| \cdot (\log n + \log r))$. ■

## 3.5 Discussion

We prove convergence within $O(gr \log n)$ rounds with high probability under the assumption of independent rounds (apart from the obvious dependence that nodes that escape are out of the contention for uninterfered escape attempts). To see if the timers really need to be synchronized to achieve this performance we did simulations of the algorithm for various settings of $k$ and $r$ and network densities. We placed $n$ nodes with a communication radius of 1 uniformly at random in a 5 by 5 rectangular area, with varying $n$ for different experiments.

From our experiments we concluded 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.

For the rest of the experiments we did a small change to the algorithm. A node that is added to the network do not elect new cluster head nodes for the

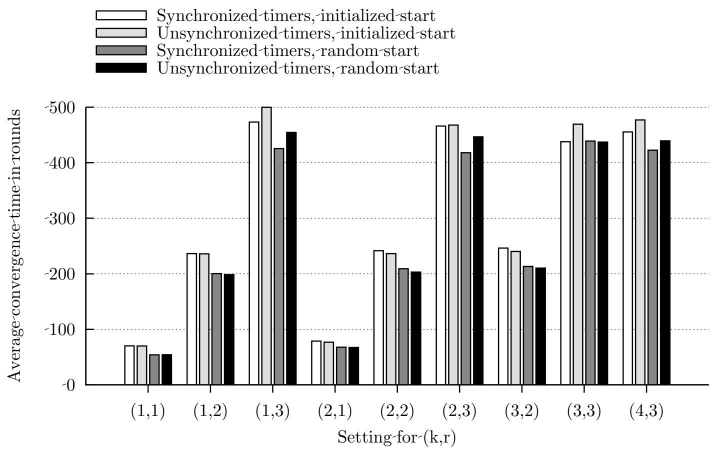

**Figure 3.3:** *Simulation results of the algorithm indicating that synchronization of timers is not needed. Here $T = gr/2$, $n = 39$.*

first $r$ full rounds. It performs all parts of the algorithm except that the condition in line 20 is always regarded as false in those rounds and lines 21 to 23 are not executed. It is trivial to make this mechanism self-stabilizing.

In Fig. 3.4 we can see the convergence behavior of the set of cluster towards local minima over time. The same trend can be seen for other choices of $k$ and $n$.

We have performed experiments to investigate the range of possible results regarding the (k,r)-dominating sets generated by our algorithm on random graphs. We compare the global minima with results given by our algorithm and with the worst (i.e., largest) possible local minima in Fig. 3.5. The results of our algorithm is in general placed in the middle between the global minima and the worst possible local minima. We can also see that even the worst possible local minima are still quite close to the global minima. Thus even if our algorithm is "unlucky" it provides a good result. This trend holds true for other choices of $n$ and $k$ as well.

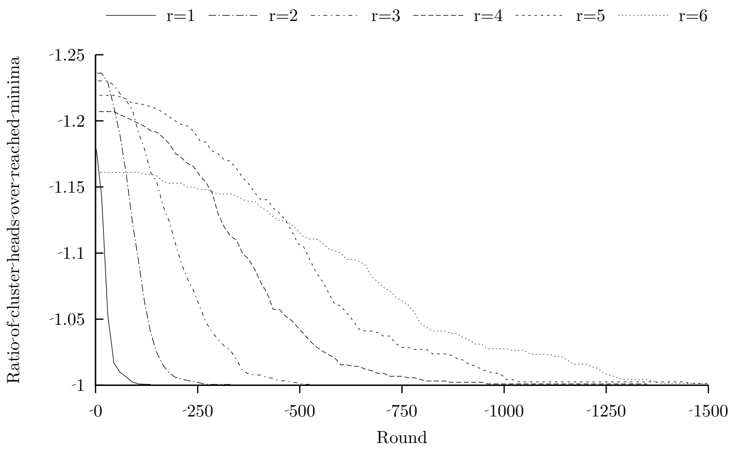

**Figure 3.4:** *Cluster head overhead over time as a ratio between number of cluster heads in given round over the eventually reached local minimum number of cluster heads. Here* $T = gr$, $k = 2$ *and* $n = 39$.

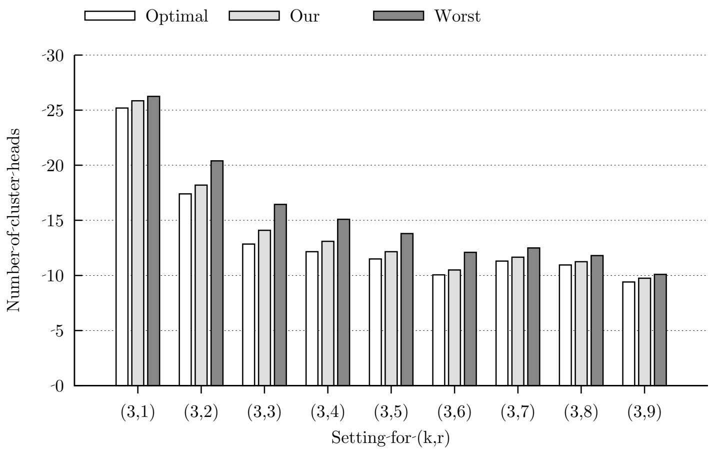

**Figure 3.5:** *Comparison between global optima, results of our algorithm and worst possible local minima for* $n = 31$

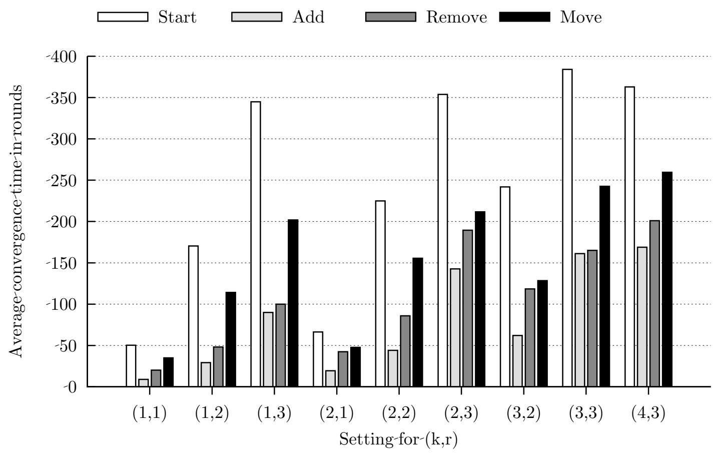

**Figure 3.6:** *Convergence times from a fresh start, after 5% node additions, after 5% node removes and after 5% node moves. Here $T = gr$ and $n = 39$.*

We also performed experiments on recovery from small changes to the topology from a converged state. The convergence times from a newly started network (“Start”) is compared in Fig. 3.6 with the convergence times after a change to a initially converged network. We investigate $5\%$ added nodes (“Add”), $5\%$ removed nodes (“Remove”) or $5\%$ moved nodes (“Move”). We achieve similar results for other choices of $n$ as well.

We can see that the least obtrusive change to the topology is added nodes. The chance is good that a new node ends up in a position in the network were there already is enough or close to enough cluster heads already. Remove is more expensive than add. Cluster heads could be among the nodes that are removed. Additionally removed nodes can also have been used as links between nodes and their cluster heads. A move is like both a remove and an add (without the rounds of abstaining from electing new cluster heads). Therefore, it is anticipated that this case converges slower than the ones with only adds or only removes.

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 applications 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, [13]) or node id:s for nodes that are not within $G_c^r$ (wormhole attacks, [14]) 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 or having them repeatedly go on and off cluster head duty over time by alternating between sending joins and letting the node escape. 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 bottleneck 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 $G_c^{2r}$ 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 [15] 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.

## 3.6 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 proved 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] Ameer Ahmed Abbasi and Mohamed Younis, “A survey on clustering algorithms for wireless sensor networks,” *Comput. Commun.*, vol. 30, no. 14-15, pp. 2826–2841, 2007.

[4] R. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds., pp. 85–103. Plenum Press, 1972.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Andreas Larsson and Philippas Tsigas, “Self-stabilizing (k,r)-clustering in wireless ad-hoc networks with multiple paths,” in *OPODIS’10, 14th International Conference On Principles Of Distributed Systems*, Tozeur, Tunisia, December 2010.

[13] 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.

[14] 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.

[15] 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.

# PAPER III

Andreas Larsson, Philippas Tsigas

## Self-stabilizing (k,r)-clustering in Clock Rate-limited Systems

Technical Report no. 2012:05, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2012.

This work, in a shortened form because of space constraints, will appear in: Andreas Larsson and Philippas Tsigas. “Self-stabilizing (k,r)-clustering in clock rate-limited systems.” In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2012)*, volume ?? of *Lecture Notes in Computer Science*, pages ??–??, Springer, 2012.

# 4 Paper III: Self-stabilizing (k,r)-clustering in Clock Rate-limited Systems

Wireless Ad-hoc networks are distributed systems that often reside in error-prone environments. Self-stabilization lets the system recover autonomously from an arbitrary system 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 a 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 assumes a bound on clock frequency differences and a limited guarantee on message delivery. It 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 $O(r\pi\lambda^3)$ time from an arbitrary system configuration, where $\pi$ is a limit on message loss and $\lambda$ is a limit on pulse rate differences. The set of cluster heads stabilizes, with high probability, to a local minimum within $O(r\pi\lambda^4 g \log n)$ time, where $n$ is the size of the network and $g$ is an upper bound on the number of nodes within $2r$ hops.

## 4.1 Introduction

Starting from an arbitrary system state, self stabilizing algorithms let a system stabilize to, and stay in, a consistent system state [1]. There are many reasons why a system could end up in an inconsistent system 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, the trade off is that self-stabilization often comes with increased costs. A self-stabilizing algorithm can never stop because it is not known in advance when temporary faults occur. Nevertheless, as long as all assumptions hold, it can converge to stable result, or, after convergence, stay within a set of acceptable states. Moreover, there are often overheads in the algorithm tied to the need to recover from arbitrary system states. They can be additional computations, larger messages, larger data structures or longer required times to achieve certain goals.

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 into reports to decrease the number of individual messages that needs to be routed through the network, e.g., aggregating sensor readings in a wireless sensor 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 reduce overall power consumption.

Clustering is a well studied problem. Due to space constraints, for references to the area in general, we point to the survey of the area with regard to wireless ad-hoc networks by Chen, Liestam and Liu in [2] and the survey by Abbasi and Younis in [3] for wireless sensor networks. In this paper we focus on self-stabilization, redundancy and 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 as possible. To find the global minimum number of cluster heads is in general computationally hard, and algorithms usually provide approximations. The (1,r)-clustering problem, a subset of the (k,r)-clustering problem, can be formulated as a classical set cover problem. This was shown to be NP complete in [4]. 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. Johnen and Nguyen present a self-stabilizing (1,1)-clustering algorithm that converges fast in [5]. Dolev and Tzachar tackle a lot of organizational problems in a self-stabilizing manner in [6]. As part of this work they present a self-stabilizing (1,r)-clustering algorithm. Caron, Datta, Depardon and Larmore present a self-stabilizing (1,r)-clustering in [7] that takes weighted graphs into account. Self-stabilization in systems with unreliable communications was introduced in [8]. In [9] a self-stabilizing (k,1)-clustering algorithm, that can cope with message loss, is presented.

There is a number of papers that do not have self-stabilization in their settings. Fu, Wang and Li consider the (k,1)-clustering problem in [10]. In [11] 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 [12].

Other algorithms do not take the cluster head approach. In [13], 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.

### 4.1.1 Our Contribution

We have constructed a self-stabilizing $(k, r)$-clustering algorithm for ad-hoc networks that can deal with message loss, as long as at least one out of $\pi$ consecutive broadcasts are successful, and that uses unsynchronized pulses, for which the ratios between pulse rates are limited by a factor $\lambda$. The algorithm makes sure that, within $O(r\pi\lambda^3)$ time, 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 $O(r\pi\lambda^4 g \log n)$ time 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 presented the first distributed self-stabilizing (k,r)-clustering in [14]. There, the system settings assumed perfect message transfers and lock step synchronization of the nodes. The current article is a further development of that work and the main idea of the algorithm is the same. The unreliable communication media, the unsynchronized nodes and the introduction of a veto mechanism to speed up convergence, all have made the current algorithm quite different, yet clearly related to the one in [14]. We present correctness proofs for quick selection of enough cluster heads ($k$ cluster heads within $r$ hops when possible) and that the set of cluster heads converges towards a local minimum and stays at that local minimum. This includes an upper bound on the time it takes, with high probability, for that convergence to happen. Furthermore, we also present experimental results on the convergence of the algorithm and how it copes with changes to the topology.

### 4.1.2 Document Structure

The rest of the paper is organized as follows. In section 4.2 we introduce the system settings. Section 4.3 describes the algorithm. Section 4.4 gives the overview of the proofs of the algorithm. We discuss experimental results, security and redundancy in Section 4.5.

## 4.2 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 $p_i$ and $\hat{G}_i^r = G_i^r \setminus \{p_i\}$. Let $g \geq \max_j |G_j^{2r}|$ be a bound, known by the nodes, on the number of nodes within $2r$ hops from any node.

Nodes are driven by a pulse going off every 1 time unit (with respect to its local clock). Pulses are not synchronized between nodes. The pulse frequency, in real time, of a node $p_i$ is denoted $\rho_i$. For any pair of nodes $p_i$ and $p_j$ the ratio $\rho_i/\rho_j \leq \lambda$, a value is a known to the nodes. Without loss of generality we assume that the frequency of the slowest clock in the system is 1 and thus the clock frequency of any node $p_i$ is in $[1, \lambda]$.

Among $\pi$ successive messages sent from one node there is at least one message, such that all immediate neighbors $p_j \in N_i$ receive that particular message. Such a message is called a *successful broadcast*. The nodes know the value of $\pi$. Apart from that assumption, messages from a node $p_i$ can be lost, be received by a subset of $N_i$, or received by all nodes in $N_i$.

## 4.3 Self-stabilizing Algorithm for $(k, r)$-clustering

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

```
Constants, and variables:
  i : Constant id of executing processor.
  $T$, $T_{cool}$, $T_{flood}$, $\kappa$ : Constants derived from r, k, $\lambda$ and $\pi$. See Definition 4.4.
  state $\in$ {HEAD, ESCAPING, SLAVE} : The state of the node.
  timer : Integer. Timer for escape attempts.
  estart : Integer. The escape schedule.
  estate $\in$ {SLEEP, INIT, FLOOD} : State for escape attempts.
  heads, slaves : Sets of Id:s tracking what nodes have which role.
  smem, sendset, data : Infotuple sets for keeping and forwarding state data.

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.
  pruneset(A): $maxt \leftarrow \{< j,ji,ttl,ttf> \in A : ttl = \max_\tau \{\tau : < j,ji,\tau,ttf> \in A\}\}$
          return $\{< j,ji,ttl,ttf> \in maxt : ttf = \max_\phi \{\phi : < j,ji,ttl,\phi> \in maxt\}\}$
  prunemem(A): return $\{< j,ji,\xi> \in A : \xi = \max_x \{x : < j,ji,x> \in A\})\}$
```

**Figure 4.1:** *Constants, variables, external functions and macros for the algorithm in Figures 4.2 and 4.3.*

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 of the algorithm is for cluster heads to constantly broadcast the fact that they are cluster heads and for all nodes to constantly broadcast which nodes they consider to be cluster heads. The set of considered cluster heads consists both of nodes that are known to be cluster heads and, additionally, nodes that are elected to become cluster heads. The content of the broadcasts are forwarded $r$ hops, but in an aggregated form to keep the size of messages down. The election process might establish too many cluster heads. Therefore, there is a mechanism for cluster heads to drop their cluster head roles, to *escape*. Eventually a local minimum of cluster heads forms a total (k,r)-dominating set (or, if not possible given the topology, it fulfills $|C_j^r| \geq k_j$ for any node $p_j$). The choice of which nodes that are picked when electing cluster heads is based on node ID:s in order to limit the number of unneeded cluster heads that are elected when new cluster heads are needed.

```
on pulse:
  $timer \leftarrow (timer + 1) \mod T$
  if estate $=$ SLEEP $\wedge \exists t$ s.t. $(i, \text{JOIN}, t) \in smem$ then state $\leftarrow$ HEAD
  if state $=$ HEAD then $(newheads, newslaves) \leftarrow (\{i\}, \emptyset)$
  else $(newheads, newslaves) \leftarrow (\emptyset, \{i\})$
  for each $j \in \{k \mid k \neq i \wedge \exists\, ki \neq$ JOIN,$t$ s.t. $(k,ki,t) \in smem\}$ do handlestate($j$)
  $(heads, slaves) \leftarrow (newheads, newslaves)$

  /* Escaping */
  if $state \in \{$HEAD, ESCAPING$\}$
    estate $\leftarrow updateestate()$
    if $estate =$ INIT $\wedge\ state =$ HEAD $\wedge\ |heads| > k$
      state $\leftarrow$ ESCAPING
      $<$ heads, slaves$>$ $\leftarrow$ $<$ heads $\setminus \{i\}$, slaves $\cup \{i\}>$
  if state $=$ ESCAPING $\wedge$ estate $=$ SLEEP
    if $\exists t$ s.t. $(i, \text{JOIN}, t) \in smem$ then state $\leftarrow$ HEAD
    else state $\leftarrow$ SLAVE
  if state $=$ SLAVE then $<$ estate,estart$>$ $\leftarrow$ $<$ SLEEP,-1$>$

  /* Add heads */
  if $|heads| < k$
    heads $\leftarrow$ heads $\cup$ $\{$smallest($k$ -$|heads|$, slaves)$\}$
    slaves $\leftarrow$ slaves $\setminus$ heads

  /* Join and send state */
  for each $j \in$ heads
    if $j \neq i$ then sendset $\leftarrow$ pruneset(sendset $\cup$ $\{<j$, JOIN, $r$, $\pi> \}$)
    else state $\leftarrow$ HEAD
  smem $\leftarrow$ stepmem(smem)
  $<$ sendset,data$>$ $\leftarrow$ stepset(pruneset(sendset $\cup$ $\{< i$, state, $r$, $\pi> \}$))
  LBcast($<$ i,data$>$ )

function updateestate:
  if timer $= 0$ then estart $\leftarrow$ uniformlyrandom($\{0, 1, \ldots, T$-$T_{cool}$-1$\}$)
  if estart $\in [0,T$-$T_{cool}$-1$]$
    if timer $\in$ [estart, estart] then return INIT
    else if timer $\in$ [estart+1, estart+$T_{flood}$-1] then return FLOOD
  return SLEEP
```

**Figure 4.2:** *Pseudocode for the self-stabilizing clustering algorithm (1/2).*

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 system state. Recovery from an inconsistent system 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.

```
function $handlestate(j)$:
  $js \leftarrow prioritystate(j, smem)$
  if $js = \text{HEAD}$
    $newheads \leftarrow newheads \cup \{j\}$
    $sendset \leftarrow \text{pruneset}(sendset \cup \{< j, \text{JOIN}, r, \pi > \})$
  else if $js = \text{ESCAPING} \wedge j \in heads$
    if $|heads| \leq k$
      $newheads \leftarrow newheads \cup \{j\}$
      $sendset \leftarrow \text{pruneset}(sendset \cup \{< j, \text{VETO}, r, \pi > \})$
    else $heads \leftarrow heads \setminus \{j\}$
  $newslaves \leftarrow (newslaves \cup \{j\}) \setminus newheads$

function $prioritystate(j, mem)$:
  if $\exists\, t$ s.t. $(j, \text{HEAD}, t) \in mem$
    return HEAD
  if $\exists\, t$ s.t. $(j, \text{ESCAPING}, t) \in mem$
    return ESCAPING
  return SLAVE

function $stepmem(mem)$:
  $newmem \leftarrow \emptyset$
  for each $< j, js, ttk >$ in mem
    $ttk \leftarrow \mathbf{min}(ttk, \kappa)$-1
    if $ttk > 0$
      $newmem \leftarrow \text{prunemem}($
          $newmem \cup \{< j, js, ttk > \})$
  return $newmem$

function $stepset(set)$:
  $< newset, newdata > \leftarrow < \emptyset, \emptyset >$
  for each $< j, ji, ttl, ttf >$ in set
    $< ttl, ttf > \leftarrow < \mathbf{min}(ttl, r), \mathbf{min}(ttf, \pi)$-1$>$
    if $ttf > 0 \wedge ttl > 0$ then
      $newset \leftarrow \text{pruneset}(newset \cup \{< j, ji, ttl, ttf > \})$
    if $ttf \geq 0 \wedge ttl > 0$ then
      $newdata \leftarrow newdata \cup \{< j, ji, ttl > \}$
  return $< newset, newdata >$

on $\text{LBrecv}(< k, infoset > )$:
  for each $< j, ji, ttl > \in infoset$
    $ttl \leftarrow \mathbf{min}(ttl, r))$
    if $ji = \text{VETO}$
      if $j = i \wedge state = \text{ESCAPING}$
        $state \leftarrow \text{HEAD}$
    else if $(j \neq i \wedge ji \neq \text{JOIN}) \vee (j = i \wedge ji = \text{JOIN})$
      $smem \leftarrow \text{prunemem}(smem \cup \{< j, ji, \kappa > \})$
    if $j \neq i \wedge ttl > 1$
      $sendset \leftarrow \text{pruneset}(sendset \cup \{< j, ji, ttl\text{-}1, \pi > \})$
```

**Figure 4.3:** *Pseudocode for the self-stabilizing clustering algorithm (2/2).*

At each pulse a node sends out its state (the algorithmic state, i.e., which role it takes in the algorithm) and forwards the 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 pulse 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 elected nodes by sending 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 $T$ pulses. A cluster head node $p_i$ picks uniformly at random one pulse out of the $T - T_{cool}$ first pulses in the period as a possible starting pulse, $estart_i$, for an escape attempt. If $p_i$ has more than $k$ cluster heads in pulse $estart_i$, then it will start an escape attempt. When starting an escape attempt a node sets it state to ESCAPING and keeps it that way for a number of pulses 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 continuing to regard $p_i$ to be a cluster head and send a VETO back to $p_i$. 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 pulses all nodes in $\hat{G}_i^r$ 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 Figures 4.2 and 4.3 with accompanying constants, variables, external functions and macros in Figure 4.1. At each pulse of a node the lines 1-31 are executed resulting in a message that is broadcast at some time before the next pulse of that node. When a message is being received, the lines 78-87 are executed.

## 4.4 Correctness

In Section 4.4.1 we show some basic results that we use further on. In Section 4.4.2 we will show that within $O(r\pi\lambda^3)$ time 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 4.1. In Section 4.4.3 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 in $O(r\pi\lambda^4 g \log n)$ time in Theorem 4.2.

**Definition 4.1** *When all system assumptions hold from a point $s$ in time and forward, we say that "we have a legal system execution from $s$". We denote a pulse of $p_i$ with $\Gamma_x^i$ for some integer $x$. Consecutive pulses of $p_i$ have consecutive indices, e.g., $\Gamma_x^i$, $\Gamma_{x+1}^i$, $\Gamma_{x+2}^i$, etc. We denote the time between $\Gamma_x^i$ and $\Gamma_{x+1}^i$ with $\gamma_x^i$.*

**Definition 4.2** *We define the set of* states *as $\{SLAVE, HEAD, ESCAPING\}$. An* infotuple *is a tuple $(j, js, ttx)$ or $(j, js, ttl, ttf)$, where $js$ is a either a state or one of $\{VETO, JOIN\}$ and is said to be* for node *$p_j$ regardless if $p_j$ is the original sender or final receiver of the infotuple. The $ttx$ field can either be a $ttl$, the number of hops the info is to be forwarded, or a $ttk$, the number of pulses for which the infotuple should be kept in $smem$ before being discarded. A $ttf$ field denotes the number of resends that is left to be done for that particular tuple.*

*We say that a state earlier in the list [HEAD, ESCAPING, SLAVE] has priority over a state that is later in that list.*

*We say that an infotuple $(j, \sigma, \tau)$ is $memorable_i$ if and only if either $j \neq i$ and $\sigma$ is a state, or if $j = i$ and $\sigma =$ JOIN and that it is $relevant_i$ if and only if either it is $memorable_i$ or if $i = j$ and $\sigma =$ VETO.*

**Definition 4.3** *A node $p_i$ is said to* handle *a state $\sigma$ for a node $p_j$ in a pulse $\Gamma_x^i$ when the $handlestate$ function is called with parameter $j$ at line 6 and the* *subsequent call to the $prioritystate$ with $j$ as a parameter returns $\sigma$, setting $js_i = \sigma$ at line 41.*

### 4.4.1 Basic properties

This section builds up a base on how the algorithm works together with the system settings. First up is the definition of various constants whose value is the result of later lemmas.

**Definition 4.4** *We define $\kappa = \lceil(2r\pi+1)\lambda\rceil$, $T_{flood} = \lceil r(4\pi+2)\lambda^2 + r(2\pi+2)\lambda\rceil$, $t_s = r(2\pi+1)\lambda^2 + r(\pi+1)\lambda + \lambda - 1$, $t_e = (T_{flood}-1)\lambda + r(\pi+1)\lambda + \kappa\lambda$, $t_h = \kappa - r(\pi-1)\lambda - 1$, and $T_{cool} = \lceil t_e + r(\pi+1)\lambda\rceil$. Furthermore, we define $T = T_{es} + T_{cool}$, where $T_{es} = \lceil \frac{2g}{\ln 2}(t_s + t_e - 2t_h + 1)\rceil$.*

**Lemma 4.1** *Assume that we have a legal system execution from time $s - (\pi-1)\lambda$ and consider a node $p_i$ that has a pulse $\Gamma^i_x$ at time $s$. Now, assume that $p_i$ has $(k, \sigma, \tau, \pi)$, with $\tau > 0$, in $sendset_i$ just before executing line 30 in $\Gamma^i_x$ and consider a node $p_j \in N_i$ and a time interval $I = [s - (\pi-1)\lambda, s + (\pi+1)\lambda]$.*

*First, there exist a pulse $\Gamma^j_y \in I$ so that $(k, \sigma, \tau')$ is received in $\gamma^j_{y-1}$, for a $\tau' \geq \tau$.*

*Second, if $(k, \sigma, \tau')$ is $memorable_j$, then $(k, \sigma, \kappa) \in smem_j$ in $\Gamma^j_y$ just before executing line 2.*

*Third, if $k \neq j$, regardless of what $\sigma$ is, and if $\tau > 1$, then there exist a pulse $\Gamma^j_{\hat{y}} \in I$ (possibly equal to $\Gamma^j_y$) in which $(k, \sigma, \hat{\theta}, \pi) \in sendset_j$ with an $\hat{\theta} \geq \tau - 1$, just before executing line 30.*

**Proof:** Consider a pulse $\Gamma^i_{\hat{x}}$ in which $(k, \sigma, \tau, \varphi) \in sendset_i$, with $0 < \tau$ and $0 < \varphi < \pi$, just before executing line 30. At line 30 $stepset$ is called and lines 68–76, which results in $(k, \sigma, \tau) \in data_i$, which is broadcast by $p_i$ in $\gamma^i_{\hat{x}}$ due to line 31. Furthermore, if $\varphi > 1$, the infotuple $\eta = (k, \sigma, \tau, \varphi - 1) \in sendset_i$ just after executing line 30. The only mechanism that can remove $\eta$ from $sendset_i$ between line 30 in $\Gamma^i_{\hat{x}}$ and line 30 in $\Gamma^i_{\hat{x}+1}$ are when sendset are updated after using $pruneset$ on the union of sendset and an infotuple $\eta' = (k, \sigma, \theta, \pi)$ with $\theta \geq \tau$ in which case $\eta$ is replaced by $\eta'$. Note that $\pi$ is the only possible value used in the fourth field of $\eta'$. Thus, if $\varphi > 1$, then $(k, \sigma, \tau', \varphi') \in sendset_i$, where $\tau' \geq \tau$ and $\varphi' \geq \varphi - 1$, just before executing line 30 in $\Gamma^i_{\hat{x}+1}$.

Applying this argument to $\Gamma^i_x$, we get that $p_i$ broadcasts $(k, \sigma, \tau'_x)$ after $\Gamma^i_x$ and have $(k, \sigma, \tau'_{x+1}, \varphi'_{x+1}) \in sendset_i$, where $\tau'_{x+1} \geq \tau'_x$ and $\varphi'_{x+1} \geq \varphi'_x - 1$. Iterating the argument we get that $p_i$ broadcasts $(k, \sigma, \tau'_{x+1})$ after $\Gamma^i_{x+1}$ and have $(k, \sigma, \tau'_{x+2}, \varphi'_{x+2}) \in sendset_i$, where $\tau'_{x+2} \geq \tau'_x$ and $\varphi'_{x+2} \geq \varphi'_x - 2$, etc. Therefore we know that for all $a \in [0, \pi - 1]$ it holds that $p_i$ broadcasts $(k, \sigma, \tau'_{x+a})$ in $\gamma^i_{x+a}$ with $\tau'_{x+a} \geq \tau$. By the system settings, out of those $\pi$ consecutive broadcasts, at least one is successful. As the maximum time between two pulses of $p_i$ is $\lambda$ we get that at time $s + \pi\lambda$ all nodes in $N_i$ have successfully received one of those tuples from $p_i$. Any node $p_j$ have at least one pulse in any time interval of length $\lambda$. Thus for any node $p_j \in N_i$, there exist a pulse $\Gamma^j_y$ in the time interval $[s, s + (\pi + 1)\lambda]$ such that $p_j$ receive a tuple $(k, \sigma, \tau')$ in $\gamma^j_{y-1}$, for a $\tau' \geq \tau$. Thus, the first lemma claim is validated.

Now consider what can happen when $p_j$ receives this $(k, \sigma, \tau')$ and executes the $LBrecv$ function starting at line 78. If $(k, \sigma, \tau')$ is $memorable_j$, the condition at line 81 will not hold and line 85 will be executed, adding $\mu = (k, \sigma, \kappa)$ to $smem_j$. Nothing can remove $\mu$ from $smem_j$ before line 1 as $\kappa$ is the maximum possible value in that field, and only calls to $prunemem$ can do changes to $smem_j$ before the execution of line 1 in $\Gamma^j_y$. Thus, the second claim of the lemma holds.

Regardless of what $\sigma$ is, if $\tau > 1$ we know that $\tau' > 1$ and if $k \neq j$, then $p_j$ will attempt to add $\hat{\eta} = (k, \sigma, \tau' - 1, \pi)$ to $sendset_j$ when executing line 87. Here we need to consider cases of an existing infotuple $\hat{\eta}' = (k, \sigma, \theta, \phi)$ already in $sendset_j$, or added to sendset before the execution of line 1 in $\Gamma^j_y$. If $\tau' - 1 \geq \theta$ or $\phi = \pi$, the remaining infotuple in $sendset_j$ is the one out of $\eta$ and $\eta'$ with the largest third field (which is $\geq \tau' - 1$) and fulfills the third claim of the lemma $\Gamma^j_{\hat{y}} = \Gamma^j_y$.

Now consider the case in which $\tau' - 1 < \theta$ and $\phi < \pi$, in which $\hat{\eta}$, that does not fulfill the third claim of the lemma, is kept. Additions (i.e., excluding changes made by $stepset$) to $sendset_j$ can only have $\pi$ in the fourth field. Given the execution is legal since time $s - (\pi - 1)\lambda$, the only way to have

$(k, \sigma, \theta, \phi) \in sendset_j$ in $\gamma^j_{y-1}$ is if $(k, \sigma, \theta, \phi + 1) \in sendset_j$ just before line 30 in $\Gamma^j_{y-1}$ is for. By iterating this argument we eventually reach pulse $\Gamma^j_{\hat{y}}$, where $\hat{y} = y - \pi + \phi$, in which $(k, \sigma, \theta, \pi) \in sendset_j$ just before executing line 30. We know that $\phi > 0$, because $\hat{\eta}$ could not have remained with a 0 in the fourth field after the call to $stepset$ in $\Gamma^j_{y-1}$. Therefore, with $\Gamma^j_y$ in the time interval $[s, s+(\pi+1)\lambda]$ and a maximum time of $\lambda$ between consecutive pulses, the time of $\Gamma^j_{\hat{y}}$ is in the interval $[s - (\pi - 1)\lambda, s + (\pi + 1)\lambda]$. Thus, the third claim of the lemma holds for this final case as well. ■

This lemma shows that the algorithm forwards information from any node $p_j$ such that it reaches all nodes in $\hat{G}^r_j$ within time $O(r\pi\lambda)$. The three factors are due to forwarding $r$ hops, only one in $\pi$ messages are guaranteed to arrive and the clock skew can allows for pulses to be up to $\lambda$ time apart.

**Lemma 4.2** *Assume that we have a legal system execution from time $s - r(\pi - 1)\lambda$ and consider a node $p_i$ that has a pulse $\Gamma^i_x$ at time $s$. Now, assume that $p_i$ has $(k, \sigma, r, \pi)$ in $sendset_i$ just before executing line 30 in $\Gamma^i_x$ and consider a node $p_j \in G^r_i$, $p_j \neq p_i$ and a time interval $\hat{I} = [s - r(\pi - 1)\lambda, s + r(\pi + 1)\lambda]$.*

*First, if $(k, \sigma, \tau')$ is $relevant_j$, there exist a pulse $\Gamma^j_y \in \hat{I}$ so that $(k, \sigma, \tau')$ is received in $\gamma^j_{y-1}$, for a $\tau' \geq 1$.*

*Second, if $(k, \sigma, \tau')$ is $memorable_j$, then $(k, \sigma, \kappa) \in smem_j$ in $\Gamma^j_y$ just before executing line 2.*

**Proof:** We say that a node $p_\ell$ fulfills $\Sigma_h$ if 1) there exist a pulse $\Gamma^\ell_{z_\ell}$ in the time interval $I_h = [s - h(\pi - 1)\lambda, s + h(\pi + 1)\lambda]$ such that $p_\ell$ receives an infotuple $(k, \sigma, \tau'_\ell)$ in $\gamma^\ell_{z_\ell - 1}$, with $\tau'_\ell \geq r - h + 1$, and 2) if $(k, \sigma, \tau'_\ell)$ is $memorable_\ell$, then $(k, \sigma, \kappa) \in smem_\ell$ just before executing line 2 in $\Gamma^\ell_{z_\ell}$, and 3) if $h < r$ and $k \neq \ell$, there exists a pulse $\Gamma^\ell_{w_\ell} \in I_h$ in which $(k, \sigma, \theta_\ell, \pi) \in sendset_j$ with an $\theta_\ell \geq r - h$, just before executing line 30.

As $(k, \sigma, r, \pi) \in sendset_i$ just before executing line 30 in $\Gamma^i_x$ with $r > 0$, by Lemma 4.1 any node $p_j \in N_i$ fulfills $\Sigma_1$.

Consider a node $p_\ell$ that is $1 < h < r$ hops away from $p_i$ and that fulfills $\Sigma_h$. As $p_\ell$ fulfills $\Sigma_h$ it has $(k, \sigma, \theta_\ell, \pi) \in sendset_j$, with a $\theta_\ell \geq r - h$, just before executing line 30 in a pulse $\Gamma^\ell_{w_\ell}$ at a time $\hat{s} \in [s - h(\pi - 1)\lambda, s + h(\pi + 1)\lambda]$.

Applying Lemma 4.1, a node $p_m \in N_\ell$ has a pulse $\Gamma^m_{z_m} \in I' = [\hat{s} - (\pi - 1)\lambda, \hat{s} + (\pi + 1)\lambda]$ such that $p_m$ receives an infotuple $(k, \sigma, \tau'_m)$ in $\gamma^m_{z_m - 1}$, with $\tau'_m \geq \theta_\ell \geq r - h = r - (h+1) + 1$. Furthermore, if $(k, \sigma, \tau'_m)$ is $memorable_m$, then $(k, \sigma, \kappa) \in smem_m$ just before executing line 2 in $\Gamma^m_{z_m}$. Moreover, if $h + 1 < r$, then $r - h > 1 \Rightarrow \theta_\ell > 1$ and by Lemma 4.1 there exists a pulse $\Gamma^m_{w_m} \in I'$ in which $(k, \sigma, \theta_m, \pi) \in sendset_j$ with an $\theta_m \geq \theta_\ell - 1 \geq r - (h+1)$, just before executing line 30 if $k \neq j$. Given that $\hat{s} \in [s - h(\pi - 1)\lambda, s + h(\pi + 1)\lambda]$ we get that $I' \subset [s - (h + 1)(\pi - 1)\lambda, s + (h + 1)(\pi + 1)\lambda]$. Therefore, if $k \neq \ell$, any node $p_m \in N_\ell$, including the subset of $N_\ell$ that are $h + 1$ hops away from $p_i$, fulfills $\Sigma_{h+1}$. A tuple that is forwarded through the network can therefore be withheld from forwarding either when $k = i$ and $\sigma$ is a state and reaches back to $p_i$, or when $k = j$ and $\sigma$ is JOIN or VETO, and reaches $p_j$ (in which case $p_j$ is the only node for which it is $relevant$ anyway). Therefore, by the induction principle, $\Sigma_r$ therefore holds for any node $p_j \in G^r_i$ that is not $p_i$ and that do not have node $p_k \neq p_i$ on all possible paths of length $\leq r$ to $p_i$, which proves the Lemma. ■

**Lemma 4.3** *Assume that we have a legal system execution from time $s - r(\pi - 1)\lambda$ and consider a node $p_i$ that has a pulse $\Gamma^i_x$ at time $s$. If $p_j \in G^r_i$, then $i \in slave_j \cup head_j$ from time $s + r(\pi + 1)\lambda$ and forward.*

**Proof:** Consider a node $p_j \in G^r_i$. We say that $u$ holds when $i \in slave_j \cup head_j$ and say that $\Sigma_a$ holds for a pulse $\Gamma^j_z$ when $(i, \sigma, \xi) \in smem_j$ just before executing line 7 in $\Gamma^j_z$, for a $\xi > a \geq 0$.

In pulse $\Gamma^i_x$, $p_i$ adds $(i, state_i, r, \pi)$ to $sendset_i$ at line 30. We denote this value of $state_i$ as $\sigma$ and note that $(i, \sigma, \cdot)$ is $relevant_j$ to all nodes $p_j \neq p_i$. By Lemma 4.2, for any node $p_j \in G^r_i$ there will exist a pulse $\Gamma^j_y$ in the time span $[s - r(\pi - 1)\lambda, s + r(\pi + 1)\lambda]$ in which $smem_j$ contains $(i, \sigma, \kappa)$ before executing line 2. In other words, $\Sigma_{\kappa - 1}$ holds for $\Gamma^j_y$.

Consider a pulse $\Gamma^j_z$ for which $\Sigma_0$ holds. No lines between lines 1–7 can do any changes to $smem_j$. Therefore, the $handlestate$ function starting at line 40 will be called in $\Gamma^j_z$ with $i$ as an argument. In the $handlestate$ function $i$ might be added to $newheads_j$ at lines 44 or 48. Line 50 adds $i$ to $newslaves_j$ if

$i \notin newheads_j$. Therefore, $u$ holds after executing line 7 in $\Gamma^j_z$. The only thing that can switch $u$ from holding to not holding is when at line 7 when lines 6–7 executes without calling $handlestate$ for $p_i$. This can only happen in a pulse if $\Sigma_0$ does not hold for that pulse. Furthermore, if $u$ holds at line 1 in $\Gamma^j_z$ (for which $\Sigma_0$ holds) $u$ will never cease to hold when executing lines 1–7 as the only change in those lines are the assignment in 7 that brings $p_j$ from a state where $u$ holds to another state where $u$ holds. Also, as no other lines can make $u$ cease to hold, $u$ holds from just after executing line 7 in $\Gamma^j_z$ up to, at least, just before executing line 7 in $\Gamma^j_{z+1}$ as well.

Now, consider a pulse $\Gamma^j_w$ for which $\Sigma_a$ holds for an $a > 0$, i.e., $(i, \sigma, \xi) \in smem_j$ just before executing line 7 for a $\xi > a > 0 \Rightarrow \xi > 1$. During the execution of lines 7–31 in $\Gamma^j_w$, the only line that can remove $(i, \sigma, \xi)$ from $smem_j$ is line 29. For $\xi > 1$, this replaces $(i, \sigma, \xi)$ with $(i, \sigma, \xi - 1)$. The only mechanism that can remove $(i, \sigma, \xi - 1)$ from $smem_j$ between line 31 of $\Gamma^j_w$ and the execution of line 1 in $\Gamma^j_{w+1}$ is the execution of line 85. The infotuple $\mu = (k, \sigma, \xi - 1)$ can only be removed if another infotuple $\mu' = (k, \sigma, \xi')$ is added where $\xi - 1 < \xi' \leq \kappa$. Therefore, $(k, \sigma, \xi') \in smem_j$, for a $\xi' \geq \xi - 1 > a - 1 \geq 0$ at line 1 in $\Gamma^j_{w+1}$. In other words, $\Sigma_{a-1}$ holds for $\Gamma^j_{w+1}$.

We know, from the beginning of the lemma proof, that $\Sigma_{\kappa-1}$ holds for $\Gamma^j_y$. By the induction principle, for $a \in [0, \kappa - 1]$, we get that $\Sigma_{\kappa-a-1}$ holds (and that $\Sigma_0$ holds as $a \geq 0$) for $\Gamma^j_{y+a}$. Therefore $u$ holds from just after line 7 in $\Gamma^j_y$ until just before line 7 in $\Gamma^j_{y+\kappa}$. The pulse $\Gamma^j_y$ of $p_j$ can not occur earlier than time $s - r(\pi - 1)\lambda$. Therefore, by the system settings, $\Gamma^j_{y+\kappa}$ can not occur earlier than time $s - r(\pi - 1)\lambda + \kappa$. The latest possible time of $\Gamma^j_y$ is $s + r(\pi + 1)\lambda$. Thus, $u$ holds in the interval $[s + r(\pi + 1)\lambda, s - r(\pi - 1)\lambda + \kappa]$.

By the system settings $\Gamma^i_{x+1}$ of $p_i$ occurs in the time interval $[s + 1, s + \lambda]$, in which $p_i$ once again adds its state to $sendset_i$ at line 30. By the proof above this means that $u$ holds in the interval $[s + r(\pi + 1)\lambda + \lambda, s - r(\pi - 1)\lambda + \kappa + 1]$ as well. As $(s - r(\pi - 1)\lambda + \kappa) - (s + r(\pi + 1)\lambda + \lambda) = \kappa - 2r\pi\lambda - \lambda \geq 0$, there will be overlap between the intervals guaranteed by $\Gamma^i_x$ and $\Gamma^i_{x+1}$. In the border case in which the guarantees from $\Gamma^i_x$ ends in the same pulse $\Gamma^j_{y+\kappa}$ as the guarantees from $\Gamma^i_{x+1}$ begin, $u$ continues to hold as $u$ holds from the beginning of $\Gamma^j_{y+\kappa}$ due to $\Gamma^i_x$ and $\Sigma_0$ holds for $\Gamma^j_{y+\kappa}$ due to $\Gamma^i_{x+1}$. Iterating the logic starting with $\Gamma^i_{x+1}$ and then with $\Gamma^i_{x+2}$, etc., we get that $u$ holds from time $s + r(\pi + 1)\lambda$ and forward as long as the execution is legal. ■

**Lemma 4.4** *Assume that we have a legal system execution from time $s$ and consider a node $p_i$ that has a pulse $\Gamma^i_x$ at time $s$ and that has $(i, \sigma, r, \pi)$, where $\sigma$ is a state, in $sendset_i$ just before executing line 30 in $\Gamma^i_x$. Assume that $p_i$ do not add $(i, \sigma, r, \pi)$ to $sendset_i$ in the time interval $I = (s, s + a)$, for an $a > s + r(\pi + 1)\lambda + (\kappa - 1)\lambda$. In other words, $p_i$ does not execute line 30 with $state_i = \sigma$ in any pulses or between any pulses in the time interval $I$. Consider any node $p_j$ in the network. Under these assumptions, the last pulse $\Gamma^j_y$ in the time interval $I$ for $p_j \in \hat{G}^r_i$ such that $p_j$ receives $(i, \sigma, \tau)$ for any $\tau$ in $\gamma^i_{y-1}$, can not be later than $s + r(\pi + 1)\lambda$. Furthermore, for any pulse of $p_j$ in the time interval $(s + r(\pi + 1)\lambda + (\kappa - 1)\lambda, s + a)$, and between those pulses, $(i, \sigma, \xi) \notin smem_j$ for any $\xi$.*

**Proof:** We start by assuming that $p_i$ will *never* add $(i, \sigma, r, \pi)$ to $sendset_i$ after $\Gamma^i_x$. Apart from line 30, line 30 is the only line that could add a tuple $(i, \sigma, \tau, \phi)$ for any $\tau$ or $\phi$ and by the Lemma assumption that can not happen. Line 30 in $\Gamma^i_x$ will change $(i, \sigma, r, \pi)$ to $(i, \sigma, r, \pi - 1)$, $\Gamma^i_{x+1}$ will change $(i, \sigma, r, \pi - 1)$ to $(i, \sigma, r, \pi - 2)$, etc. Thus the last pulse for which $p_i$ sends $(i, \sigma, r)$ is $\Gamma^i_{x+\pi-1}$. With a maximum time of $\lambda$ between two consecutive pulses of a node, the last possible pulse $\Gamma^k_w$ for a node $p_k \in N_i$ for which $p_k$ receives $(i, \sigma, r)$ from $p_i$ in $\gamma^k_{w-1}$ can not happen later than $s + (\pi + 1)\lambda$. There can be no case in which $p_k$ receives $(i, \sigma, r)$ in $\gamma^k_{w-1}$ from another node in $N_k$ as no other node than $p_i$ can add $(i, \sigma, r, \pi)$ to their sendset during the legal execution.

When $p_k$ receives $(i, \sigma, r)$, it will put $(i, \sigma, r - 1, \pi)$ in $sendset_k$ and iterating the argument above the last possible pulse $\Gamma^\ell_v$ for a node $p_\ell \in N_k$ to receives $(i, \sigma, r - 1)$ from $p_k$ can not happen later than $s + 2(\pi + 1)\lambda$. There can be no case in which $p_\ell$ receives $(i, \sigma, r - 1)$ in $\gamma^k_{w+c}$ for any $c > 0$ from another node in $N_\ell$ as $(i, \sigma, r - 1, \pi)$ expires just as for $p_k$. Iterating further, in a similar manner as for the proof of Lemma 4.2, we get that the last pulse $\Gamma^j_y$ for $p_j \in G^r_i, i \neq j$ such that $p_j$ receives $(i, \sigma, \tau)$ for any $\tau$ in $\gamma^i_{y-1}$, can not be later than $s + r(\pi + 1)\lambda$. It follows that $\Gamma^j_y$ is the last pulse for which $(i, \sigma, \kappa) \in smem_j$ just before executing line 29. Executing line 29 in $\Gamma^j_y$ reduces the infotuple to $(i, \sigma, \kappa - 1) \in smem_j$, line 29 in $\Gamma^j_y$ reduces it to $(i, \sigma, \kappa - 2)$, etc., until finally in $\Gamma^j_{y+\kappa-1}$ line 29 $(i, \sigma, 1)$ is removed from $smem_j$. Therefore, with $z = y + \kappa - 1$, we get that $(i, \sigma, \xi) \notin smem_j$ (for any $\xi$) in $\Gamma^j_{z+b}$ or $\gamma^j_{z+b-1}$ for all $b \geq 1$. The pulse $\Gamma^j_z$ can not happen later than $s + r(\pi + 1)\lambda + (\kappa - 1)\lambda$.

We now have to deal with the cases in which $p_i$ *do* add $(i, \sigma, r, \pi)$ to $sendset_i$ in a $\Gamma^i_{\hat{x}}$ at a time $\hat{s} \geq s + a$. By the arguments above, no node $p_j$ in the network have any $(i, \sigma, \theta, \phi) \in sendset_j$ for any $\theta$ or $\phi$ at time $s + r(\pi + 1)\lambda$. Therefore, even if $p_i$ adds $(i, \sigma, r, \pi)$ to $sendset_i$ at time $\hat{s}$, no node $p_j \in G^r_i$ can have a pulse $\Gamma^j_{\hat{y}}$ in the time interval $(s + r(\pi + 1)\lambda + (\kappa - 1)\lambda, s + a)$ in which $(i, \sigma, \xi) \in smem_j$. ■

**Lemma 4.5** *Let $\hat{s} = s - r(2\pi + 1)\lambda^2$ and assume that we have a legal system execution from time $\hat{s}$, and consider a node $p_i$ that has a pulse $\Gamma^i_x$ at time $s$ and that has $(k, \sigma, r, \pi)$ in $sendset_i$ just before executing line 30 in $\Gamma^i_x$. Assume that $(k, \sigma, \cdot)$ is $memorable_j$. Then, there exists a pulse $\Gamma^j_y \in I = [s + r(\pi + 1)\lambda - 1, s + r(\pi + 1)\lambda + \lambda - 1]$ in which $\exists \xi > 0$ such that $(k, \sigma, \xi) \in smem_j$.*

*Second, now consider only the case when $k = i$ and $\sigma$ is a state and consider a state $\hat{\sigma} \neq \sigma$. Assuming that $p_i$ do not have $(i, \hat{\sigma}, r, \pi) \in sendset_i$ just before executing line 2 in any pulse in the time interval $(s - r(2\pi + 1)\lambda^2, s + r(\pi + 1)\lambda + \lambda - 1)$. In this Lemma we denote this as $\Sigma_{\hat{\sigma}}$ holding for $\Gamma^i_x$. Then, $\nexists \xi'$ such that $(i, \hat{\sigma}, \xi') \in smem_i$ just before executing line 2 in $\Gamma^j_y$.*

*Third, if $\Sigma_{\sigma'}$ holds for all $\sigma'$ that have a higher priority than $\sigma$, then $p_j$ handles $\sigma$ for $p_i$ in $\Gamma^j_y$ (see Definition 4.2).*

**Proof:** By Lemma 4.2, there exists a pulse $\Gamma^j_z \in [s - r(\pi - 1)\lambda, s + r(\pi + 1)\lambda]$ in which $(k, \sigma, \kappa) \in smem_j$ just before executing line 2. Furthermore, by the arguments in the proof of Lemma 4.3, $(k, \sigma, \xi) \in smem_j$ for some $\xi \geq 1$ just before executing line 2 in $\Gamma^j_{z+\kappa-1}$. Moreover, $\Gamma^j_{z+\kappa-1}$ cannot happen before $s - r(\pi - 1)\lambda + \kappa - 1 = s - r(\pi - 1)\lambda + \lceil (2r\pi + 1)\lambda \rceil - 1 \geq s + r(\pi + 1)\lambda + \lambda - 1$.

Therefore, there exists a pulse $\Gamma^j_y \in I \equiv [s+r(\pi+1)\lambda-1, s+r(\pi+1)\lambda+\lambda-1]$ in which $(k, \sigma, \xi) \in smem_j$ just before executing line 2 for some $\xi \geq 1$.

We now consider only the case when $k = i$ and $\sigma$ is a state and consider a state $\hat{\sigma} \neq \sigma$. Without loss of generality, we assume that $p_i$ have $(i, \hat{\sigma}, r, \pi) \in sendset_i$ just before executing line 2 in a pulse $\Gamma^j_{\hat{x}}$ at time $\hat{s}$. By Lemma 4.4, for any node $p_j$ in the network, $p_j$ does not have $(i, \hat{\sigma}, \xi) \in smem_j$ for any $\xi$ in the time interval $(t_{low}, t_{high}) \equiv (\hat{s}+r(\pi+1)\lambda+(\kappa-1)\lambda, s+r(\pi+1)\lambda+\lambda-1)$. Furthermore, $t_{low} = s - r(2\pi+1)\lambda^2 + r(\pi+1)\lambda + (\kappa-1)\lambda \leq s + r(\pi+1)\lambda - \lambda < s + r(\pi+1)\lambda - 1$, which is the lower end of $I$. In the other end, $t_{high} = s + r(\pi+1)\lambda + \lambda - 1$, which is the higher end of $I$.

Now, if $\Sigma_{\sigma'}$ holds for all $\sigma'$ that have a higher priority than $\sigma$, then $\nexists \xi'$ such that $(i, \sigma', \xi') \in smem_i$ just before executing line 2 in $\Gamma^j_y$ for any such $\sigma'$. We have established that $(i, \sigma, \xi) \in smem_i$ just before executing line 2 in $\Gamma^j_y$ for some $\xi$. Therefore, the $handlestate$ function will be called with parameter $i$ at line 6 in $\Gamma^j_y$ which implies that the $prioritystate$ function will be called in line 41 with parameter $i$ and $smem_j$. The prioritystate function in lines 52-57 simply returns the state for the node in the argument with highest priority that exists in $smem$, which is $\sigma$ for $i$. Thus, $p_j$ handles $\sigma$ for $p_i$ in $\Gamma^j_y$. ■

The corollary shows that the mechanisms that keeps data for a certain time, guarantees that eventually nodes in $\hat{G}^r_j$ will see the correct state of node $p_j$ if it stays in that state long enough. Compared to Lemma 4.2 this introduces another factor $O(\lambda)$ time. This is because if a node wants to make sure that $O(r\pi\lambda)$ time has passed it needs to count $O(r\pi\lambda)$ pulses, but $O(r\pi\lambda)$ pulses can take $O(r\pi\lambda^2)$ time. Building Lemmas on top of each other, this is the mechanism that adds additional factors of $\lambda$ the further we go in the proof chain.

**Corollary 4.1** *Assume that we have a legal system execution from time $s - \lambda$ and consider a node $p_i$ that has a pulse $\Gamma^i_x$ at time $s$. Let $\chi = \lceil r(2\pi+1)\lambda^2 + r(\pi+1)\lambda + 2\lambda \rceil$. Now assume that a node $p_i$, in each of the pulses $\Gamma^i_x$–$\Gamma^i_{x+\chi-1}$, adds $(i, \sigma, \pi)$ to $sendset_i$ and does not add $(i, \sigma', \pi)$ to $sendset_i$ for any $\sigma \neq \sigma'$. Then, for any node $p_j \in G^r_i$, $p_j$ has a pulse $\Gamma^j_y$ at a time $\hat{s} = s + r(2\pi+1)\lambda^2 + r(\pi+1)\lambda - 1 + t$ for a $t \in [0, 2\lambda]$, in which $p_j$ handles*

*$\sigma$ for $p_i$. Furthermore, $\hat{s}$ happens between the execution of the pulses $\Gamma^i_x$ and $\Gamma^i_{x+\chi-1}$*

**Proof:** There will be a pulse $\Gamma^i_{\hat{x}}$ at a time $s_0 = s + r(2\pi+1)\lambda^2 + t_0$, for a $t_0 \in [0, \lambda]$ that fulfills the requirements of Lemma 4.5 of no $(i, \hat{\sigma}, r, \pi) \in sendset_i$ for a $\hat{\sigma} \neq \sigma$ in the time interval $(s_0 - r(2\pi+1)\lambda^2, s_0 + r(\pi+1)\lambda + \lambda - 1)$. Therefore, there exists a pulse $\Gamma^j_y$ at a time $s_1 = s_0 + r(\pi+1)\lambda - 1 + t_1$ for a $t_1 \in [0, \lambda]$ in which $p_j$ handles $\sigma$ for $p_i$. We can expand this to $s_1 = s + r(2\pi+1)\lambda^2 + r(\pi+1)\lambda - 1 + t_2$ for a $t_2 \in [0, 2\lambda]$. According to the system settings, $\Gamma^i_{x+\chi-1} = \Gamma^i_{x+\lceil t+r(\pi+1)\lambda+2\lambda \rceil -1} =$ can not happen before $s + t + r(\pi+1)\lambda + 2\lambda - 1$. ■

### 4.4.2 Getting Enough Cluster Heads

This section builds up a case showing that the algorithm will elect enough cluster heads. We show how cluster heads are elected in Lemmas 4.6, 4.7 and 4.8. In Lemma 4.9 we take a look at how the escape mechanism works. Finally, in Theorem 4.1, we put it all together and show that within $O(r\pi\lambda^3)$ time from starting a legal execution, each node $p_j$ in the network will get $k_j$ cluster heads within $r$ hops.

**Definition 4.5** *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 $C^r_j$ as the set of cluster heads in $G^r_j$. Furthermore, we define $H_x$ to be the set of cluster heads in the network at time $x$.*

We now look how the addition of cluster heads work while temporarily disregarding the escaping mechanism. In this setting we will show that within a finite time we will have $|C^r_i| \geq k_i$ for any node $p_i$. Later on we will lift this restriction and show that $|C^r_i| \geq k_i$ will still hold even when regarding the more general case.

**Lemma 4.6** *Assume that we have a legal system execution from time $s$. Assume that, for all nodes in the network, their $state$ can never be ESCAPING,* *their $estate$ is always SLEEP and lines 10-18 are never executed. Under these assumptions, any node $p_i$ will have $k_i$ cluster heads within $r$ hops from time $s + (3r\pi + 2)\lambda$ and forward.*

**Proof:** Consider any node $p_i$ and a node $p_j \in G_i^r$. By the system settings, $p_j$ has some pulse $\Gamma_y^j \in [s + r(\pi - 1)\lambda, s + r(\pi - 1)\lambda + \lambda]$. Together with Lemma 4.3 and the proof of that lemma, $j \in slaves_i \cup heads_i$ holds for a pulse in $[s + r(\pi - 1)\lambda + r(\pi + 1)\lambda, s + r(\pi - 1)\lambda + r(\pi + 1)\lambda + \lambda]$ and from that point forward.

Thus there exists a pulse $\Gamma_x^i$ before time $\hat{s} = s + r(\pi - 1)\lambda + r(\pi + 1)\lambda + \lambda$ in which $k \in slaves_i \cup heads_i$ holds for all $p_\ell \in G_i^r$. At line 21 in $\Gamma_x^i$ $heads_i$ 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 22-23 will be executed. Out of the nodes in $slaves_i$ , the smallest $\min(|slaves_i|, k - |heads_i|)$ nodes will be added to $heads_i$ in line 22 and removed from $slaves_i$ in line 23. Thus, after execution of line 23 $heads_i$ will contain $k_i = \min(|G_i^r|, k)$.

For each node $p_j \in heads_i$ either a infotuple $(j, \text{JOIN}, r, \pi)$ is added to $sendset_i$ (at line 27, when $j \neq i$) or the state is set to HEAD directly (at line 28, when $j = i$). By Lemma 4.2, for each of the $p_j \neq p_i$ in $heads_i$, there exists a pulse $\Gamma_{z_j}^j \in [\hat{s} - r(\pi - 1)\lambda, \hat{s} + r(\pi + 1)\lambda]$ such that $p_j$ receives $(j, \text{JOIN}, \tau_j)$ for a $\tau_j \geq 1$ in $\gamma_{z_j - 1}^j$. When $(j, \text{JOIN}, \tau_j)$ is received by $p_j$ the condition at line 81 does not hold, the condition at line 84 holds and $(j, \text{JOIN}, \kappa)$ is added to $smem_j$. Nothing in $\gamma_{z_j - 1}^j$ can remove $(j, \text{JOIN}, \kappa)$ from $smem_i$. Therefore the condition in line 3 holds in $\Gamma_{z_j}^j$ and $state_j$ is set to HEAD.

Under the lemma assumptions, the only ways that $state_j$ can change are the execution of lines 3, 28, or 83, all setting $state_j$ to HEAD. Thus, from time $\hat{s} + r(\pi + 1)\lambda = s + (3r\pi + 2)\lambda$ and forward, 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 receive joins remains or becomes a cluster head.

**Lemma 4.7** *Assume that we have a legal system execution from time $s - \lambda$ and consider a node $p_i$ that has a pulse $\Gamma_x^i$ at time $s$. If $p_i$ has $(i, \mathit{JOIN}, \xi) \in smem_i$,* *for some $\xi > 0$ before executing line 2, then $p_i$ is a cluster just before executing line 4 in $\Gamma^i_x$ and will stay cluster head throughout the rest of $\Gamma^i_x$ and throughout $\gamma^i_x$. Furthermore, if node $p_i$ is a cluster head just before executing line 2 in $\Gamma^i_x$, then it is a cluster head throughout the entire $\Gamma^i_x$.*

**Proof:** Let $\sigma$ be $state_i$ and $\epsilon$ be $estate_i$ just before executing line 2 in $\Gamma^i_x$. By the lemma assumptions the condition $\exists t$ such that $(i, \text{JOIN}, t) \in smem$ holds just before line 2 and as nothing among lines 1–17 can change $smem_j$ it holds at line 17 as well. If $\epsilon = \text{SLEEP}$, then $state_i$ is set to HEAD at line 3, making $p_i$ a cluster head. If $\epsilon \neq \text{SLEEP} \wedge \sigma \in \{\text{HEAD}, \text{ESCAPING}\}$, then no change in done to $state_i$ in lines 2–3.

The remaining case to consider is when $\epsilon \neq \text{SLEEP} \wedge \sigma = \text{SLAVE}$. Now, $estate_i$ can only be changed by lines 11 and 18 and $state_i$ can only be set to SLAVE in line 17. Therefore, neither $estate_i$, nor $state_i$ can be changed in $\gamma^i_{x-1}$ or in lines 21–31 of $\Gamma^i_{x-1}$. However, $state_i$ can only be SLAVE after line 21 during the legal execution of $\Gamma^i_{x-1}$ if $state_i = \text{SLAVE}$ at line 18, which would set $estate_i$ to SLEEP, contradicting the assumptions of $\epsilon \neq \text{SLEEP}$. Thus, in all possible cases $p_i$ is a cluster head just before line 4 in $\Gamma^i_x$.

The only line in lines 4–31 that can make $p_i$ becoming a non cluster head is line 17, which will not be executed as the condition holds at line 16. Furthermore, there is no way for $state_i$ to be set to SLAVE in lines 2–3. Therefore, if node $p_i$ is a cluster head just before executing line 2 in $\Gamma^i_x$, then it is a cluster head throughout the entire $\Gamma^i_x$. ■

In the following Lemma we show that a node that is continuously wanted as a cluster head eventually becomes one.

**Lemma 4.8** *Assume that we have a legal system execution from time $s$. Assume that a node $p_j \neq p_i$ wants a node $p_i \in G^r_j$ to be cluster head as soon as it knows about it and is never willing to let it escape. In other words, (1) if $i \in slaves_j$ and $i \notin heads_j$ after line 7 in a pulse $\Gamma^j_y$, we assume that the condition in line 21 holds in $\Gamma^j_y$ and that $i$ is added to $heads_j$ at the execution of line 22, and (2) the condition in line 46 would hold when $handlestate$ is called with $i$ as a parameter in any pulse of $p_j$.*

*Under these assumptions, there exists a pulse $\Gamma^i_x$ such that $state_i \neq$ SLAVE after executing line 3 in $\Gamma^i_j$, and such that $state_i \neq$ SLAVE in any $\Gamma^i_{x'+1}$ or $\gamma^i_{x'}$ for any $x' \geq x$.*

**Proof:** As seen in the proof of Lemma 4.6, there exists a pulse $\Gamma^j_y$ before time $\hat{s} = s + r(\pi - 1)\lambda + r(\pi + 1)\lambda + \lambda$ in which $i \in slaves_j \cup heads_j$ holds before executing line 2. By the lemma assumptions either $i \in heads_j$ before executing line 2 in $\Gamma^j_y$, or $i$ is added to $heads_j$ at line 22. Therefore $(i, JOIN, r, \pi) \in sendset_j$ just before executing line 30 in $\Gamma^j_y$ as it is added at line 27. Therefore, by Lemma 4.2, there exists a pulse $\Gamma^i_x$ such that $p_i$ receive $(j, JOIN, \tau)$ for some $\tau$ in $\gamma^i_{x-1}$.

By Lemma 4.3 $i \in slaves_j \cup heads_j$ holds before executing line 2 in any pulse $\Gamma^j_{y'}$ for $y' \geq y$. Therefore, $(i, JOIN, r, \pi) \in sendset_j$ just before executing line 30 in each such $\Gamma^j_{y'}$ and, with similar arguments as in the proof of Lemma 4.3, $p_i$ will have $(i, \text{JOIN}, \xi) \in smem_j$, for some $\xi$, just before executing line 3 in all pulses $\Gamma^i_{x'}$ for $x' \geq x$. Therefore, by Lemma 4.7 $state_i \neq$ SLAVE after the execution of line 3 and $state_i \neq$ SLAVE in a $\Gamma^i_{x'+1}$ or $\gamma^i_{x'}$ for any $x' \geq x$. ■

We continue to take a look at how the escape mechanism operates to know in what ways it could interfere with electing enough cluster heads.

**Definition 4.6** *A node $p_i$* initiates an escape attempt *in a pulse $\Gamma^i_x$ if the condition holds in line 12 and lines 13–14 are executed in $\Gamma^i_x$.*

Lemma 4.9 shows that the escape mechanism works, that a cluster head that is not needed can escape that responsibility.

**Lemma 4.9** *Assume that we have a legal system execution from time $s - T_{cool}$ and consider a node $p_i$ that initiates an escape attempt in a pulse $\Gamma^i_x$ at time $s$.*

*If all nodes $p_j \in G^r_i$ have $|C_j| > k$ at time $s + t_h$ and no node $p_\ell \in \hat{G}^{2r}_i$, initiates an escape attempt in any pulse in $[s - t_e + t_h, s + t_s - t_h]$ then node $p_i$ will set $state_i$ to SLAVE in pulse $\Gamma^i_{x+T_{flood}}$ and have $state_i =$ SLAVE throughout any $\gamma^i_{x'}$ or $\Gamma^i_{x'+1}$ for any $x' \geq x + T_{flood}$.*

*If, on the other hand, there exists a node $p_j \in G_i^r$ that is having $|heads_j| \leq k$ with $k \in heads_j$ when $p_j$ is first handling ESCAPING for $p_i$, then $p_j$ will not set $state_i$ to SLAVE in this escape attempt.*

**Proof:** The escape attempt initiated in $\Gamma_x^i$ can only happen in the following way. In $\Gamma_x^i$, $state_i = \text{HEAD}$ before executing line 2. Line 11 calls $updateestate$. The condition at line 35 holds and line 36 finds that $timer_i = estart_i$ and therefore returns INIT setting $estate_i = \text{INIT}$. The execution continues at line 12 where the condition holds and $state_i$ is set to ESCAPING and $i$ is removed from $heads_i$ and added to $slaves_i$. The branch at line 15 will not be entered in $\Gamma_x^i$ as $estate_i \neq \text{SLEEP}$. The branch at line 18 will not be entered in $\Gamma_x^i$ as $state_i = \text{ESCAPING}$. The branch at line 21 will not be entered in $\Gamma_x^i$ as $|heads_i|$ will be at least $k$ as it was strictly larger than $k$ and one entry was removed. The loop at line 26 will not be run for $j = i$ in $\Gamma_x^i$ as $i$ was removed from $heads_i$. Thus at the end of $\Gamma_x^i$, $state_i = \text{ESCAPING}$.

Consider a pulse $\Gamma_{\hat{x}}^i$, where $\hat{x} > x$, that is the first pulse after $\Gamma_x^i$ in which $state_i$ is set to something else that ESCAPING (or in the case of line 83 setting $state_i$ to $HEAD$, the pulse after that event, i.e., that change happens in $\gamma_{\hat{x}-1}^i$). The only two ways for $estate_i$ to change is either in line 11, that will set $estate_i$ to FLOOD in $\Gamma_{x+1}^i$–$\Gamma_{x+T_{flood}-1}^i$ unless $state_i$ has been set to SLAVE or line 18, that will only be executed if $state_i = \text{SLAVE}$. Thus, for any condition that requires $estate_i = \text{SLEEP}$, either $state_i$ will have to be set to SLAVE (which in will in turn set $estate_i$), or the condition can only hold in pulse $\Gamma_{x+T_{flood}}^i$ or later.

We will see that $|heads_i| < k$ will not hold for $p_i$ after $\Gamma_x^i$ in which $|heads_i| > k$ held before executing line 12. Therefore line 28 will not be executed in $\Gamma_{\hat{x}}^i$. The assignment in line 3 will not be executed in $\Gamma_{\hat{x}}^i$ for any $\hat{x} \in [x, x + T_{flood} - 1]$ as the condition requires $estate_i = \text{SLEEP}$. Likewise, line 16 or line 17 will not be executed in $\Gamma_{\hat{x}}^i$ for any $\hat{x} \in [x, x + T_{flood} - 1]$ as the condition in line 15 requires $estate_i = \text{SLEEP}$ as well. Thus, the only way for $estate_i$ to change strictly between pulses $\Gamma_x^i$ and $\Gamma_{x+T_{flood}}^i$ for $\hat{x} \in [x, x + T_{flood} - 1]$ is if $p_i$ receives an infotuple $(i, \text{VETO}, \cdot)$ and consequently executes line 83 setting $state_i = \text{HEAD}$. We say that $p_i$ aborts it's escape attempt in such a case. A node $p_j \in G_i^r$ will only send such a VETO as by executing line 48 by handling ESCAPING for $p_i$ which in turn only happens as a result of receiving an infotuple $(i, \text{ESCAPING}, \cdot)$.

If $p_i$ does not abort its escape attempt, it sets $estate_i$ to FLOOD in pulses $\Gamma_{x+1}^i$–$\Gamma_{x+T_{flood}-1}^i$. Therefore, by Lemma 4.5 any node $p_k \in \hat{G}_i^r$ have had a pulse that handles ESCAPING for $p_i$ at time $s + r(2\pi+1)\lambda^2 + r(\pi+1)\lambda + \lambda - 1 = s + t_s$ Moreover, as long as $p_i$ does not abort its escape attempt, in $\Gamma_{x+T_{flood}}^i$ $estate_i$ is set to SLEEP and the condition holds at line 12 after which $state_i$ is either set to HEAD in line 16 or to SLAVE in line 17. Thus the last pulse (for this escape attempt) in which $p_i$ adds $(i, \text{ESCAPING}, r, \pi)$ to $sendset_i$ is $\Gamma_{x+T_{flood}-1}^i$. That pulse cannot happen after $s + (T_{flood} - 1)\lambda$. Therefore, by Lemma 4.4 no node handles ESCAPING for $p_i$ after $s + (T_{flood} - 1)\lambda + r(\pi + 1)\lambda + (\kappa - 1)\lambda = s + t_e - \lambda$ (for this escape attempt). Thus by $s + t_e$ any node $p_j \in \hat{G}_i^r$ have handled the state $p_i$ changed to and if that is HEAD, $i \in heads_j$ again. Furthermore, by arguments similar to the proof of Lemma 4.4, after time $s + t_e + r(\pi+1)\lambda = s + T_{cool}$, no potential $(i, \text{VETO}, \cdot, \cdot)$ is left in $sendset_j$ for any node $p_j \in \hat{G}_i^r$. Therefore, a new escape attempt by $p_i$ after $s + T_{cool}$ would not be affected by remnants of the escape attempt started at time $s$. Moreover, there is at least $T_{cool}$ pulses between two escape attempts (see line 34). As we assume a legal execution from time $s - T_{cool}$, we know that no remnants from earlier escape attempts or from old invalid data in data-structures will interfere with the escape attempt started at time $s$. In the other end, before the escape attempt, $p_i$ adds $(i, \text{HEAD}, r, \pi)$ to $sendset_i$ in $\Gamma_{x-1}^i$ that can not happen after $s - 1$. By Lemma 4.2 the last possible pulse $\Gamma_y^j$ such that $p_j \in \hat{G}_i^r$ receive of $(i, \text{HEAD}, \cdot)$ in $\gamma_{y-1}^j$ (before this escape attempt) is, at $s - r(\pi - 1)\lambda - 1$. The first pulse in which HEAD is cleared from $smem_j$ is $\Gamma_{y+\kappa}^j$ which cannot happen before time $s + \kappa - r(\pi - 1)\lambda - 1 = s + t_h$.

Now, instead, assume that all nodes $p_j \in G_i^r$ have $|C_j| > k$ at $s + t_h$ and no node $p_\ell \in G_i^{2r}$, where $p_\ell \neq p_i$, initiates an escape attempt in any pulse in $[s - t_e + t_h, s + t_s - t_h]$. In $\Gamma_x^i$, node $p_i$ have $estate_i$ set to INIT and $state_i$ set to ESCAPING. We saw above that as long as $p_i$ does not receive an infotuple

$(i, \text{VETO}, \cdot)$ in pulses $\Gamma^i_{x+1}$–$\Gamma^i_{x+T_{flood}-1}$, line 11 sets $estate_i$ to FLOOD in each of those pulses.

Consider the pulse $\Gamma^j_y$ of a node $p_j \in \hat{G}^r_i$ that is the first pulse after $s$ in which $p_j$ handles ESCAPING. By Corollary 4.1 there do exist a pulse $\Gamma^j_z$ at a time $s_0 = s + r(2\pi + 1)\lambda^2 + r(\pi + 1)\lambda - 1 + t_0$ for a $t_0 \in [0, 2\lambda]$ in which $p_j$ handles ESCAPING for $p_i$. But $\Gamma^j_z$ might not be the first one, so $y \leq z$. When $p_j$ handles ESCAPING for $p_i$ in $\Gamma^j_y$, the condition at line 42 does not hold and consequently the condition at line 45 will be checked. It holds as $\Gamma^j_y$ is the first in which ESCAPING are handled – $i \in heads_j$ as HEAD must have been handled in $\Gamma^i_{y-1}$. Furthermore, the condition at line 46 will not hold, as according to the assumptions that all nodes $p_j \in G^r_i$ have $|C_j| > k$ and thereby $|heads_j| > k$, as no other node $p_\ell \in \hat{G}^{2r}_i$ have any interfering remnants of escape attempts left in the system as then cannot start after $s - t_e + t_h$ or starts escape attempts early enough (before $s + t_s - t_h$) to interfere with this escape attempt by $p_i$. Only nodes in $\hat{G}^{2r}_i$ have the potential to interfere with an escape attempt by $p_i$ as other $G^r_i \cap G^r_m = \emptyset$ for any node $p_m \notin \hat{G}^{2r}_i$. Therefore, $p_j$ do not send any infotuple $(i, \text{VETO}, \cdot)$ and $i$ is removed from $heads_j$ at line 49. This removal makes sure that $p_j$ does not allow too many nodes to escape, i.e., it ensures that $|heads_j| \geq k$. Line 50 will ensure that $i$ is added to $newheads_j$. The execution then continues down to line 7 that ensures that $i \in slaves_j$ and $i \notin heads_j$. No $(i, JOIN, r, \pi)$ is added to sendset at line 27 in $\Gamma^i_y$. By the proof of Lemma 4.3, pulses $\Gamma^j_{y+1}$, $\Gamma^j_{y+2}$, etc. will all handle ESCAPING. In these pulses the condition in line 45 does not hold as $i$ is already not in $heads_j$. Thus pulse $\Gamma^j_{y-1}$ is the last pulse (in the time interval of this escape attempt) in which $(i, \text{JOIN}, r, \pi)$ is added to $sendset_j$. By the arguments in the proof of Lemma 4.4, as long as $p_j$ continues to not send any JOIN:s, there will be no $(i, \text{JOIN}, \cdot)$ originating from $p_j$ in $smem_i$ after time $s_2 = s_1 + r(\pi + 1)\lambda + (\kappa - 1)\lambda$, where $s_1 = s_0 - \lambda \leq= s + r(2\pi + 1)\lambda^2 + r(\pi + 1)\lambda + \lambda - 1$ is the latest possible $\Gamma^j_{y-1}$. Thus, $s_2 \leq s + r(2\pi + 1)\lambda^2 + r(\pi + 1)\lambda + \lambda - 1 + r(\pi + 1)\lambda + (\kappa - 1)\lambda = s + r(2\pi + 1)\lambda^2 + r(2\pi + 2)\lambda + \kappa\lambda - 1 < s + \lceil r(4\pi + 2)\lambda^2 + r(2\pi + 2)\lambda \rceil - 1 = s + T_{flood} - 1$. Now pulse $\Gamma^i_{x+T_{flood}-1}$ can not happen before $s + T_{flood} - 1$.

Thus at pulse $\Gamma^i_{x+T_{flood}}$ in which $p_i$ sets $estate_i$ to SLEEP and the condition at line 15 therefore holds, it does not handle any JOIN for $p_j$. As this holds for any $p_j \in \hat{G}^r_i$, the condition does not hold at line 16 and $state_i$ is set to SLAVE in line 17, proving that part of the lemma.

Now, we will instead assume that there will be a node $p_j$ that handles ESCAPING for $p_j$ while having $i \in heads_j$ and $|heads_j| \leq k$. Consider the first pulse, $\Gamma^j_y$, (after $s$) in which $p_j$ handles ESCAPING. Having $i \in heads_j$, the condition at line 45 holds in $\Gamma^i_y$ and execution continues to line 46 in which the condition will also hold according to our assumptions on $p_j$. An infotuple $(i, \text{VETO}, r, \pi)$ is added to $sendset_j$, and, more importantly $j$ is added to $newheads_j$ so that $i \in heads_j$ will continue to hold after line 7 in $\Gamma^i_y$ and therefore when $p_j$ handles ESCAPING for $p_j$ in future pulses. There cannot be a case in which $p_i$ sets $state_i$ to SLAVE without any node handling ESCAPING for $p_i$, as $p_i$ can only change $state_i$ either by setting it to HEAD or changing it to SLAVE after having $state_i =$ ESCAPING for $T_{flood}$ pulses in which Lemma 4.5 guarantee that all nodes in $\hat{G}^r_i$ handles ESCAPING for $p_i$. We have already seen in Lemma 4.8 that as long as a node $p_j \in \hat{G}^r_i$ continues to send $(i, \text{JOIN}, \cdot)$ in every pulse, then $p_i$ will never set $state_i$ to SLAVE. Therefore, $p_i$ will not set $state_i$ to SLAVE as long as $|heads_j| \leq k$ continues to hold. The $(i, \text{VETO}, \cdot)$ infotuple that will reach $p_j$ is therefore not needed for the algorithm to work, but when $p_i$ receives it will set $state_i$ to HEAD and abort the escape attempt earlier to lessen the probability of different escape attempts to interfere with each other by having $p_i$ possibly stop sending out ESCAPING earlier and thus having other nodes $p_j$ put back $i$ into $|heads_j|$ earlier and thus potentially allow for other nodes to escape.

Now we still need to address the claim that $|heads_i|$ will not suddenly dip below $k$. We assume that we have a legal execution from time $s - T_{cool}$, so any remnants of bad values or old escape attempts have been cleared. Furthermore, we saw above that a node $p_\ell$ that wants to escape from being a cluster head must get clearance from all nodes in $G^r_\ell$. Thus $p_i$ would turn down the escape attempt by a node $p_\ell$ such that $\ell \in heads_i$. Therefore, $heads_i$ will not go from

$|heads_i| \geq k$ to $|heads_i| < k$. This holds for any node in $G^r_\ell$ (not merely a node $p_i$ that are in the middle of an escape attempt). ■

Theorem 4.1 shows that, within time $T_{cool} + (5r\pi + 4)\lambda \in O(T_{flood}\lambda) = O(r\pi\lambda^3)$ from an arbitrary configuration, all nodes $p_i$ have at least $k_i$ cluster heads within $r$ hops and that the set of cluster heads in the network can only stay the same or shrink from that point on. From Corollary 4.1 we get the factor $O(\lambda^2)$ time for a node to know that it has reached out. This theorem introduces another factor of $O(\lambda)$ because a node needs to be sure that another node has finished something as discussed previously.

**Theorem 4.1** *Assume that we have a legal system execution from time $s$. Then any node $p_j$ will have $k_j$ cluster heads from time $s + T_{cool} + (3r\pi + 2)\lambda$ and onward. Moreover, a node that is not in $H_t$ for a time $t \geq s+T_{cool}+(5r\pi+4)\lambda$ can not be in $H_{t'}$ for a $t' \geq t$ and consequently $|H_{t'}| \leq |H_t|$ for any $s+T_{cool}+(5r\pi + 4)\lambda \leq t \leq t'$.*

**Proof:** From Lemma 4.6 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 any pulses or pulse intervals after time $(3r\pi + 2)\lambda$. From Lemma 4.9 we get that after time $s + T_{cool}$ no node $p_i$ that is a cluster head can change $state_i$ to SLAVE if that would leave a node $p_j \in G^r_i$ with $|heads_j|$. Therefore, any node $p_j$ will have $k_j$ cluster heads from time $s + T_{cool} + (3r\pi + 2)\lambda$.

Furthermore, by Lemma 4.2 any such node $p_j$ will know it by receiving $(i, \text{HEAD}, \cdot)$ or $(i, \text{HEAD}, \cdot)$ before $\hat{s} = s + T_{cool} + (3r\pi + 2)\lambda + r(\pi + 1)\lambda$. Thus no node is trying to elect new cluster heads after time $\hat{s}$ and by similar arguments as the proof of Lemma 4.4 no node $p_\ell$ that is not already a cluster head will receive $(\ell, \text{JOIN}, \cdot)$ after $\hat{s} + r(\pi + 1)\lambda$. So with all nodes $p_j$ having $|heads_j| \geq k_j$ buy $\hat{s}$ and all JOIN:s received by $\hat{s} + r(\pi + 1)\lambda$ the condition in line 21 can only hold in $\Gamma^j_y$ after $\hat{s}$ if $|slaves_j| = 0$ and therefore no additional node can be elected. Thus, a node that is not in the set of cluster heads in the entire network at a time $t \geq s + T_{cool} + (5r\pi + 4)\lambda$, $H_t$, can never be in $H_{t'}$

for a $t' \geq t$. Therefore, $|H_{t'}| \leq |H_t|$ for any $t' \geq t \geq s + T_{cool} + (5r\pi + 4)\lambda \in s + O(r\pi\lambda^3)$. ■

### 4.4.3 Convergence to a Local Minimum

Lemma 4.9 shows that a cluster head node that is not needed can escape the cluster head responsibility if it does not interfere with escape attempts by other nodes. This section shows that the set of cluster heads converges to a local minimum. We first show that an unneeded cluster head node can escape, with high probability (Lemma 4.10) in $O(T\lambda) = O(gr\pi\lambda^4)$ time. The extra $\lambda$ is due to the usual reason and $T \in O(gr\pi\lambda^3)$. The factor $g$ is a bound on the number of nodes that could interfere with a given escape attempt. It is part of $T$ to give a node a constant probability of escaping in its one try in a period of $T$ time (given that it is not needed as a cluster head).

**Lemma 4.10** *Assume that we have a legal system execution from time $s$ and consider a node $p_i$ that is a cluster head. Assume that $|C_j^r| > k$ holds for all nodes $p_j \in \hat{G}_i^r$ from time $s + T_{cool} + (5r\pi + 4)\lambda$ and as long as $p_i$ remains a cluster head. Then, node $p_i$ will have $state_i =$ SLAVE after time $s + T_{cool} + (5r\pi + 4)\lambda + (\beta + 1)T\lambda$ with at least probability $1 - 2^{-\beta}$.*

**Proof:** As long as node $p_i$ remain a cluster head it will in each pulse enter the branch in line 10 and call $updateestate$ in line 11. From Theorem 4.1 we know that from time $s_0 = s + T_{cool} + (5r\pi + 4)\lambda$ a node can only stay slave or to go from being a cluster head to being a slave. Assume that $p_i$ *starts a period* in $\Gamma_x^i$ at time $\hat{s}$, i.e., $timer_i = 0$ at line 34 in $\Gamma_x^i$ and that it is the first pulse, for which $\hat{s} \geq s_0$, that starts a period. By the system settings and the fact that $p_i$ starts a new period every $T$ pulses, we get that $\hat{s} \leq s + T_{cool} + (5r\pi + 4)\lambda + T\lambda$.

In $\Gamma_x^i$, $estart_i$ is set randomly, from a uniform distribution, to an integer in $[0, T_{es} - 1]$. Thereby, $p_i$ is scheduled to initiate an escape attempt in pulse $\Gamma_w^i$, for $w = x + estart_i$ at time $s'$. With our assumptions, Lemma 4.9 gives that if no other node $p_j \in \hat{G}_i^{2r}$ initiates any escape attempt in the time span $[s' - t_e + t_h, s' + t_s - t_h]$, then $p_i$ will set $state_i =$ SLAVE in $\Gamma_{w+T_{flood}}^i$. If no such escape attempts are done by any such node $p_j$ we say that node $p_i$ initiates an *uninterfered escape attempt* in $\Gamma_w^i$.

The maximum number escape attempts by a node $p_j \in \hat{G}_i^{2r}$ that can interfere with the escape attempt initiated by $p_i$ in $\Gamma_w^i$ is two (as the time between the first and last of three escape attempts is larger than $T$ which is larger than required time span for Lemma 4.9. Thus, the maximum number or interfering escape attempts that can interfere with the escape attempt that is initiated by $p_i$ in $\Gamma_w^i$ is $2(g-1)$. The probability that a node $p_j$ initiates an escape attempt in a time interval $[t_0, t_1]$ is always less than $(t_1 - t_0 + 1)/T_{es}$. Thus the probability that a specific escape attempt will interfere with the escape attempt initiated by $p_i$ in $\Gamma_w^i$ cannot, with the help of Lemma 4.9, be more than $\rho = \hat{t}/T_{es}$ for $\hat{t} = t_s + t_e - 2t_h + 1$. By making the extremely *pessimistic* assumption (i.e., that gives us a higher probability for interference than what is really the case) that we not only have $2(g-1)$ potentially interfering escape attempts, but that they are all independent as well. Escape attempts made by the same node $p_j$ is not really independent, but by assuming they are we increase the probability of interference. Let $A_w^i$ be the event that the escape attempt initiated in $\Gamma_w^i$ is uninterfered. Then we get

$$\begin{aligned}
\mathrm{P}[A_w^i] \geq (1-\rho)^{2(g-1)} &= \left[\rho' := \frac{1}{\rho}\right] = \left(\left(1 - \frac{1}{\rho'}\right)^{\rho'-1}\right)^{(2(g-1))/(\rho'-1)} \\
&> \left(\frac{1}{e}\right)^{(2(g-1))/(\rho'-1)} = \exp\left(-\frac{2(g-1)}{\rho'-1}\right) \qquad (4.1) \\
&= \exp\left(-\frac{2(g-1)}{1/\rho - 1}\right). \qquad (4.2)
\end{aligned}$$

From this it is straightforward to show that:

$$T_{es} = \lceil \frac{2g}{\ln 2}(t_s + t_e - 2t_h + 1) \rceil \Rightarrow \mathrm{P}[A_w^i] \geq \frac{1}{2}. \qquad (4.3)$$

The random choices in different executions of the line 34 by the same node or by different nodes are all mutually independent. However, what happens in different periods are not mutually independent. If a node $p_j$ sets $state_j =$ SLAVE not initiating any more escape attempts. This only increases the probability for $A_{w'}^i$ for a later escape attempt in $\Gamma_{w'}^i$ if the one initiated in $\Gamma_w^i$ was not uninterfered. Therefore, by assuming total independence between all escape attempts and continue to assume that there are always $2(g-1)$ potentially interfering escape attempts for an escape attempt by node $p_i$, the calculated lower bound on the probability of an undisturbed escape attempt gets worse.

Consider the $\beta$ periods that starts in $\Gamma^i_x$, $\Gamma^i_{x+T}$, ..., $\Gamma^i_{x+(\beta-1)T}$, and let $w_a$ be the value such that $\Gamma^i_{w_a}$ is the pulse in the period starting in $\Gamma^i_{x+a\cdot T}$ in which the escape attempt starts. Furthermore, let $A^i = \bigcup_{t=0}^{\beta-1} A^i_{w_a}$. Thus with the assumption of period independence that gives us a worse bound we get

$$\begin{aligned} P[A_i] =& P\left[\bigcup_{a=0}^{\beta-1} A^i_{w_a}\right] = 1 - P\left[\bigcap_{a=0}^{\beta-1} \bar{A^i_{w_a}}\right] = 1 - \prod_{a=0}^{\beta-1} P[A_i] \\ >& 1 - \prod_{a=0}^{\beta-1} \frac{1}{2} = 1 - 2^{-\beta}. \end{aligned} \tag{4.4}$$

With $\hat{s} \leq s + T_{cool} + (5r\pi + 4)\lambda + T\lambda$, we get that, with probability at least $1 - 2^{-\beta}$, $state_i =$ SLAVE after time $s + T_{cool} + (5r\pi + 4)\lambda + (\beta + 1)T\lambda$. ■

Theorem 4.2, shows that with high probability the entire network reaches a local minimum within $O(r\pi\lambda^4 g \log n)$ time.

From Theorem 4.1 we got that all nodes $p_i$ have at least $k_i$ cluster heads within $r$ hops in $T_{cool} + (3r\pi + 2)\lambda$ time after an arbitrary configuration.

Theorem 4.2 shows that with at least probability $1 - 2^{-\alpha}$ the set of cluster heads in the network stabilizes to a local minimum within $s + T_{cool} + (5r\pi + 4)\lambda + (\alpha + \log n + 1)T\lambda$ time. The factor $O(\log n)$ is multiplied by the result from Lemma 4.10 because we go from probabilistic guarantee that one specific node gets an uninterrupted escape attempt to that all cluster heads get such an attempt. And the number of cluster heads is bounded by the number, $n$, of nodes in the network.

**Theorem 4.2** *Assume that we have a legal system execution from time $s$. With at least probability $1 - 2^{-\alpha}$, by time $\hat{s} = s + T_{cool} + (5r\pi + 4)\lambda + (\alpha + \log n + 1)T\lambda$ there will be no cluster head node $p_i$ in the network for which $\min_{p_j \in G_i}(|C^r_j|) > k$ holds, and $H_{\hat{s}+t} = H_{\hat{s}}$ holds for any positive $t$.*

**Proof:** In this proof we use the same notations and the concepts as in the proof of Lemma 4.10. Here $w_a$ is replaced by $w_a^i$ to indicate that the escape attempt in the $a$:th period of $p_i$ after $s + T_{cool} + (5r\pi + 4)\lambda$ is initiated in $\Gamma^i_{w_a^i}$.

Let $\beta = \alpha + \log n$, where $n = |\mathcal{P}|$ is the size of the entire network. Let $A$ be the event all nodes in the network get at least one uninterfered escape attempt in $[s + T_{cool} + (5r\pi + 4)\lambda, \hat{s}]$. We get that

$$\begin{aligned} P[A] =& 1 - P[\bar{A}] = 1 - P[\bigcup_{p_i \in \mathcal{P}} \bar{A}_i] \geq \text{[Boole's inequality]} \\ \geq& 1 - \sum_{p_i \in \mathcal{P}} P[\bar{A}_i] \geq \text{[Lemma 4.10]} \geq 1 - \sum_{p_i \in \mathcal{P}} 2^{-\beta} \\ =& 1 - n2^{-\beta} = 1 - 2^{\log n - \beta} = 1 - 2^{-\alpha}. \end{aligned} \tag{4.5}$$

Thus by the proof of Lemma 4.10 all nodes in the network gets an uninterfered escape attempt with at least probability $1 - 2^{-\alpha}$ in $[s + T_{cool} + (5r\pi + 4)\lambda, \hat{s}]$. Together with Lemma 4.10, this concludes that with at least probability $1 - 2^{-\alpha}$ all nodes $p_i$ for which $|C_i^r| > k$ holds at their uninterfered escape attempt will have set $state_i$ to SLAVE before time $\hat{s}$. From this follows that there is no node for which $\min_{p_j \in G_i}(|C_j^r|) > k$ holds. Hence, by Lemma 4.9, no node $p_i$ that is cluster head at time $\hat{s}$ can ever set $state_i$ to SLAVE after $\hat{s}$ and therefore $H_{\hat{s}+t} = H_{\hat{s}}$ holds for any positive $t$. Moreover, $\hat{s} \in s + O(r\pi\lambda^4 g \log n)$. ■

## 4.5 Discussion

To experimentally test performance, we did simulations of the algorithm for various settings of $k$ and $r$. We placed $40$ nodes with a communication radius of 1 uniformly at random in a 5 by 5 rectangular area. From our experiments we concluded that using a $g$ that gives us 95% guarantees of being an upper bound on every $|G_j^{2r}|$ for any given $p_j$, is not required to get good performance. The calculated bounds are not tight. In the experiments we have therefore used a tenth of that value for $g$ instead.

In addition, we performed experiments on recovery from small changes to the topology from a converged system state. The convergence times from a

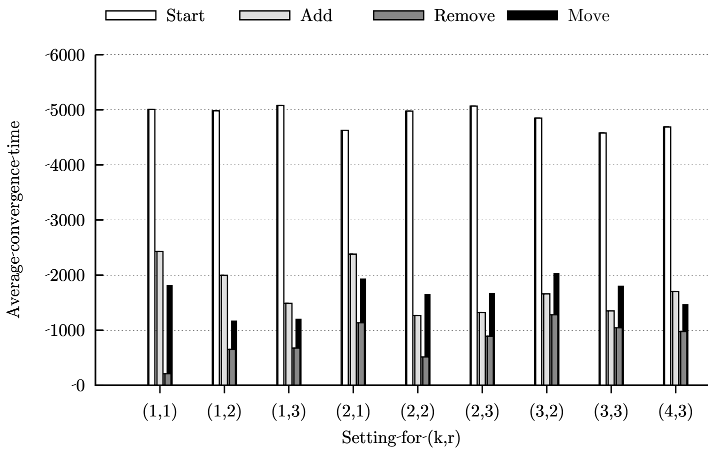

**Figure 4.4:** *Convergence times from a fresh start, after 10% node additions, after 10% node removes and after 10% node moves.*

newly started network ("Start") is compared in Figure 4.4 with the convergence times after a change to a initially converged network. We investigate $10\%$ added nodes ("Add"), $10\%$ removed nodes ("Remove") or $10\%$ moved nodes ("Move").

We can see that the least obtrusive change to the topology is removed nodes. The chance is good that a removed node is not a cluster head and thus do not upset the balance. An add is more expensive than a remove. Nodes might end up in an area where there is not so many cluster heads and therefore have to start elect new nodes. A move is like both a remove and an add. Therefore, it is anticipated that this case converges slower than the ones with only adds or only removes.

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 applications 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, [15]) or node id:s for nodes that are not within $G_c^r$ (wormhole attacks, [16]) 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 or having them repeatedly go on and off cluster head duty over time by alternating between sending joins and letting the node escape. 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 bottleneck between nodes without any other paths between them then it can lie about a node $p_\ell$ being a cluster heads and refuse to forward any joins to $p_\ell$. 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 $p_\ell$ that nodes not in $G_\ell^r$ are cluster heads. In the worst case it can eventually make $p_\ell$ 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 $G_c^{2r}$ as the maximum $ttl$ of a message is $r$ and is enforced at message receipt.

## 4.6 Conclusions

We have presented a self-stabilizing $(k, r)$-clustering algorithm for ad-hoc networks that can deal with a bounded amount of message loss, and that merely requires a bound on the rate differences between pulses of different nodes in the network. The algorithm makes sure that, within $O(r\pi\lambda^3)$ time, all nodes have at least $k$ cluster heads (when possible) and it stabilizes within $O(r\pi\lambda^4 g \log n)$ time with high probability. We have also discussed how the algorithm can help us with fault tolerance and security.

## Bibliography

[1] S. Dolev, *Self-Stabilization,* MIT Press, 2000.

[2] Y. P. Chen, A. L. Liestman, and J. Liu, *Clustering Algorithms for Ad Hoc Wireless Networks*, vol. 2, chapter 7, pp. 154–164, Nova Science Publishers, 2004.

[3] A. A. Abbasi and M. Younis, "A survey on clustering algorithms for wireless sensor networks," *Comput. Commun.*, vol. 30, no. 14-15, pp. 2826–2841, 2007.

[4] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds., pp. 85–103. Plenum Press, 1972.

[5] C. Johnen and L. H. Nguyen, "Robust self-stabilizing weight-based clustering algorithm," *Theor. Comput. Sci.*, vol. 410, no. 6-7, pp. 581–594, 2009.

[6] S. Dolev and N. Tzachar, "Empire of colonies: Self-stabilizing and self-organizing distributed algorithm," *Theor. Comput. Sci.*, vol. 410, no. 6-7, pp. 514–532, 2009.

[7] E. Caron, A. K. Datta, B. Depardon, and L. L. Larmore, "A self-stabilizing k-clustering algorithm using an arbitrary metric," in *Euro-Par*, 2009, pp. 602–614.

[8] Y. Afek and G. Brown, "Self-stabilization over unreliable communication media," *Distributed Computing*, vol. 7, pp. 27–34, 1993.

[9] V. Ravelomanana, "Distributed k-clustering algorithms for random wireless multihop networks," in *ICN 2005*, pp. 109–116. Springer, 2005.

[10] Y. Fu, X. Wang, and S. Li, "Construction k-dominating set with multiple relaying technique in wireless mobile ad hoc networks," in *CMC '09*, Washington, DC, USA, 2009, pp. 42–46, IEEE Computer Society.

[11] M. A. 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.

[12] Y. Wu and Y. Li, "Construction algorithms for k-connected m-dominating sets in wireless sensor networks," in *MobiHoc '08*, New York, NY, USA, 2008, pp. 83–90, ACM.

[13] K. Sun, P. Peng, P. Ning, and C. Wang, "Secure distributed cluster formation in wireless sensor networks," in *ACSAC '06*, Washington, DC, USA, 2006, pp. 131–140, IEEE Computer Society.

[14] A. Larsson and P. Tsigas, "A self-stabilizing (k,r)-clustering algorithm with multiple paths for wireless ad-hoc networks," in *ICDCS 2011*, Minneapolis, MN, USA, 2011.

[15] J. Newsome, E. Shi, D. Song, and A. Perrig, "The sybil attack in sensor networks: analysis & defenses," in *IPSN '04*, New York, NY, USA, 2004, pp. 259–268, ACM.

[16] Y. Hu, A. Perrig, and D. B. Johnson, "Wormhole detection in wireless ad hoc networks," Tech. Rep., Rice University, Department of Computer Science, 2002.

# 5 Future Work

Wireless sensor networks is an upcoming area that holds great promise for the future. When the sensor nodes become smaller and smaller and cheaper and cheaper the possibilities opens up for a vast number of different applications. However, as with many areas in its infancy, there are a number different problems that arises when taking these applications from ideas to deployed systems with interaction from the environment and from attacks that cannot always been foreseen.

Attacks from within the network from nodes taken over by an adversary is a serious threat in a network where the nodes are physically accessible. Therefore, there is a great need for algorithms that can function even in situations where nodes inside the network attacks the network from within. Collusion between several malicious nodes makes the situation even worse. In addition, algorithms can not rely on having an undisturbed deployment and setup of the network before having to care about attacks. Otherwise, if malicious nodes already have been deployed in the area, they can become inside nodes just by following protocol during the setup.

In addition to malicious attacks, problems arises from the general environment. Sensor nodes are often placed in hostile environments that can disturb their functionality, make them faulty and outright destroy them. It is therefore important to be able to handle loss of nodes, faulty behavior of nodes and other faults in the system. Algorithm are needed that take both security and fault tolerance into account.

In research, there is always a need for proper models. In the area of security in sensor networks, two important modelling aspects are the model of the adversary and the model of the network and the communication. What are the powers of the adversary. What can be expected in terms of computing power, what techniques are feasible on the hardware level, how many malicious node can we expect the adversary to have command over and what are the limitations of them, etc. For the communication, what kind of assumptions can be made about connectivity between nodes? How reliable is the communication medium, can we rely upon bidirectional communication, are the communication links stable over time, are nodes moving around, etc.?

Models, although essential, are just models. An important characteristic of an algorithm is what happens if the assumptions of a model is broken and what happens after temporary deviations from assumptions. Depending on in which way the assumptions are broken, functionality can be unaffected, degraded or reduced to nothing. Fault tolerance in general tries to foresee problems that can arise and take those into account within the model. However, at some point those assumptions can be broken as well. In the situation of a large sensor network, trying to recover manually after a break down can be unfeasible or too costly. This is why self-stabilization is a useful technique for sensor network. Regardless of what state the network has ended up in due to deviations from assumptions, if the assumptions once again hold, the network can recover. Self-stabilization most often comes with a price in terms of overhead and not being able to ever stop, but the fault tolerance properties gained by it are powerful. When implementing a sensor network application, this tradeoff needs to be considered. More research is needed to find self-stabilizing algorithms that are suitable for sensor networks and that are as resource efficient as possible.

Many number of services are needed when building a sensor network application. The technique of layering services on top of each other, building functionality on top of functionality and at the same time separate them into manageable units is powerful. In sensor networks where battery power is of high concern and message transmissions are the most expensive operations, the separation between services can pose a problem. If many different algorithms, implementing the needed services, transmit partly the same information, battery power is wasted. In resource constrained sensor networks, the need is great for algorithms that play well together in an efficient manner.

In a society where sensor networks exist in abundance, privacy is of utmost importance. When we have sensor networks in public spaces, in the offices, inside the body for medical applications, in the electricity networks, etc., a lot of information, possibly highly sensitive, is handled by these networks. Here research is needed to establish methods to handle privacy in an efficient manner. The privacy needs is for knowing users of the networks, such as when sensors collects medical information from a patient, as well as for general monitoring of environments that results in peoples personal information being handled by the sensor network.

To conclude, research needs to be done to further develop secure and fault tolerant algorithms with good models of the environment that can be combined in an resource efficient manner.