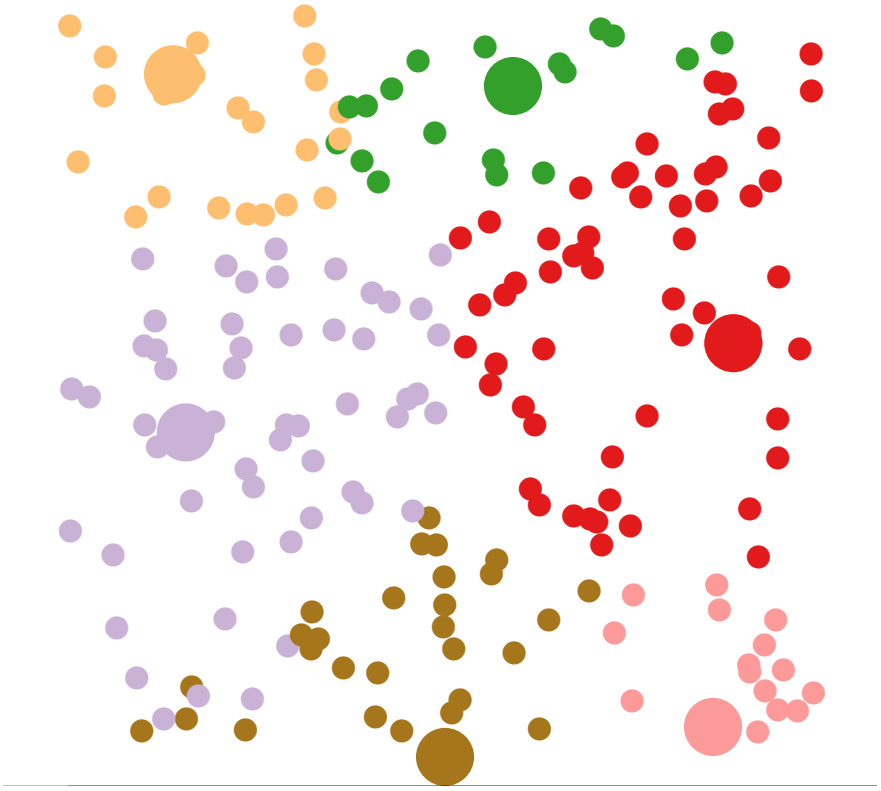


Bringing Order to Chaos

Clustering in Wireless Sensor Networks

Master's thesis in Computer Systems and Networks

Mattias Nilsen
André Samuelsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Bringing Order to Chaos

Clustering in Wireless Sensor Networks

Mattias Nilsen
André Samuelsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Bringing Order to Chaos
Clustering in Wireless Sensor Networks
Mattias Nilsen
André Samuelsson

Supervisor: Olaf Landsiedel, Computer Science and Engineering
Examiner: Marina Papatriantafilou, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example of a network with 200 nodes, clustered by the process presented in this thesis. Each colour represents a different cluster and the large nodes are cluster heads.

Typeset in LaTeX
Gothenburg, Sweden 2018

Bringing Order to Chaos
Clustering in Wireless Sensor Networks
Mattias Nilsen
André Samuelsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Wireless Sensor Networks are becoming more and more popular with the decrease in cost to manufacture sensor nodes and the increased popularity of cyber-physical systems. One of the most significant challenges for Wireless Sensor Networks is to minimise the energy consumption of the nodes, as their battery capacity is often limited, and they are expected to work without human intervention for several years at a time. Another challenge for Wireless Sensor Networks is scalability since they may scale to thousands of nodes. Clustering is a widely used technique to both decrease energy consumption and increase scalability. In this thesis, we aim to increase the network lifetime and the scalability of the A^2 system, by integrating clustering with it. Our starting point, A^2, is a system which brings distributed consensus to multi-hop networks implemented on ContikiOS. Our work consists of designing and implementing a clustering scheme, based on the HEED clustering algorithm, to partition the network and create a hierarchical communication medium. We evaluate our work in the Cooja simulator and on the Flocklab testbed, and compare it to the original implementation of A^2 using the metrics stability, reliability, latency, and energy usage. Our evaluation shows that we achieve similar reliability to the A^2 system but lower stability. However, for the largest network we evaluated, with 200 nodes, we achieve both better latency and lower energy consumption.

Keywords: Wireless Sensor Networks, Scalability, Clustering, HEED, Chaos, A^2 Synchrotron

Acknowledgements

We would like to thank our supervisor Olaf Landsiedel for his invaluable support and continuous feedback during the project. We would also like to thank Beshr Al Nahas for answering our technical questions regarding A^2. Furthermore, we thank our examiner Marina Papatriantafilou for her feedback on the report.

Mattias Nilsen & André Samuelsson, Gothenburg, August 2018

Contents

List of Figures xiii

List of Tables xv

List of Algorithms xvii

1 Introduction 1
- 1.1 Problem and Aim . . . 2
- 1.2 Limitations . . . 2
- 1.3 Contributions . . . 3
- 1.4 Thesis Outline . . . 3

2 Background 5
- 2.1 Wireless Sensor Networks . . . 5
- 2.2 Chaos . . . 6
  - 2.2.1 Synchronous Transmission . . . 6
  - 2.2.2 The Initiator . . . 7
  - 2.2.3 Association . . . 7
  - 2.2.4 Flags Field . . . 8
  - 2.2.5 Completion Flooding . . . 8
  - 2.2.6 Timeout Mechanism . . . 8
- 2.3 Agreement in the Air . . . 8
  - 2.3.1 The Scheduler . . . 9
  - 2.3.2 Dynamic Group Membership . . . 9
  - 2.3.3 Frequency Agility . . . 10
- 2.4 Gossiping . . . 11
- 2.5 Clustering . . . 12
  - 2.5.1 HEED . . . 12
- 2.6 Clustering Objectives . . . 13
  - 2.6.1 Primary Clustering Objectives . . . 13
  - 2.6.2 Secondary Clustering Objectives . . . 14
- 2.7 Clustering Technology Properties . . . 15
  - 2.7.1 Cluster Properties . . . 15
  - 2.7.2 Cluster Head Properties . . . 16
  - 2.7.3 Clustering Process Properties . . . 16

3 Related Work **19**
3.1 UHEED . . . 19
3.2 BCDCP . . . 19
3.3 LEACH . . . 20
3.4 Extreme Chaos . . . 21
3.4.1 Estimation Vector . . . 21
3.4.2 Flow Control . . . 22
3.5 Discussion . . . 22

4 Design of the Clustering Process **25**
4.1 Clustering Process Overview . . . 25
4.2 The Clustering Service . . . 26
4.2.1 Designing for Scalability . . . 26
4.2.2 Transmission Policy with Gossiping . . . 26
4.2.3 Election of Cluster Heads . . . 26
4.2.4 The Final Phase . . . 28
4.2.5 Configuration Parameters . . . 29
4.3 Joining Clusters . . . 30
4.4 Demotion of Cluster Heads . . . 30
4.5 Communication . . . 31
4.5.1 Intra-cluster . . . 31
4.5.2 Inter-cluster . . . 31
4.6 Clustering Objectives and Properties . . . 32
4.7 Discussion . . . 33
4.7.1 Comparing the Clustering Service's Communication to Gossiping . . . 34
4.7.2 Limiting the Number of Cluster Heads . . . 34
4.7.3 The Final Phase . . . 35
4.7.4 The Cost Function . . . 35
4.7.5 Separating Cluster Communication . . . 36
4.7.6 Clustering Risks A^2's Fault Tolerance . . . 36

5 Implementing Clustering in A^2 **37**
5.1 Modifications to A^2 . . . 37
5.1.1 Flags Field for Cluster Heads . . . 37
5.1.2 Separating Communication Between Clusters . . . 38
5.1.3 Forwarders During Cluster Head Rounds . . . 38
5.1.4 Initiating Communication in a Clustered Network . . . 39
5.1.5 Interaction Between Services . . . 40
5.2 The Clustering Service . . . 40
5.2.1 The Clustering Service Packet Payload . . . 41
5.2.2 The Catch-up Mechanism . . . 41
5.2.3 The Demote Service . . . 43
5.3 Discussion . . . 43
5.3.1 Destructive Interference During CH Rounds . . . 43
5.3.2 Forwarders during CH rounds . . . 44
5.3.3 Fault Tolerance for Dynamic Initiators . . . 44

5.3.4 Clustering Without Completion Flags . . . 45
5.3.5 Completion Flags during Cluster Head Rounds . . . 45

6 Evaluation **47**
6.1 Evaluation Setup . . . 47
6.1.1 The Cooja Simulator . . . 47
6.1.2 The Flocklab Testbed . . . 48
6.1.3 Metrics . . . 48
6.1.4 Limitations . . . 50
6.2 Clustering Parameters . . . 51
6.2.1 Round Re-synchronisation Threshold . . . 51
6.2.2 Competition Radius . . . 53
6.2.3 Minimum Cluster Size . . . 54
6.2.4 Nodes per Cluster Ratio . . . 55
6.3 Comparing A^2 with Clustered A^2 . . . 56
6.3.1 Reliability and Stability . . . 57
6.3.2 Latency . . . 57
6.3.3 Energy . . . 58
6.3.4 Flocklab . . . 59
6.4 Discussion . . . 60
6.4.1 Stability and Fault Tolerance . . . 60
6.4.2 Clustering Parameters . . . 62
6.4.3 Running Other Applications in a Clustered Network . . . 63
6.4.4 Achieving Better Scalability . . . 64

7 Conclusion **67**
7.1 Future Work . . . 67
7.1.1 Sleep Schemes . . . 67
7.1.2 Transmission Power . . . 68
7.1.3 Fault Tolerance . . . 68
7.1.4 Measuring Throughput . . . 68
7.1.5 Using Other Clustering Algorithms . . . 68

Bibliography **69**

A Appendix 1 **I**
A.1 Re-synchronization Latency Results . . . I
A.2 Parameter Latency Plots . . . II

B Appendix 2 **III**
B.1 Parameter Values for the A^2 Comparison . . . III

List of Figures

2.1 An example of a Chaos round were three nodes reach consensus on the maximum of three proposed values [3]. 7
2.2 The original A^2 architecture [4]. 9
2.3 The Join service exemplified using three nodes [4]. 10

5.1 **Layered system architecture of A^2 [4].** Shows the placement of the Cluster and Demote services. The Cluster module in Synchrotron handles per-cluster channel hopping and dynamic assignment of initiators. The Clustering and Demote services in A^2 contains the logic for electing and demoting cluster heads respectively. 38
5.2 The network on the right has elected the cluster heads in the CH list. Shown here is how Node 6 uses the flags field during CH and cluster rounds. The symbol 'x' represents the index which Node 6 uses as its flag, and the symbol '-' represents flag indices which are in use by other nodes. An empty box indicates that the index is not used. In the CH-round CHs uses the index from the CH-List. During a cluster round each CH will be an initiator and use index 0. 39
5.3 An example network and its application map. Showing which applications the network executed, and which nodes are initiators during the different phases. 40
a The first 50 rounds for a network with 10 nodes, and which application each node is running. Node 1 is the initiator and thus runs the Clustering service in round 1 while the rest of the nodes associates with it. 40
b Locations of the ten nodes showing three different initiator setups. Node 1 is initiator rounds 1-20 and 25-30. Nodes 5 and 9 are initiators in rounds 21-24, 31-36 and every cluster (odd) round in the interval 37-50. Node 5 is the initiator in every CH (even) round in the interval 37-50. 40

6.1 Resynchronisation threshold tests for different values of Competition radius. Both reliability and stability increases as the re-sync threshold increases. 53
a Re-synchronisation threshold 1. 53
b Re-synchronisation threshold 2. 53
c Re-synchronisation threshold 3. 53

6.2 The reliability and stability for different network sizes. Each network size has been tested with competition radius 1, 2, and 3. 54
6.3 The reliability and stability for different network sizes. Each network has been tested with minimum cluster size values off, 2, and 4. 55
6.4 Shows the number of CHs after the clustering process has finished, and the number of demoted CHs. 55
6.5 The reliability and stability for different network sizes. Each network size has been tested with max node count off, 5, 10, and 15. 56
6.6 The number of elected CHs before the Demote service runs, and the number of demoted CHs. 56
6.7 Reliability and stability comparison between A^2 with clustering and original A^2. 58
a Networks with 50 nodes. 58
b Networks with 200 nodes. 58
6.8 Latency comparison between A^2 with clustering and original A^2. . . . 59
a Networks with 50 nodes. 59
b Networks with 200 nodes. 59
6.9 Energy comparison between A^2 with clustering and original A^2. . . . 59
a Networks with 50 nodes. 59
b Networks with 200 nodes. 59
6.10 The results of running A^2 and our clustering implementation on the Flocklab testbed. We show the mean and min/max for stability and reliability, and the mean and standard deviation for latency and energy consumption. 60
a Reliability. 60
b Stability. 60
c Latency. 60
d Energy. 60
6.11 Example of what happens when a node requests the Join service to be scheduled, blue is the correct application, green is the Join service, and yellow is the Clustering service. The first cluster repeatedly schedules the Join service without any effect. 61

A.1 Competition radii tests for different values of resynchronisation threshold. I
a Re-synchronisation threshold 1. I
b Re-synchronisation threshold 2. I
c Re-synchronisation threshold 3. I
A.2 The latency results for the parameter tests. II
a Competition radius. II
b Minimum cluster size. II
c Nodes per cluster ratio. II

List of Tables

2.1 An overview of primary and secondary clustering objectives [5]. . . . 13
2.2 Cluster properties and their options. 15
2.3 Cluster head properties and their options. 16
2.4 Clustering process properties and their options. 17

4.1 Our primary and secondary clustering objectives. 32
4.2 Our properties related to clusters, cluster heads and the clustering process. 32

5.1 The parameters and their sizes in the Clustering service packet payload. 41
5.2 Probability that a node has announced itself as CH in a specific round without the Catch-up mechanism. 42
5.3 Probability that a node has announced itself as CH in a specific round with the Catch-up mechanism. 42

6.1 The parameters we evaluate and their default values 50

B.1 List of topologies with 50 nodes and the competition radius used for each topology. III
B.2 List of topologies with 200 nodes and the nodes per cluster ratio used for each topology. III

List of Algorithms

1 The repeat phase adaptation of the HEED algorithm. It shows how a node elects to announce itself as cluster head. The algorithm is adapted for A^2 in two ways. We utilise our parameter $nodesPerClusterRatio$ at Line 3 and set the announcement slot at Line 15. 28

2 The final phase of the clustering algorithm. The only edit from the original HEED final phase is the usage of $pickBestCH$, which is our definition of the cost function that the HEED algorithm requires. . . 29

3 Our cost function for picking the best cluster head. 30

1

Introduction

The technology of Wireless Sensor Networks (WSNs) is an active research area [1, 2]; it enables small low-powered computer nodes to work in cooperation. The first application of Wireless Sensor Networks was for different military applications such as target tracking and troop movements [1]. However, WSNs have quickly expanded to many other fields, such as natural disaster tracking and weather monitoring. The nodes are battery powered, with limited processing and storage capabilities. Equipped with a low-power radio, they can communicate with other nearby nodes. They also use sensors to collect data about the environment around them. This data is then used as input into some computation and either forwarded to a base station or disseminated throughout the network.

One advantage of WSNs is the ability to deploy nodes in the network in an ad hoc manner. Furthermore, the nodes' low powered nature make them inexpensive to manufacture. However, there are multiple challenges which require thorough consideration when implementing protocols for a WSN [1]. Since nodes are battery powered, they should restrict the time they spend sending, receiving, and processing data. Another difficulty is maintaining high message propagation speed while not congesting the network. Additionally, scaling is a challenge and restrictions on the number of nodes that can participate in a WSN appear in many forms. For example, the maximum size of the transmitted packets, the communication protocol, and the network topology, can all impact the scalability of a WSN.

The Chaos protocol [3], presented in 2013, is the first protocol built for WSNs to have native support for all-to-all data sharing. Chaos builds on two core mechanisms: synchronous transmissions and user-defined merge operators. Synchronous transmissions mean that the nodes in a network follow a global schedule that tells them when to wake up and either transmit or receive data. User-defined merge operators consist of some code that defines how a node processes and merges received data. For example, calculating the maximum value over a set of proposed values. By using these two mechanisms, a node running the Chaos protocol can independently decide what action to perform in the next slot: sending data or receiving data. Additionally, the nodes perform all processing (the execution of a merge operator) as part of the network protocol after they transmit or receive data.

Further development of the Chaos protocol resulted in A^2 [4]. A^2 addresses some shortcomings of the Chaos protocol and also introduces the A^2 *Synchrotron*, a syn-

chronous transmission kernel which has several features: frequency hopping, high precision time synchronisation, and the ability to schedule multiple applications to run in the network at different intervals. On top of Synchrotron, A^2 implements distributed consensus protocols such as two- and three-phase commit. Due to the new communication model in Chaos and the new features in the A^2 system, both Chaos and A^2 shows significant improvement in performance and reliability compared to similar protocols [3, 4].

A typical way to improve the network lifetime and scalability in WSNs is to use clustering [5, 6]. Clustering is the practice of electing a set of nodes as *Cluster Heads* (CH) and assigning each remaining node to one of these CHs. Nodes selecting the same CH belong to the same cluster and will only communicate within that subset of the network. The benefit of clustering is primarily reducing the number of packet transmissions and consequently the amount of data the network has to handle. Since each CH only has to communicate with a subset of the network, the number of packet transmissions in each cluster is reduced by a factor approximately equal to the number of CHs in the network. Furthermore, the CHs can aggregate and filter out redundant data from within their clusters, before forwarding it to a base station, or disseminating the aggregate to the network.

1.1 Problem and Aim

Building on A^2 Synchrotron, we aim to increase network lifetime and improve scalability while maintaining A^2's high probability of reaching consensus, which is higher than 99%. To achieve these goals, we extend A^2's design with a clustering mechanism. We evaluate the implementation of the new design and compare it to the A^2 Synchrotron using the following metrics:

- Reliability measures the success rate of an application running on the network.
- Stability measures the number of topology changes that occur for the network.
- Latency measures how quick a network executes an application and terminates.
- Energy usage measures the energy consumed by the radio and CPU of a node.

1.2 Limitations

We impose several limitations on the design and implementation of our clustering algorithm. We will not design a new clustering algorithm specifically for the A^2 system. We will consider an existing clustering algorithm and only make minor modifications to implement the algorithm on the A^2 system. Furthermore, we will only provide a reference implementation for the Clustering service, and not implement features such as fault tolerance for crashing nodes or corrupted data.

1.3 Contributions

In this thesis, we make the following contributions.

- We design a clustering scheme for the A^2 system, based on the HEED algorithm [7].
- We provide an implementation of the clustering scheme in Contiki OS [8].
- We evaluate our reference implementation in Cooja [9], the simulator for Contiki OS, and on the Flocklab testbed [10].

1.4 Thesis Outline

We organise the rest of this thesis as follows. In Chapter 2 we introduce WSNs, Chaos and the A^2 system, as well as the areas of clustering and gossiping. In Chapter 3 we present research on clustering and another thesis attempting to increase the scalability of Chaos. In Chapter 4 we present our design of the clustering implemented on top of A^2. In Chapter 5 we talk about the implementation specific details and issues encountered while implementing clustering on top of A^2 in Contiki. In Chapter 6, we present our evaluation of the clustering algorithm and comparison to the existing A^2 system. Finally, in Chapter 7, we conclude and list future work.

2

Background

In this chapter, we introduce the technology of Wireless Sensor Networks, the Chaos protocol, the A^2 system, and gossiping protocols. Furthermore, we explain what clustering is and provide insight into different objectives when clustering to a network. Finally, we present a categorisation of properties for clustering algorithms.

2.1 Wireless Sensor Networks

A *Wireless Sensor Network* (WSN) consists of many small low powered computer nodes equipped with sensors, cooperatively working towards a goal. Typical applications include target tracking for the military, monitoring the weather, and tracking natural disasters [1]. There are several characteristics of a WSN that introduces challenges. However, they are also at the core of what makes WSNs useful. Examples of these characteristics are the low power nature and ad hoc deployment of the nodes. A typical node consists of four hardware units: sensors, processor, radio, and power [11], there can also be additional units for generating power or sensing its location.

One of the primary objectives of a node is to preserve as much battery power as possible. Therefore, the hardware often has limited capabilities, restricting both the radio communication range and the processing power [12]. An example of a node that we use in this thesis is TMote Sky [13], which has a CPU speed of 8MHz, 10KB RAM, and 48KB of flash storage. There are several ways to measure the lifetime of a WSN, and a typical measurement is the time until the first node has used up all its energy reserves and powers down. However, other measurements include the time until at least one node loses connection with the network, time until the network loses connection with a base station, or when a percentage of nodes have powered down.

Since the nodes are deployed in an ad hoc manner and use restricting hardware, an essential aspect of any protocol running on a WSN is fault tolerance. The protocol has to be able to handle a variety of failures such as packet loss due to interference or transmission error, node failures due to battery depletion, or other transient faults [2]. Moreover, in multi-hop networks that require forwarders, packets pass through several nodes, increasing the probability for these failures to occur before

reaching the intended destination. There are two primary categories for achieving fault tolerance in WSN protocols: retransmission and redundancy [2]. In a protocol based on retransmission, each node waits for an acknowledgement of each packet it sends; if it receives none, the node will retransmit the packet. A protocol based on redundancy, on the other hand, includes extra bits in the packet which are used to detect, and sometimes even correct, faulty bits in any packets received.

2.2 Chaos

The Chaos protocol is the first WSN protocol that supports native all-to-all communication without sequential phases for collection, processing, and dissemination of information [3]. Traditionally, every node in the network schedules these phases at the same time, even if some nodes do not require it. In Chaos, on the other hand, every node independently decides what action to perform (transmit or receive data). However, all nodes perform one of these actions at the same time to enable them to communicate with each other without a central coordinator; this is called synchronous transmissions.

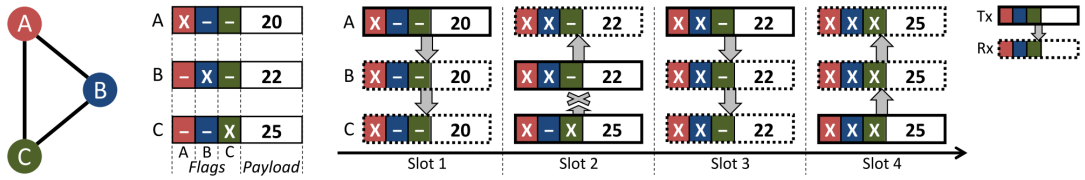
The smallest time component in the Chaos protocol is a *slot*, in which every node performs one of two actions: transmit or receive. At the end of a slot, every node decides what their action will be in the next slot. A node will transmit in the subsequent slot if it receives a packet with less or more information than itself. In the first case, it tries to spread its information to the node which sent a packet with less information. In the other case, a node merges the new information with its local data and then transmit in the next slot. A node will listen in the next slot if one of three scenarios happened in the previous slot: if it was transmitting, if it did not receive a packet, or if it received a packet but the packet did not contain any new information.

Next, Chaos combines multiple consecutive slots to create a *round*. During one round, Chaos executes one application. For example, the Max application in which all nodes spread their ID across the network and the merge operator executed on packet reception returns the maximum between the received number and a nodes local state. The number of slots in a round depends on the network size, a network with more nodes requires more slots.

2.2.1 Synchronous Transmission

Synchronous transmissions mean that all nodes execute rounds and slots at the same time. In Fig. 2.1 time progresses towards the right, and we see that all slots are aligned. This transmission scheme enables Chaos to benefit from two physical phenomena:

The capture effect occurs when packets are colliding (received at the same time by one node). If one signal is at least 3dBm stronger and the timing between the



(a) Nodes A, B, and C are in communication range of each other.

(b) All nodes prepare a packet with the proposed number and their own bit set in the flags.

(c) Node A initiates Chaos by sending its prepared packet in slot 1. B and C merge their own with A's data by taking the larger number and setting the bit of A. Both transmit synchronously in slot 2. A receives from B due to capture, merges data and manipulates the flags, and sends in slot 3. B does not send in slot 4 since it has nothing new to tell. Chaos completes in slot 4 when all nodes are aware of the maximum.

Figure 2.1: *An example of a Chaos round were three nodes reach consensus on the maximum of three proposed values [3].*

packets are within a certain threshold, a node will correctly decode the stronger signal, and ignore the weaker one [14]. Due to this effect, if a node receives two transmissions at the same time, one of them will, with high probability, be decoded correctly.

Constructive interference occurs when multiple nodes are transmitting the same packet at the same time. If that happens, the resulting signal strength is increased, which increases the probability that the packet will be received correctly by listening nodes.

2.2.2 The Initiator

In Chaos, a preconfigured node called *the initiator* needs to be present, node *A* in Fig. 2.1. The initiator is the node which initiates communication in a round. It also determines parameters such as what application to run and timing parameters for slots and rounds. All other nodes exclusively listen for packets until one is received which has propagated from the initiator. Both when the network boots up, and at the beginning of every round.

2.2.3 Association

When the network is booting, the nodes perform a process called association. During this time all nodes, except for the initiator, constantly listens while randomly choosing the radio channel, until they receive a packet with which they can synchronise. From this packet, a node learns timing parameters, the next application, slot, and round number. A node will also enter association when it has not received a valid packet for a configurable number of rounds, depending on the application that is running in the network.

2.2.4 Flags Field

The *flags field* is used by a node to keep track of which nodes have contributed during a round. It is a list of boolean values and can be seen in Fig. 2.1, marked as *flags*. Each node in the network corresponds to an index in the list. At the start of a round, the initiator sets the value at its index to true. As packets propagate through the network, other nodes also set the value of their indices to true. Once a node has merged a received packet which sets all values in the flags field to true it knows that all nodes in the network have participated and the information it has is a complete view from the network.

The Chaos protocol has a limitation to its scalability due to the flags field. Since every node requires a 1-bit space in the flags field, if there is an upper limit on the packet size imposed by either the link-layer protocol or the nodes, the number of nodes in the network is limited by the maximum size of the packet.

2.2.5 Completion Flooding

Completion flooding is executed at the end of a round if a node notices that all flags in the packet are set to true. Once in this phase, the node broadcasts its completed packet every other slot for a configurable number of slots. Aggressively transmitting the packet causes neighbouring nodes to reach completion which causes them to enter the completion phase. Since all nodes in this phase broadcast the same packet, the signal benefits from constructive interference which makes the signal of the complete packet stronger causing the capture effect to apply more often.

2.2.6 Timeout Mechanism

To prevent premature termination a *timeout mechanism* triggers if a node has not received a packet for some number of slots. Premature termination is when some nodes do not have all the information, but the communication has stopped. The number of slots is selected randomly by each node from an interval. When a node reaches that timeout, the node transmits a packet. If nearby nodes see that a node near them has less information than they do, they will transmit in the next slot, and thus communication is restarted.

2.3 Agreement in the Air

As the original implementation of Chaos has much potential, further research conducted by Al Nahas et al. [4] resulted in the A^2 Synchrotron. A^2 is a layer that provides services to the application layer, as seen in Fig. 2.2. The Synchrotron kernel is a further development of Chaos. In this section we, First, describe the

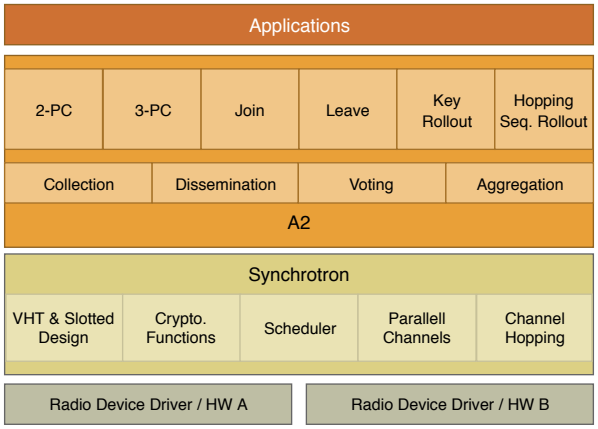


Figure 2.2: *The original A^2 architecture [4].*

scheduler in Synchrotron. Second, we cover the dynamic group membership provided by A^2's Join and Leave services. Last, we explain how Synchrotron achieves frequency agility with parallel channels and channel hopping. The remaining architecture description shown in Fig. 2.2 is explained in [4].

2.3.1 The Scheduler

The scheduler in A^2 allows the initiator to schedule multiple applications dynamically. During each round, the initiator sets a *next app* field in the packet to instruct the rest of the network which app should run in the next round. With the scheduler, a network can run multiple applications or services at different intervals, without any interference between applications. The scheduler also facilitates the scheduling of the dynamic group membership, scheduling the Join and Leave services as needed when nodes either request to join or disappear from the network.

2.3.2 Dynamic Group Membership

Dynamic Group Membership is implemented using the Join service, which allows the network to assign indices for nodes in the flags field dynamically. A node wanting to join the network sets the join flag in the packet header. Once the initiator receives a packet with a join flag, in say round r, it will in round $r+1$ tell all nodes to schedule the Join service for round $r+2$. The join round runs in two phases, collect and disseminate. We show an example of the Join service with three nodes in Fig. 2.3.

The Collect Phase is the first phase, where the network constructs a list in which nodes that want to join the network adds their node ID. The initiator will switch to the second phase (Slot 5 in Fig. 2.3) when it notices full participation of the network and does not record any changes to the join list for a couple of slots, or if the list is full.

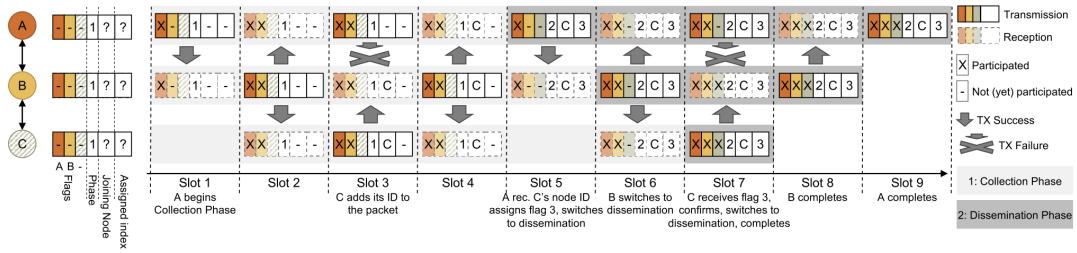


Figure 2.3: *The Join service exemplified using three nodes [4].*

The Disseminate Phase is the second phase, where the initiator assigns each new node an index in the flags field. The initiator then disseminates the join list with updated information about the joining nodes flag indices. Each joining node can now set the flag in their index as an acknowledgement (C sets its flag in Slot 7). In case a node misses the dissemination, it will request another join round and be assigned the same flag index again.

The Leave service removes nodes from the flags field if they do not participate for a set number of rounds. When it triggers, a join round is scheduled to disseminate the information about what nodes are present. The Leave service ensures that disappearing nodes does not hinder the completion phase.

2.3.3 Frequency Agility

In the presence of interference, the performance of A^2 degrades. Al Nahas et al. solves this by introducing frequency hopping and parallel channels.

Frequency hopping consists of each node switching which radio channel they use to transmit and receive data in unison in every slot. By switching channels, using the default hopping sequence from the IEEE 802.15.4e standard [15], the network avoids interference from other network traffic, which might occupy a single channel, allowing A^2 to exist side-by-side with other wireless systems.

Parallel Channels alleviates the issues caused by having a dense network with many nodes transmitting simultaneously in the same area. Many simultaneous transmissions on the same frequency cause interference, which damages the probability of packet capture. Synchrotron allows for a configurable number of channels from which nodes pick randomly in each slot to use for reception or transmission. The number of parallel channels require configuration and could damage the performance of a network if it is set too high or low.

2.4 Gossiping

Gossiping, presented by Demers et al. [16] in 1987, is an effective and straightforward method for disseminating updates throughout a large distributed system. Gossip algorithms are also called epidemic algorithms since they work similarly to how diseases spread, thus, gossiping borrows some terminology from epidemiology. For example, a node with new information is infective, and a node unaware of that update is susceptible to that node. Gossip algorithms spread updates using elementary actions called push and pull. One common problem is that nodes cannot know when the network has reached consensus; however, networks that use gossip protocols have eventual consistency.

During one gossiping round, a node either pushes or pulls updates from a uniformly random selected node. Both actions are required to infect all nodes effectively; the argument for this is intuitive: Consider a network with n number of nodes. A new update originating at node u is hard to find for all other nodes since the probability to select u is $\frac{1}{(n-1)}$. However, if u decides to use the push action, it is sure to infect another node. Moreover, as the number of infected nodes grows to more than $\frac{n}{2}$, the push action has less than 50% probability of infecting a susceptible node. However, the susceptible nodes have greater than 50% probability of selecting an infected node when they use the pull action.

Using a push and pull scheme to spread updates throughout a distributed system is effective. However, if the nodes in the system communicate using direct communication in every round, a gossiping algorithm cannot achieve both time- and communication-optimality [17], that is, completing in only $\mathcal{O}(log\, n)$ rounds using only $\mathcal{O}(n)$ messages. Karp et al. [17] investigate this problem, and they show that every algorithm that spreads a rumour in $\mathcal{O}(ln\, n)$ rounds needs $\omega(n)$ transmissions, where n is the number of nodes in the system.

In further developments of gossiping protocols, it is possible for nodes to only have a partial view of the network, called a group, as described by Eugster et al. [18]. Every message sent is *piggybacked* with information about what nodes the transmitter knows about, allowing a receiver to update its membership group dynamically, replacing some nodes in its group with some of the newly received ones to maintain an average group size. When a new node wants to join the network, it contacts either an arbitrary or dedicated node, which is called the bootstrapping node. The bootstrapping node first decides if it should save the new node to its group and then forwards the information about the new node to its group members. Each member then repeats the same process recursively up to a configured depth, which depends on the size of the group. The new node initialises its group with the bootstrapping node and then add nodes to its group dynamically using the information that is piggybacked in every packet.

Only maintaining a partial view of the network improves scalability as it, in larger networks, becomes infeasible for a node to maintain information about all nodes. In exchange for scalability, the algorithm loses reliability. The reason for this is that

the larger the view of the network a node has, the less the risk is for partitioning to occur [18].

2.5 Clustering

Clustering is the task of partitioning a network and electing nodes, called Cluster Heads (CH), which are responsible for a network-wide communication overlay [6]. All other nodes only communicate within their cluster. Each CH retains information from its cluster and forwards that to other CHs or a base station. A base station is not a sensor node; it is not battery powered and usually only collects data from other nodes. In some networks, there is no base station; the aim is instead to inform all nodes in the network of the calculated values. In either case, it is the job of the CH to make sure that the relevant parties get the final value.

A phenomenon that can occur when nodes or CHs are using a many-to-one communication pattern, such as when aggregating data to a base station, is the hot spot problem [19]. Nodes that are closer to the base station will need to handle more network traffic, and this can lead to them depleting their energy earlier than other nodes in the network. In the worst case, this can cause the network to become cut off from the base station entirely, completely disabling the network.

Clustering can be beneficial in several aspects; we list some objectives of clustering in Section 2.6. One of the most critical components of a WSN is its energy consumption and using the radio to transmit and receive data is generally the most expensive operation of a node [20]. Clustering has been shown to decrease the amount of communication, making clustering an excellent candidate to preserve energy in a WSN.

2.5.1 HEED

In this thesis, we base our work on the HEED algorithm [7], a probabilistic algorithm with the primary objective of saving energy. To this end, HEED uses residual node energy as its main parameter to elect CHs and intra-cluster communication cost as its second parameter. An example of intra-cluster communication cost is the number of nodes that have already joined a cluster. In HEED, the clustering algorithm executes at regular intervals ranging from seconds to hours depending on the application the network is running and how significant the increase in energy consumption is for CHs.

At the start of the clustering algorithm, each node sets an initial probability to, for example, $C_{prob} = 5\%$. Then, the node applies its proportion of residual energy as weight according to

$$CH_{prob} = C_{prob} * \frac{E_{residual}}{E_{max}}$$

to calculate its probability of becoming CH. Where $E_{residual}$ and E_{max} is the residual energy and the maximum possible energy respectively. The value of CH_{prob} is limited by a carefully selected lower bound, inversely proportional to the value of E_{max} to ensure that the algorithm terminates in $\mathcal{O}(1)$ time. At the end of every iteration of the algorithm, this probability is doubled, until the algorithm terminates or CH_{prob} reaches 1.

Furthermore, there are two stages for any node announcing itself as CH; a node first becomes a *tentative* CH until it either resigns or becomes a *final* CH. A node announces itself as a tentative CH with probability CH_{prob} if it has not heard from any other CH and becomes a final CH if $CH_{prob} = 1$. A CH resigns from being tentative if it finds another CH to join with lower cost than itself. Last, if a node reaches the end of the algorithm without hearing a final CH, it will announce itself as a final CH to ensure that CHs cover the whole network.

2.6 Clustering Objectives

There are two categories of clustering objectives, primary and secondary, and when designing a clustering algorithm achieving one or more of the primary objectives is often the goal. In contrast, the secondary objectives are usually not targeted explicitly in the algorithm design, but instead often indirectly achieved when clustering the network [5]. We list some common primary and secondary objectives in Table 2.1.

2.6.1 Primary Clustering Objectives

All the primary objectives are usually desired when implementing a clustering algorithm. However, it can be difficult to achieve all of them at the same time. There are often trade-offs between the objectives, and instead, it is common to only target a few of them.

Scalability is a consequence of clustering since the network is partitioned, making the network artificially less dense. Factors that need to be considered when designing

Table 2.1: *An overview of primary and secondary clustering objectives [5].*

Clustering Objectives	
Primary	**Secondary**
Scalability	Increased connectivity
Fault-tolerance	Reduced routing delay
Data aggregation/fusion	Collision avoidance
Load balancing	Using sleep schemes
Stabilised network topology	
Maximal network lifetime	

for scalability include network density and routing delays.

Fault tolerance is often a requirement for mission-critical applications. A clustering algorithm can be fault tolerant by periodically reclustering the network. If a node dies and some part of the network loses connectivity, a reclustering can reconnect the network again.

Data aggregation/fusion is the act of filtering out redundant data at some point before it reaches its destination. If the application a WSN is running is only interested in the average, or otherwise redundant data is sent, the CH can filter out that data and decrease the number of packets transmitted.

Load balancing is achieved by periodically changing the nodes that are CHs. Assuming that being a CH requires more battery, load balancing will increase the total lifetime of the network since the CH role is distributed among all nodes.

Stabilised network topology concerns the mobility of nodes. If a node moves and switches cluster, the CH can register this change and keep the network topology up to date.

Maximal network lifetime is a common objective often, in part or entirely, caused by the other objectives. For example, proper load balancing means avoiding early node deaths, and data aggregation means the network can filter unnecessary traffic. Both of these properties work towards an energy efficient cluster which implies an extended network lifetime.

2.6.2 Secondary Clustering Objectives

The secondary objectives are not usually a target when implementing clustering algorithms but rather achieved indirectly by targeting the primary objectives.

Increased connectivity is achieved since in a clustered network only the CHs need to be connected to each other. All other nodes only have to be connected to their cluster, which relaxes the requirement for a network to be considered fully connected.

Reduced routing delay is desired for some applications, packets need to arrive within a specified time limit; this can be a challenge in a widespread WSN. With clustering, the communication within a cluster can arrive faster since the clusters are smaller and closer together than the network as a whole.

Collision avoidance is achieved since transmission collisions may cause packet loss, which is energy wasted on doing nothing. To minimise packet loss, clusters can split their communication between different radio channels or different time slots.

Using sleep schemes can be beneficial since in some applications there is no need for the whole network to be active all the time. For example, some nodes might only

Table 2.2: *Cluster properties and their options.*

Cluster Properties	
Property	**Options**
Cluster size	Equal Unequal
Cluster count	Constant (preset) Variable
Intra-cluster communication	Single-hop Multi-hop
Inter-cluster communication	Single-hop Multi-hop

provide routing paths. In this case, all paths need not always be active. Letting some nodes sleep prolongs the network lifetime.

2.7 Clustering Technology Properties

There exist several ways of categorising the properties that define clustering algorithms. Afsar et al. [5] describe three main categories: Cluster properties, Cluster Head properties and Clustering process properties. Here, we summarise each category and list the properties within each of them.

2.7.1 Cluster Properties

The *cluster properties* summarised in Table 2.2 describe properties of the clusters the algorithm creates.

The size of clusters can be tailored to be either equal or unequal. In an equal clustering algorithm the algorithm will enforce an equal size on all clusters while in an unequal algorithm the clusters will vary in size according to some criteria, for example, distance to a base station; if a cluster is closer to a base station, the cluster size will be smaller.

The number of clusters can be either fixed or variable. When it is variable, it could be due to a probabilistic CH election process. A fixed number of clusters is usually a property that centralised algorithms have.

The communication distance can be either *single-hop* or *multi-hop* for both *intra-cluster* and *inter-cluster communication*. Inter-cluster requires multi-hop communication when the number of CHs are few and spread out since they will need help with forwarding to reach the other CHs. Intra-cluster communication needs to be multi-hop when there is a bound on the number of CHs since every node might not be able to reach a CH in a single hop.

Table 2.3: *Cluster head properties and their options.*

CH Properties	
Property	**Options**
Mobility	Mobile Stationary
Node types	Homogeneous Heterogeneous
Role	Relay Aggregation/Fusion

2.7.2 Cluster Head Properties

The cluster head properties listed in Table 2.3 define different behaviours for the cluster heads.

The mobility of a node is either mobile or stable. A mobile CH could induce frequent topology changes, which adds overhead.

The node type is either homogeneous or heterogeneous. A network with heterogeneous nodes means some nodes have additional resources. The additional resources could be a higher energy reservoir, increased computing power or increased transmitting range, which could make the node more suitable to be CH.

The role of a CH is either only to relay messages or to both relay messages and perform the same operations regular nodes do, such as collecting and aggregating data.

2.7.3 Clustering Process Properties

The clustering process describes the properties used to create a cluster, these properties are often more high level and describe the process as a whole. The clustering process properties are summarised in table Table 2.4.

The method used for clustering can either be distributed or centralised. A distributed algorithm has the potential to create clusters faster since each node can make local decisions. However, a centralised approach can create more optimised clusters since it has a global view of the network.

CH election can either be preset, random or attribute based. Preset is when the clustering is configured before the network is deployed. The opposite of a preset approach is a random approach, letting the nodes become cluster heads with a certain probability. Last, nodes may select CHs with a more complicated process using attributes such as remaining energy or the number of neighbours.

The algorithm complexity is either constant or variable. A constant complexity means setup time is not dependent on network size while a variable time algorithm

could converge on better choices for cluster heads.

The nature of the clustering process can either be proactive or reactive. Proactive means that it does not consider what application will run on it. In contrast, it can also be reactive meaning it tries to optimise the cluster for a specific application and the data flow required by it.

The dynamism of the algorithm can be either dynamic or static. In a dynamic approach, CHs are elected based on the current conditions of the network while a static approach would yield the same result regardless of when the algorithm is executed.

Table 2.4: *Clustering process properties and their options.*

Clustering Process	
Property	**Options**
Method	Distributed Centralised
CH election	Preset Random Attribute based
Algorithm Complexity	Constant Variable
Nature	Proactive Reactive
Dynamism	Dynamic Static
Objectives	See table 2.1

3

Related Work

There is a variety of research on how to increase scalability and energy efficiency in Wireless Sensor Networks. In this section, we present a few clustering algorithms to highlight their key ideas, similarities, and differences. We also present research that improves the scalability of the Chaos protocol using estimation vectors. Finally, we end with a discussion and comparison to our clustering process.

3.1 UHEED

In Section 2.5.1, we present HEED, a probabilistic, equal clustering algorithm. Unequal HEED (UHEED) [21], presented in 2012, builds on HEED and makes the algorithm unequal to solve the hot spot problem. This is the only difference to the HEED algorithm.

UHEED creates smaller clusters closer to the base station to mitigate the hot spot problem. The idea is that CHs that are closer to the base station will need to route more traffic from other clusters and thus have a higher energy consumption than a CH further away from the base station.

The clusters get progressively smaller by defining a *competition radius*, which is the area a node considers when choosing CH. UHEED uses a formula to calculate the competition radius, developed by Lie et al. [22], that uses the maximum distance between nodes and the distance to the base station as parameters. Evaluations of UHEED showed that it has an equal or better performance in virtually every case when compared to state of the art clustering algorithms [21].

3.2 BCDCP

Base-station Controlled Dynamic Clustering Protocol [23] is a centralised and dynamic clustering protocol presented in 2005. The clusters are created by a base station, with much higher energy levels and better computational power than the nodes, which enables the algorithm to leverage global properties of the network to create efficient clusters.

BCDCP uses a four-step process, executed iteratively until the target number of clusters is reached. The process begins by collecting a set $\mathbb{S}$ of nodes with higher than average energy levels, which are candidate CHs. Then execute the following steps.

1. Choose two nodes s_1 and s_2 from $\mathbb{S}$ that have a maximum separation distance.
2. Group all remaining nodes in the current cluster (this includes all nodes in the first iteration) with s_1 or s_2, depending on which is closest.
3. Balance the two groups so that they are approximately equal; this forms the two subclusters.
4. Split $\mathbb{S}$ into subsets $\mathbb{S}_1$ and $\mathbb{S}_2$ according to the subcluster groupings created in step 3.

Performing these steps produces a set of clusters which are distributed uniformly throughout the network, where the communication cost within a cluster is minimised [23].

BCDCP is evaluated by simulations conducted in Matlab, and compared to several other clustering algorithms. From these simulations, Muruganathan et al. conclude that BCDCP achieves better performance regarding energy usage, and first node death compared to other state of the art clustering algorithms. However, they note that BCDCP is most effective for networks spread over a considerable distance, and the benefit of using BCDCP is less for smaller networks.

3.3 LEACH

A reactive clustering protocol developed in 2002 is *Low-Energy Adaptive Clustering Hierarchy* (LEACH) [24]. Heinzelman et al. designed LEACH for applications that measure something in the environment where the interesting results are not the individual values but some computation using those values. The clustering algorithm in LEACH is distributed and probabilistic with the primary objective of saving energy.

The LEACH algorithm is divided into rounds. Rounds in LEACH are conceptually different to rounds in Chaos. Each round begins with a setup phase that forms the clusters, followed by a steady state phase where nodes collect data and CHs aggregate and forward that data to the base station. In the setup phase, each node i has a probability $P_i(t)$ at time t to announce themselves as CH. This probability depends on a number k, which is the target number of clusters. In a network that contains N nodes, the probability $P_i(t)$ is chosen so that the following formula holds:

$$E[\#CH] = \sum_{i=1}^{N} P_i(t) * 1 = k$$

Also, to maximise network lifetime, each node takes on equal responsibility of becoming a CH. Thus, if there are N nodes in the network, each node will, on average, be CH every N/k rounds.

The nodes which have elected themselves as CH need to inform the rest of the network of this decision. They do this by sending out an advertisement message containing their ID. All other nodes will listen for these messages and map each CH with the energy required to transmit a packet to it. The node then picks the CH that requires the lowest transmission energy.

Leach is evaluated and compared to other clustering algorithms in simulations. The results show that LEACH achieves better energy usage and data throughput when compared to other distributed clustering algorithms as well as static clustering of the network. However, it is outperformed by LEACH-C, a centralised version of LEACH.

3.4 Extreme Chaos

Chronopoulos [25] address the scaling bound on the number of participating nodes described in Section 2.3, and introduce flow control to limit the high number of initial transmitters in a Chaos round. The flags field requires at least one bit per node, but together with the additional information required in the packet (synchronisation, error-detection and payload), the number of possible participating nodes is less than a thousand [3]. Chronopoulos also discuss how the flags field affects the latency of transmissions. They note that increasing the network size by a factor of 10 increases the delay in the network by a factor of 100 [25]. Last, they note that the flags field also imposes an overhead on the energy consumption when it is transmitted.

3.4.1 Estimation Vector

To replace the flags field, Chronopoulos propose to use an estimation vector, which is the most prominent part of the new protocol. Just as the flags field, the estimation vector is used for keeping track of how much of the network has contributed to the aggregate. Chronopoulos argue that extreme values spread quickly throughout the network and therefore nodes do not need to make sure they have communicated with every other node. As such nodes only need to set a percentage of the flags. However, this does not reduce the size of the flags field. The new vector estimates the number of nodes which has contributed to the aggregate, and by also estimating the network size the nodes infer the contribution ratio.

3.4.2 Flow Control

An additional issue addressed by Chronopoulos is the high number of transmitters at the beginning of a round, which is caused by the transmission policy of Chaos. Nodes only transmit when they receive new information, however, at the beginning of a round, there is a high probability that most nodes will receive new information. Too many transmitters cause interference and slow packet propagation since nodes need to retransmit packets. The mechanism to counter this flooding is called flow control, and its purpose is to prevent congestion from occurring by applying an *exponentially decaying back-off probability* [25]. The back-off is a node deciding, according to some probability, to not send during a slot in which it intended to send. The exponential decay is referring to the probability decreasing at an exponential rate throughout the round. Decreasing the congestion in early slots will speed up the progress of an aggregate process which enables a round to finish earlier.

To not have adverse effects the parameters used for calculating the next probability for back-off requires careful testing. Having a too low probability of back-off would mean the initial flooding is not constrained enough. In contrast, a too high probability means dropping later transmissions which would hinder the aggregation process. Additionally, flow control can affect large and small networks differently. In a large and sparse network, an extended period with initial high flooding can occur. Chronopoulos argue that this is because nodes far from the initiator join the network in later slots. However, Chronopoulos does not consider large multi-hop networks since their primary focus is to optimise Chaos for local neighbourhoods.

3.5 Discussion

In this chapter, we presented three different clustering algorithms and Extreme Chaos. We discuss and compare the clustering algorithms to the HEED algorithm, and contrast the work on estimation vectors and flow control to clustering.

The clustering algorithms presented here all exhibit different characteristics, and restrict the network in different ways to accomplish their goals or solve specific problems. UHEED, which is further work on the HEED algorithm, primarily tries to solve the hot spot problem. However, since the A^2 system does not currently have any concept of base stations for any of its applications, the hot spot problem does not apply in this thesis.

LEACH, on the other hand, is similar to HEED since it is also probabilistic and wants to maximise network lifetime. However, it assumes that the data the application running on the network transmits can be efficiently aggregated. LEACH also evenly distributes the CH role between all nodes in the network, which can be an undesirable property since, for example, nodes at the edges of the network might be unsuitable CHs.

Furthermore, BCDCP is a centralised algorithm, and as such, it can leverage both

a global view of the network and use more resources, since it is assumed that the node responsible for the clustering has more computational power. No such node is assumed to exist by the A^2 system, which makes this and other centralised algorithms unsuitable. It could be feasible to employ this protocol on a typical node, however, in that case, there is a high risk that the network would not scale well.

Last, Extreme Chaos also addresses the scalability issues that the A^2 system exhibits. However, the approach is fundamentally different to clustering. Both the estimation vector and flow control are probabilistic mechanisms that affect Chaos in every round. A probabilistic clustering algorithm, on the other hand, only exhibits uncertainties during the clustering process, then, during normal operation of the Chaos protocol, the process is strictly deterministic.

4

Design of the Clustering Process

In this chapter, we explain the design behind our clustering implementation and put its objectives and properties into context. First, we provide an overview of the clustering process. Second, we go into detail about the Clustering service, explain how we design for scalability, and explain the CH election algorithm. Third, we explain of how nodes pick cluster heads from the elected set of CHs. Fourth, we describe a Demote service which runs after the Clustering service to remove suboptimal clusters. Fifth, we explain how communication works between CHs and how clusters avoid interfering with each other's communication. Last, we discuss decisions we make regarding the design and the effect of those decisions.

4.1 Clustering Process Overview

To increase the scalability and lower the energy consumption of A^2, we design and implement a clustering process based on the HEED clustering algorithm [7]. The process is responsible for clustering the network, appointing nodes as CHs, and letting nodes join clusters.

Our clustering process consists of four distinct phases. First, the Clustering service runs, and every node does three things simultaneously: While *(i)* learning about the network topology by collecting statistics about received packets, they also *(ii)* try to become CH, and if they hear another node announcing itself as CH within the configured competition radius they *(iii)* select an appropriate cluster to join. Second, each elected CH runs the Join service from A^2 inside their cluster. Third, all clusters which deem themselves too small disband, and all nodes in these clusters select other clusters. Fourth, each CH rerun the Join service to determine the final set of nodes in their cluster. This process is repeated at a certain round interval to accommodate for changing battery levels for nodes in the network.

When the clustering process is complete, the network executes some application, in our case, the Max application. At this point, we split the rounds in A^2 into *cluster* and *cluster head* rounds. In cluster rounds, each cluster will separately execute an application to completion. In CH rounds, the CHs execute the same application again, but they remember the information they got in the previous cluster round. What that information is, depends on the application; in the Max application's case,

it is the maximal value found in each cluster. The process of alternating between *cluster* and *cluster head* repeats until the clustering process starts again.

4.2 The Clustering Service

The Clustering service is responsible for clustering the network, and it needs to run first to enable a clustered communication medium. In this section, we describe the design of the Clustering service, how we design for scalability, how the clusters are formed, and the parameters that control the Clustering service.

4.2.1 Designing for Scalability

Because our primary goal is to increase the scalability of the A^2 system, a challenge is the packet size restriction of 127 bytes, imposed by the 802.4.15 standard [15]. If an application adds data to the packet and if that data grows linearly with the number of nodes in the network, then the application will quickly reach the packet size restriction. Because the flags field is part of all packets and does grow linearly, we want to prevent it from growing too large. Consequently, we need to cluster the network before running the Join service. Therefore, we design the Clustering service to run without completion flags. Running the Join service after the Clustering service puts the scaling restriction of the flags field locally in each cluster. However, the network cannot perform completion flooding or early turnoff during the Clustering service, as completion is calculated from the flags field. Nonetheless, it is vital that all nodes learn of all CHs since nodes use that information to maintain a flags field for CH rounds.

4.2.2 Transmission Policy with Gossiping

Without completion flags, nodes cannot determine when consistency has been reached. Therefore, we inspire the transmission policy of the Clustering service from gossiping protocols, to benefit from the high probability of gossiping to disseminate updates successfully. The transmission policy for the Clustering service is that a node should transmit in the next slot if it received a list that contains a CH which the node did not know about, or the list is missing a CH the node does know about; otherwise a node will try to receive data.

4.2.3 Election of Cluster Heads

The CH election algorithm is based on the HEED algorithm [7], it is probabilistic and attribute based. The Clustering service is executed over several rounds, and the algorithm is divided into two phases: the main phase, shown in Algorithm 1,

and the final phase, shown in Algorithm 2. The main phase probabilistically elects a set of CHs based on their residual energy, nodes with more residual energy has a higher chance of becoming a CH. In this phase, every node also chooses a cluster to join. The final phase ensures that CHs cover the whole network.

In every round, all nodes have set a probability to announce themselves as CH, and they double that probability at the end of every round until it reaches 1; at that point, each node has either heard another CH they can join or elected themselves to be CH. The algorithm ensures that a stable set of CHs is elected in constant time [7].

Nodes calculate their initial probability (CH_{prob}), to announce itself as CH, according to

$$CH_{prob} = C_{prob} * \frac{E_{residual}}{E_{max}} \tag{4.1}$$

taken from HEED [7]. C_{prob} is an initial probability which is used to get the process started; it does not have any impact on the final number of cluster heads [7]. $E_{residual}$ is the residual energy of the node and E_{max} is the maximum energy of the node. Furthermore, CH_prob is bounded by p_{min}, a constant lower bound inversely proportional to E_{max}, which ensures that CH_{prob} reaches 1 in constant time.

The number of consecutive Clustering service rounds required to form stable clusters differs between networks. It depends on the network topology, the diameter of the network, and the lower bound on the initial probability. For example, a sparse network requires more rounds than a dense network.

There are two states for a CH in the HEED algorithm: tentative and final. A CH is tentative until its probability of becoming CH reaches 1, then it becomes final. This distinction is made to prune out nodes which have lower residual energy. Since the Clustering service runs for a fixed number of rounds, if a CH is still tentative when the Clustering service begins the final phase and it can find another cluster to join, it will be demoted and join that cluster.

Furthermore, we control the number of nodes that can become CHs in two ways. As can be seen in Algorithm 1 at Line 3, a node is only allowed to announce itself as CH with probability CH_{prob} if either of two things holds. First, taken from HEED, a node must not have seen another CH within their competition radius (measured in hops). Second, further defined by us, a node may still elect itself if the $neighbourRatio$, that is, the number of neighbours divided by the number of CHs within a nodes competition radius, is larger than the parameter *nodes per cluster ratio*. We explain this parameter in more detail in Section 6.2, but its primary purpose is to make dense networks sparser, thus decreasing interference.

However, if two nodes decide to become CH in the same round, then both of them will begin to announce themselves at the start of the next round. This may cause two nodes within each others competition radii to become CHs; to decrease the risk of this, nodes will wait until a random slot during the first half of the next round

Algorithm 1 The repeat phase adaptation of the HEED algorithm. It shows how a node elects to announce itself as cluster head. The algorithm is adapted for A^2 in two ways. We utilise our parameter *nodesPerClusterRatio* at Line 3 and set the announcement slot at Line 15.

```
1:  procedure HEED_REPEAT
2:      if prevCH_prob ≤ 1 then
3:          if validCHList ≠ ∅ or neighbourRatio > nodesPerClusterRatio then
4:              myCH ← pickBestCH(validCHList)
5:              if myCH = myNodeID then
6:                  if CH_prob = 1.0 then
7:                      CHState ← FINAL
8:                  else
9:                      CHState ← TENTATIVE
10:                 end if
11:             end if
12:         else if CH_prob = 1 then
13:             CHState ← FINAL
14:         else if random(0, 1) ≥ CH_prob then
15:             announcementSlot ← random(1, maxAnnouncementSlot)
16:         end if
17:         prevCH_prob ← CH_prob
18:         CH_prob ← min(2CH_prob, 1)
19:     end if
20: end procedure
```

before they begin to announce themselves. This gives another newly elected CH a chance to hear another nearby CH, which they can join before they announce themselves. We limit this random back off to the first half of a round to ensure that the CHs do not announce themselves too late not to be noticed by other CHs.

4.2.4 The Final Phase

HEED's final phase executes during the last three rounds of the Clustering service. It performs two important functions, it demotes tentative CHs and ensures that CHs cover the whole network. We show the pseudo code for the final phase in Algorithm 2.

The final phase serves the same purpose in both our algorithm and HEED, but CHs are tentative for different reasons. In HEED, the only reason a CH is tentative in the final phase is if it announced itself as CH and then found another CH with a lower cost to join, according to a cost function. In HEED, the cost function is based on the communication cost between nodes and the CHs [7]. However, in our algorithm, a CH is tentative in the final phase if it does not have enough energy to reach a probability of 1.0 before the final phase begins. Additionally, the final phase ensures that the whole network is covered. As can be seen in Algorithm 2, if a node does

Algorithm 2 The final phase of the clustering algorithm. The only edit from the original HEED final phase is the usage of $pickBestCH$, which is our definition of the cost function that the HEED algorithm requires.

```
1: procedure HEED_FINAL_PHASE
2:     if CHState ≠ FINAL then
3:         if CHList ≠ ∅ then
4:             myCH ← pickBestCH(CHList)
5:             CHState ← NOT_CLUSTER_HEAD
6:         else
7:             CHState ← FINAL
8:         end if
9:     end if
10: end procedure
```

not consider any CH to be valid, i.e. exist within the competition radius, it will announce itself as CH; this guarantees that cluster heads cover the whole network.

4.2.5 Configuration Parameters

In this section, we describe the parameters that change the behaviour of the Clustering service: competition radius, minimum cluster size, and nodes per cluster ratio. We evaluate the effect of changing these parameters for different network topologies in Chapter 6.

Competition radius, measured in hop count, is a measurement of how far away a node can be from a CH and still join that CH. In a small and dense network, a small competition radius will make the network less dense by partitioning it into many clusters and thus lowering the number of packet collisions. On the other hand, in a large and sparse network, a high competition radius will keep the number of cluster heads low to ensure that the clusters do not get too small.

Minimum cluster size is the number of nodes, including the CH that has to be a part of the cluster for it to be considered valid. If a cluster has few nodes, one of the three following scenarios is likely to have occurred. First, the network could have been small or sparse, clustering such a network makes it even more sparse which could damage reliability and stability. Second, the CH could be poorly located, for example, close to the edge of the network or far away from other nodes. Third, the CH could be close to another CH that most of the nodes around them chose to join instead. In both the second and the third case our algorithm elected a bad CH which can be pruned relatively easily when the clustering process is complete. In the first scenario, it is hard for small and sparse networks to benefit from clustering.

Nodes per cluster ratio, tries to enforce a maximum cluster size to make a dense network artificially sparser. It is not a strict limit on the number of nodes in a cluster. Instead, it is a limit on the number of CHs that can announce themselves

depending on the number of neighbours they have. A node with more neighbours than *nodes per cluster ratio* will still be able to announce itself even if it has heard from another CH within its competition radius, dividing the neighbours between itself and other CHs, which makes the network less dense.

4.3 Joining Clusters

The primary objective of a node is to choose a cluster head that requires the least cost to join. To do this, we define a function that tries to find the CH that is closest to that node. Since a typical WSN is an ad hoc network, we have to gather information about the network while our Clustering service is running. Each node keeps track of two metrics to estimate how close other nodes are: The number of packets they receive from their neighbours (nodes that can be reached in a single hop) and the number of hops required to reach each CH. A node only considers CHs that have a hop count smaller than the competition radius and then chose the CH that they received the most packets from if there exist several valid CHs. We show the pseudo code for this process in Algorithm 3.

Algorithm 3 Our cost function for picking the best cluster head.

```
1: procedure pickBestCH
2:     for all CH ∈ CHList do
3:         if CH.hopCount ≤ competitionRadius then
4:             if CH.receivedPackets = max(CHList.receivedPackets) then
5:                 chosenCH ← CH
6:             end if
7:         end if
8:     end for
9:     return chosenCH
10: end procedure
```

4.4 Demotion of Cluster Heads

When a cluster head is demoted it becomes a normal node and all other nodes that have joined its cluster, including itself, joins another cluster. There are two ways a CH can be demoted: CHs are automatically demoted at the beginning of the Clustering service and if a node announces itself as CH, but fewer nodes than *minimum cluster size* join its cluster, it will also demote itself.

The Clustering service is designed to be executed periodically by the network to adapt to changes, such as nodes dying or nodes depleting their energy faster than others. To facilitate this, all previous CHs are demoted when the Clustering service starts. The Clustering service does not take into account previous clusterings of

the network and does not rely on data collected when the network is executing applications.

Additionally, the demote service is designed to be run after the consistent group membership protocol, provided by A^2 [4], has converged on a result for each cluster. The purpose of the demote service is to enforce a *minimum cluster size*. At this point, each CH knows how many nodes have joined their cluster. If the number of nodes in a cluster falls below the threshold *minimum cluster size* the CH will announce itself as demoted.

4.5 Communication

There are two separate instances of communication happening in a clustered network: intra-cluster and inter-cluster communication. The communication works similarly to the original A^2 design with some modifications.

4.5.1 Intra-cluster

The intra-cluster communication happens within a cluster, during this phase, the CH acts as the initiator for its cluster. We use the channel hopping functionality provided by the A^2 Synchrotron [4] to make all nodes jump between radio channels in a sequence to minimise foreign interference. However, we also split all clusters into different channels to minimise interference between clusters. To accomplish this, each cluster applies a unique offset to their channel hopping sequence. The number of clusters that we can split in this way is limited by the number of available radio channels, which currently is 16. If there exist more clusters than channels, the clusters will overlap. However, the communication will still work since clusters ignore packets from other clusters.

4.5.2 Inter-cluster

The inter-cluster communication takes place during CH rounds in which the CHs propose the aggregate value that its cluster agreed upon in a previous cluster round. All non-CH nodes act as forwarders and do not propose any values of their own, they only forward packets according to the transmission policy. Their participation is not counted when the completion of the network is calculated. Forwarders are required since our process can only guarantee an upper bound of 2 on the hop count between cluster heads with the lowest competition radius.

Table 4.1: *Our primary and secondary clustering objectives.*

Clustering Objectives
Scalability
Maximal network lifetime
Load balancing
Collision avoidance
Increased connectivity

Table 4.2: *Our properties related to clusters, cluster heads and the clustering process.*

Cluster Properties	Cluster Head Properties
Unequal cluster size	Stationary nodes
Variable cluster count	Homogeneous nodes
Single/multi-hop intra-cluster communication	CH takes on normal duties
Multi-hop inter-cluster communication	

Clustering Process Properties	
Distributed clustering process	Proactive nature
Attribute based CH election	Dynamic clustering
Constant algorithm complexity	

4.6 Clustering Objectives and Properties

In this section, we list the clustering objectives we design for and motivate why we make those choices. We also classify our algorithm according to the cluster, cluster head, and clustering process properties presented in Section 2.7.

In Table 4.1 we list the primary and secondary objectives for our clustering design. Scalability is the first objective since both Chaos and A^2 have a restriction on the maximum number of nodes that can participate in the network [3]. Clustering improves scalability by partitioning the network into smaller subsets. Our second primary objective is maximal network lifetime, which is a common challenge in WSNs [5, 12] and clustering increases the efficiency of a network which directly affects its lifetime. Finally, load balancing is our last primary objective. Load balancing is achieved by moving the CH role between different nodes, which increases the time until the first node death. Prolonging the time until the first node death directly increases the lifetime of the network.

Additionally, we have two secondary objectives: Collision avoidance and increased connectivity. Collision avoidance occurs because the clusters communicate on different radio channels, which reduces interference. Furthermore, we gain increased connectivity since the requirement for the network to be considered connected is weakened. That is, each node only needs to have a connection with its cluster, and a CH only requires a connection with its cluster and the other CHs.

Furthermore, our clustering design fulfils several properties, listed in Table 4.2. First, we have unequal cluster sizes; we do not enforce that all clusters need to be equal in size. However, we enforce a minimum cluster size by demoting CHs if they have less than a certain number of followers. We also enforce a ratio for the number of nodes per cluster, but this parameter depends on network density. The second parameter is variable cluster count, the number of clusters depends on many variables such as network topology and competition radius, also we use a probabilistic approach. Third, if the intra-cluster communication is single or multi-hop depends on the competition radius. If the competition radius is one, nodes have a single-hop communication with their cluster head. However, it is possible for two nodes in the same cluster to have multi-hop communication. For example, two nodes might require the CH to route communication between them. Finally, inter-cluster communication is multi-hop. CHs communicate normally but require non-CH nodes to forward their messages during CH rounds.

Moreover, we have cluster head properties. First, our design focus on stationary nodes. Second, all nodes are homogeneous. However, if we deploy a node with more resources (such as an improved battery) with our process, it will have an advantage in the election process since nodes with a higher energy level are more likely to become CHs. Finally, the role of a CH is to perform the same work as ordinary nodes in addition to the work required by a CH.

Finally, we have clustering process properties. First, each node executes the clustering process in a distributed fashion; this leads to a more scalable network compared to a centralised process. Second, our election process is attribute based and takes into account the residual energy of nodes to elect CHs, to work towards maximising the network lifetime. Third, the complexity of the election algorithm is constant; this is because we use local decisions to elect CHs. Fourth, the nature of the clustering process is proactive to accommodate A^2's ability to schedule multiple applications in the network. Lastly, our clustering process is dynamic since it is attribute based; the process takes into account the current condition of the network, such as the residual energy of the nodes.

4.7 Discussion

In this section, we present our discussion on the design we describe in this chapter. We discuss how the transmission policy of our clustering process is similar to a gossiping protocol, as well as some decisions we make regarding the design, and the impact of these decisions.

4.7.1 Comparing the Clustering Service's Communication to Gossiping

Because the Clustering service does not have a flags field, nodes cannot know when they have a complete picture of the CH list. However, we argue that our design of the transmission policy of the Clustering service is sufficiently similar to a gossiping protocol to benefit from the high success rate and eventual consistency which gossiping protocols posses. To compare with a gossiping protocol, we argue for similarities between our transmission policy and the push and pull actions.

Every transmission from a node is always a push action, but it could simultaneously be a pull action if the node is susceptible to one or more of its neighbours. It is a push action since a node includes all of the CHs it knows in each packet it sends. However, there is no guarantee that the node is infective to any of the receiving nodes since they might already know all of its information. Furthermore, the transmission could also be a pull action since a receiving node might have knowledge which the transmitter does not have. The receiver will then in the next slot perform a push action, and the previous transmitter will attempt to receive the push.

However, a difference to traditional gossiping is that the push and pull actions are not targeted at a single node. Instead, a transmission is often received by many neighbouring nodes, because of the wireless communication medium. The concept of pushing updates to multiple nodes at the same time is called multi-casting, and only being able to push updates to a subgroup, such as the neighbours of a node, is called subgroup gossiping. While subgroup gossiping can be restricting since updates need to propagate through the network via neighbouring nodes, multi-casting is beneficial compared to peer-to-peer updates since it could allow an update to spread faster. Furthermore, research into multi-cast subgroup gossiping has shown that there exist good lower bounds on message propagation, which depend on the underlying graph [26].

Consequently, since we can enforce that there always exists at least one update which only the initiator knows about at the start of every round. Then the transmission policy will ensure that the update triggers communication throughout the network and nodes with updates of their own will have an opportunity to push their information which propagates throughout the network due to the transmission policy. From these arguments, we conclude that the Clustering service is very similar to a gossip protocol. This ensures that the Clustering service benefits from a gossip protocols high chance of disseminating updates to the network and that it has eventual consistency, even without completion flags.

4.7.2 Limiting the Number of Cluster Heads

In our design, we randomise the slot in which a node starts announcing itself as CH to limit the number of CHs that are neighbours to each other. Because nodes double the probability to announce themselves as CH every round, we can go from

a relatively low probability to a high probability in the span of one round. For example, a node which has $CH_{Prob} = 35\%$ in a round, will have $CH_{Prob} = 70\%$ in the next round, which could lead to many nodes announcing themselves at the same time.

We cannot guarantee that this mechanism fixes the problem, it relies on two things to work; that the two nodes that announced themselves in the same round randomised two numbers sufficiently far apart from each other and that successful communication happens between the nodes between those two slots. Additionally, the Clustering service does not take into consideration which node would be a better CH, which means that this mechanism could cancel election of nodes better suited to be CHs. However, since this mechanism is only used for nodes which are neighbours or close to neighbours, depending on the competition radius, the CHs that are removed should be relatively similar.

4.7.3 The Final Phase

The final phase of the clustering process almost identical to the final phase in HEED. However, there is one difference, CHs are tentative if they have too low residual energy. This is because we do not model the cost for a node to be CH, which means a node cannot tell if it would be better for it to demote itself and join another CH instead.

Furthermore, there is a drawback to the final phase which can happen if either the parameters to the Clustering service are poorly configured, or we are applying clustering to a network not suited for clustering, such as very sparse networks. For example, if the Clustering service is not scheduled for enough rounds, such that no CH can reach $CH_{prob} = 1.0$, then every node in the network will consider itself uncovered in the final phase, and announce themselves as CHs. Having a network where all nodes are CHs is comparable to having a network with no clustering, which will only waste energy and time since the network will still interleave CH rounds with cluster rounds.

4.7.4 The Cost Function

Our cost function tries to find the closest CH by counting the number of packets received from all nodes. All nodes will choose the CH they have received the most packets from.

We argue that the number of received packets from a node is a reasonable approximation of the distance to that node, due to the capture effect [14]. The signal strength decreases as the distance increases, and a difference in signal strength is integral for the capture effect to apply. Consequently, nodes that are closer to each other will more often correctly decode each other's packets than nodes that are far away.

However, since we can only count packets from neighbours, this cost function does not work when the competition radius is higher than one. Since no direct transmit can be made from a node that has a hop count two or higher, nodes which do not have a CH as their neighbour will have to choose a CH at random from the closest CHs in the network. If the node has a bad connection to that cluster, it will eventually trigger a resynchronisation and drop out of it.

4.7.5 Separating Cluster Communication

Our design separates intra- and inter-cluster communication with cluster and CH rounds respectively; this is convenient as it enables scheduling of cluster rounds more often than CH rounds. A usage of this is to let the CHs aggregate data from its cluster during multiple rounds before a single CH round is scheduled in which all the aggregated data could be forwarded to other CHs or a base station. Another reason for having CH rounds and cluster rounds is to only have one type of communication for each cluster during a single round, either inter- or intra-cluster.

In contrast, it could be possible to switch between intra- and inter-cluster communication within a single round. However, doing so bring the challenge of keeping the network synchronised when we switch from many initiators to one; this is a challenge since each cluster may require differently long before switching to inter-cluster communication, and would thus require that some clusters sleep until they reach a predetermined slot.

4.7.6 Clustering Risks A^2's Fault Tolerance

Since clustering partitions the network, it interferes with some existing fault tolerance features built into the A^2 system. The A^2 system depends on redundant messages and generating constructive interference to make all nodes reach consensus during a round. Clustering, as a technique, does not involve redundant messages, constructive interference or the capture effect. A risk when clustering a network is to make it too sparse for the A^2 system to succeed. However, as observed by Chronopoulos [25], the number of messages sent in the A^2 system can be too high, especially at the start of a round. Chronopoulos implemented a flow control feature, which, like clustering, will lower the number of messages sent by A^2. Flow control produced good results but adds two additional parameters: the percentage of messages that should not be sent, and how fast this percentage decreases in a round. Similarly, not all networks produce good results when they are clustered; the results depend on the number of nodes in the network and the density of the network.

5

Implementing Clustering in A^2

In this section, we describe problems we encounter and implementation specific considerations we make when implementing our design in A^2. We begin by outlining the modifications we make to the A^2 system to accommodate the clustering implementation. Next, we explain implementation specific details regarding the Clustering and Demote services. Last, we discuss the consequences of some implementation details, and how they affect performance and fault tolerance.

5.1 Modifications to A^2

We make several changes to the architecture of the A^2 system, in Fig. 5.1 we highlight the architectural additions we have made to the A^2 system in Fig. 5.1; we have added a Cluster module in the Synchrotron kernel and the Demote and Clustering services in the A^2 layer. The bottom layer, device drivers, is not part of this thesis. However, we modified and added code in the other layers to implement clustering.

The Cluster module of the Synchrotron layer handles the low-level details of channel hopping and dynamic assignment of initiators. Also, we implement the Clustering service which elects the CHs, announces them to the network, and lets every regular node select a CH to join using the Join service from the A^2 layer. We also implement the Demote service which runs after all clusters have run the Join service, it demotes CHs that have fewer nodes than the parameter *minimum cluster size*. Finally, we make some slight modifications to the application layer to implement forwarding during CH rounds.

5.1.1 Flags Field for Cluster Heads

To separate CH rounds from cluster rounds the nodes switch back and forth between intra- and inter-cluster communication every other round. Doing this means CHs are switching back and forth between different indices in the flags field as displayed in Fig. 5.2. The Clustering service sets the flag indices used by CHs during CH rounds, and the Join service sets the flag indices used during cluster rounds. During CH rounds the CHs use the same procedure for calculating completion as during

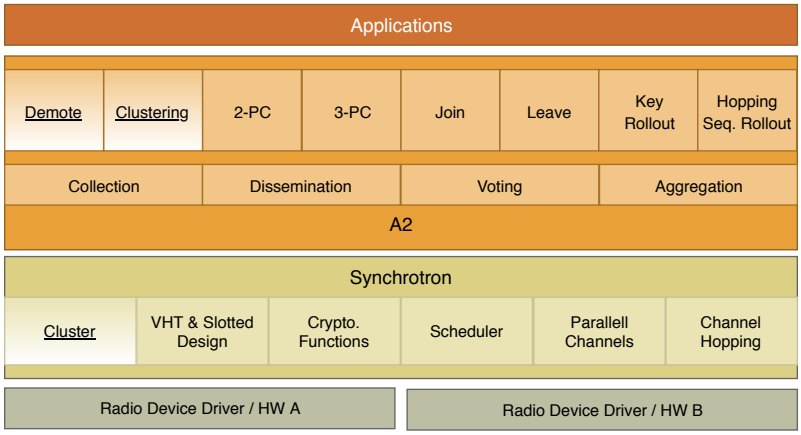


Figure 5.1: ***Layered system architecture of*** A^2 ***[4].*** *Shows the placement of the Cluster and Demote services. The Cluster module in Synchrotron handles per-cluster channel hopping and dynamic assignment of initiators. The Clustering and Demote services in* A^2 *contains the logic for electing and demoting cluster heads respectively.*

cluster rounds. However, their index in the flags field corresponds to their index in the CH list, and the length of the CH list determines the length of the flags field.

5.1.2 Separating Communication Between Clusters

We separate the clusters in two different ways. First, we use a modified version of the channel hopping in A^2. The original A^2 implementation hops between radio channels in two different ways as outlined in Section 2.3.3. In our implementation, we separate the clusters into different channels. A^2's parallel channels functionality is deactivated since one cluster should not be large enough to interfere with its communication. The nodes still follow the original hopping sequence but offset the channel number by the *cluster-index*, determined during the Clustering service. The cluster-index is unique, deterministic and since it is an index the numbers are sequential which makes it ideal to use for this purpose.

Second, each node includes its cluster ID in the packet header; if a node receives a packet from another cluster, the node discards that packet; nodes do not check this condition during CH rounds. Consequently, since all packets are slightly different in CH rounds, the constructive interference used by A^2 in the completion phase does not work during CH rounds.

5.1.3 Forwarders During Cluster Head Rounds

Since the HEED algorithm aims to elect non-overlapping cluster-heads, the cluster heads require assistance to communicate. Therefore, during CH-rounds all non-CH nodes act as forwarders. We implement the forwarders by letting all nodes run the

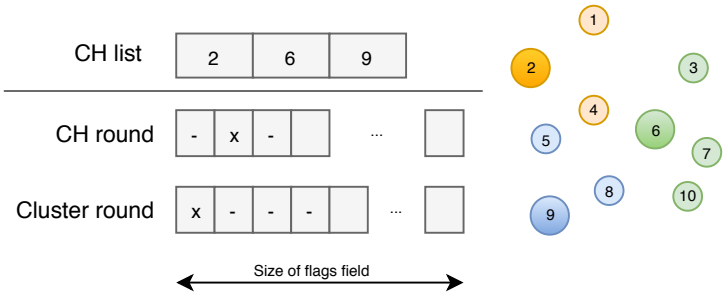


Figure 5.2: *The network on the right has elected the cluster heads in the CH list. Shown here is how Node 6 uses the flags field during CH and cluster rounds. The symbol 'x' represents the index which Node 6 uses as its flag, and the symbol '-' represents flag indices which are in use by other nodes. An empty box indicates that the index is not used. In the CH-round CHs uses the index from the CH-List. During a cluster round each CH will be an initiator and use index 0.*

Max application as usual but with the modification that non-CH nodes suggest 0 as their maximum value which is always overwritten by the CHs max value.

5.1.4 Initiating Communication in a Clustered Network

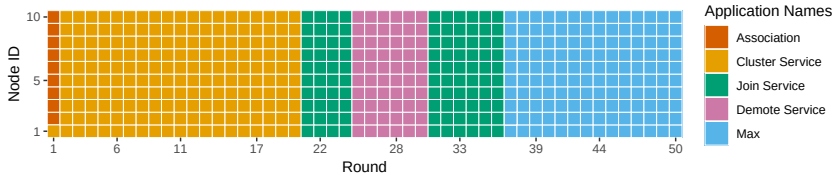
In our implementation, there are three different types of communication in which the initiator is different, as shown in Fig. 5.3b. First, when either the Clustering service or the Demote service is running, the network is using a single preconfigured node as the initiator. Second, during cluster rounds, each CH acts as the initiator in its cluster. Third, during CH rounds the first CH in the CH list is the initiator.

Switching the initiator between different nodes introduce problems. Namely, the network can end up in a state where none of the nodes considers themselves initiator, which means every node in the network is stuck associating indefinitely. Two scenarios can occur.

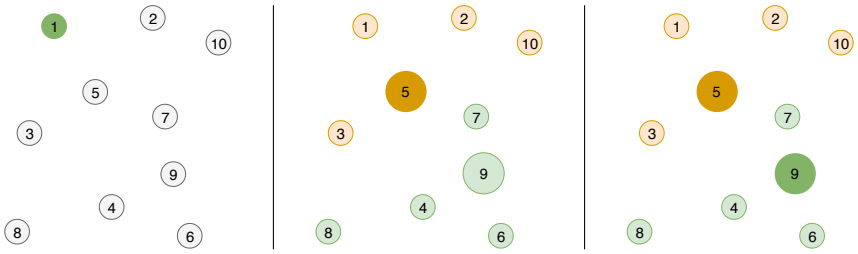
First, if the CH with the lowest ID loses connection to the network, there will be no initiator during CH rounds. If that happens, no CH rounds will be executed by the network until after it has performed a re-clustering.

Second, a scenario which can occur that makes all nodes associate indefinitely.

1. The preconfigured initiator node, p loses connection with the network during a cluster round, when it does not consider itself initiator.
2. The next scheduling of the Clustering service happens before p can associate with the network again.
3. In the Clustering service all nodes wait for packets from p, but since p is associating they never receive any packets.



(a) *The first 50 rounds for a network with 10 nodes, and which application each node is running. Node 1 is the initiator and thus runs the Clustering service in round 1 while the rest of the nodes associates with it.*



(b) *Locations of the ten nodes showing three different initiator setups. Node 1 is initiator rounds 1-20 and 25-30. Nodes 5 and 9 are initiators in rounds 21-24, 31-36 and every cluster (odd) round in the interval 37-50. Node 5 is the initiator in every CH (even) round in the interval 37-50.*

Figure 5.3: *An example network and its application map. Showing which applications the network executed, and which nodes are initiators during the different phases.*

5.1.5 Interaction Between Services

In this thesis, we implement two different services: the Clustering service and the Demote service. Furthermore, we use the Join service, provided by A^2, within the clusters to get a consistent membership. However, for the clustering process to succeed, they have to be run in a specific order. First, the Clustering service needs to be run and converge on a set of CHs. Following that, the network executes the Join service within each cluster, allowing each CH to learn how many nodes have decided to join its cluster. Then the Demote service executes, during which CHs demote themselves if they have too few nodes in their cluster. When the network has completed the Demote service, it reruns the Join service since some clusters might have demoted themselves; the network needs to make sure that any node that changed cluster is picked up by a new cluster. At this point the clustering process is complete, and the intended operation of the network can continue.

5.2 The Clustering Service

In this section, we explain implementation specific details that are related to the Clustering service. We go into detail about the packet payload and what information we transmit from a node and why; we also describe challenges we face due to not having completion flags. Finally, we explain a Catch-up mechanism we implement

Table 5.1: *The parameters and their sizes in the Clustering service packet payload.*

Name	**Size(bytes)**
Source ID	1
Consecutive cluster rounds	1
Cluster head count	1
Cluster head list	$3 * 30 = 90$
Total	**93**

in the Clustering service to handle nodes joining the network during the clustering process.

5.2.1 The Clustering Service Packet Payload

As mentioned in the design chapter, nodes running the Clustering service exchanges information to inform the network of how many CHs have announced themselves. In this section, we describe the payload of the Clustering service packet.

The content of the Clustering service packet payload is summarised in Table 5.1. The transmitting node's ID is used in the Clustering service to count the number of packets received from a node. The consecutive cluster rounds is a counter of how many rounds has elapsed since the Clustering service started to run; we use this parameter in the Catch-up mechanism, which we explain in Section 5.2.2. The last entry is the CH list, which uses three bytes of space per entry. The list has a maximum size of 90 bytes, allowing a maximum of 30 CHs. Each entry in the CH list contains the CH's ID; the hop count to it; and its status, which is either *Tentative* or *Final*.

The packet's payload size is restricted by the maximum packet size of the IEEE 802.15.4 standard, which is 127 bytes [15]. It is restricted further by the A^2 header size and data added in the link layer, giving us a maximum of 112 bytes for the payload size. As can be seen in Table 5.1, we leave 19 bytes in the packet since some features which we are not using in A^2 take up space in the packet header, we do not want the Clustering service to break when other features of the A^2 system are activated.

5.2.2 The Catch-up Mechanism

The last action a node performs in a round is to double their probability to announce itself as tentative CH. However, a problem can occur if a node associates with the network during the execution of the Clustering service. Specifically, its probability to announce itself as CH, CH_{prob}, will have fallen behind compared to other nodes; this is most noticeable when the network has recently started, during the initial association of all nodes. For example, nodes associating with the initiator in round

Table 5.2: *Probability that a node has announced itself as CH in a specific round without the Catch-up mechanism.*

Round(r)	CH_{Prob}	**Cumulative probability starting at round r**						
1	0.005	0.005						
2	0.010	0.015	0.005					
3	0.020	0.035	0.015	0.005				
4	0.040	0.073	0.035	0.015	0.005			
5	0.080	0.147	0.73	0.035	0.015	0.005		
6	0.160	0.284	0.147	0.073	0.035	0.015	0.005	
7	**0.320**	**0.513**	**0.284**	**0.147**	**0.073**	**0.035**	**0.015**	**0.005**

Table 5.3: *Probability that a node has announced itself as CH in a specific round with the Catch-up mechanism.*

Round(r)	CH_{Prob}	**Cumulative probability starting at round r**						
1	0.005	0.005						
2	0.010	0.015	0.010					
3	0.020	0.035	0.030	0.020				
4	0.040	0.073	0.069	0.060	0.040			
5	0.080	0.147	0.143	0.134	0.117	0.080		
6	0.160	0.284	0.280	0.273	0.258	0.227	0.160	
7	**0.320**	**0.513**	**0.511**	**0.506**	**0.496**	**0.474**	**0.429**	**0.320**

one randomises their initial CH_{prob} in round two. However, in round two the initiator has already doubled its own CH_{prob} once.

As a remedy, we implemented a *Catch-up mechanism*. In the Clustering service's packet payload, we attach the number of rounds the Clustering service has executed, which a node saves once received. At the beginning of a round, all nodes that have received a value for this variable increments it. The counter describes the number of times each node's initial CH_{prob} should have doubled since the start of the Clustering service. An advantage remains for nodes already part of the network or the ones which join earlier in the service, since they have had more chances to announce themselves as CH. However, as Table 5.2 and Table 5.3 show, this advantage is very small.

To motivate the usefulness of the Catch-up mechanism, we present the probability that a node has announced itself as CH at round r without the Catch-up mechanism in Table 5.2 and with the Catch-up mechanism in Table 5.3. For example, if we want to see the probability that a node which associated with the network round 3 has announced itself as CH in round 6 with the Catch-up mechanism. We look at Table 5.3 for the column that has its first value in round 3 and in that column look at the value for round 6, which is 0.273.

Since the probability of a node announcing itself as CH in any round is independent, we can describe a function $P(r)$ for a node's cumulative probability to have announced itself as CH at round r, shown in Eq. (5.1). Let CH_{prob} be the starting

probability of a node and b the round where a node n associates with the network and starts to run the Clustering service.

$$P(r) = 1 - \prod_{i=b}^{r}(1 - 2^i * CH_{Prob}). \tag{5.1}$$

Looking at Table 5.2 and Table 5.3, where we set $CH_{prob} = 0.005$ as the initial probability, the most interesting data points are the two bottom rows, emphasised in bold. Without the Catch-up mechanism, there is a significant difference in the cumulative probability that a node has announced itself as CH. However, with the Catch-up mechanism, the difference is minimal except for in the most extreme case when a node starts at round seven, compared to a node that started at round one. A specific example shows that when we do not use the Catch-up mechanism a node that starts the Clustering service at round three will at round seven have a $0.513 - 0.147 \approx 38$ percentage points less chance of having announced itself as CH compared to a node that started at round one. However, with the Catch-up mechanism the difference is only $0.513 - 0.506 \approx 0.7$ percentage points. The same pattern can be seen for other starting points as well, giving a clear motivation that the Catch-up mechanism is useful.

5.2.3 The Demote Service

The Demote service is responsible for removing CHs that are considered suboptimal. It demotes all CHs which have fewer nodes in their cluster than the parameter *minimum cluster size*.

The Demote service behaves similarly to the Clustering service; each CH decides to demote itself locally. The service is then run for some number of rounds to ensure that the information spreads to the whole network. At the end of the last Demote service round, all demoted CHs and all nodes that joined their clusters select a new CH from the CHs that remain.

5.3 Discussion

In this section, we discuss the consequences of implementing clustering in A^2, we motivate our choices and propose solutions to problems that arise from the implementation.

5.3.1 Destructive Interference During CH Rounds

Since we include the cluster ID in every packet, there is no constructive interference during CH rounds. One prominent feature in Chaos is the completion flooding,

which relies on constructive interference to achieve both high reliability and early turnoff [3]. In our implementation, nodes will still perform completion flooding, but they will not have constructive interference.

Gauging the impact of this is hard. However, we argue that it is not significant. The purpose of the completion flooding is to get a significant portion of the nodes to agree on the final value quickly. In CH rounds, the number of nodes that propose values and need to agree on the final value is small, compared to the total number of nodes in the network. Therefore, the completion flooding has less of an impact on the completion of the network.

5.3.2 Forwarders during CH rounds

In our implementation, the forwarders execute the same code as CHs during CH rounds, with the difference that they do not propose any value of their own. However, a more general implementation is possible. A forwarder should only require the knowledge of how to merge two packets for an application. However, an application's merge operator is not abstracted to a separate function. If applications would provide their merge operation through an interface, a more general implementation in A^2 could use an applications knowledge of how to merge packets without actually running the application. This would make the clustering implementation easier to adapt to new applications, and the nodes could be more effective since they only have to run the code that merges the flags and not the entire application.

5.3.3 Fault Tolerance for Dynamic Initiators

As we describe in Section 5.1.4, switching initiators between cluster and cluster head rounds caused some problems. In this section, we discuss two different solutions to these problems.

The first solution could be to always use the preconfigured initiator during CH rounds, regardless of the state of the network. However, this would create an even higher dependency on this node, increasing its energy usage, which would reduce the lifetime of that node and therefore the network. Furthermore, we would have to handle the cluster the preconfigured node joins as a special case to avoid having two initiators in that cluster.

Another solution could be to use a fallback error handling scheme. If the preconfigured initiator thinks that the network has not made any progress for some time, it could assign itself as initiator and restart the network. However, the problem is that it is hard to detect that this has happened accurately. First, the initiator could use a timeout counted in rounds; if it does not receive any packet before the timeout reaches a certain threshold, it assumes that the whole network is stuck associating and assign itself as the initiator. Second, since the length of a round is preconfigured, it could calculate the time until it should take over as the initiator (since the Clustering service is statically scheduled) and assign itself as initiator at that point.

However, both of these solutions require that the preconfigured initiator can accurately keep track of the time, even in the presence of faults. If its clock drifts too much, or other transient faults cause the clock to be out of sync with the rest of the network, the node could assign itself as the initiator at the wrong time and start to broadcast erroneous packets. In that case, the network could break down since it would have multiple initiators telling them different things regarding the next application to schedule and the timing of the rounds.

5.3.4 Clustering Without Completion Flags

Because we do not have completion flags in the Clustering service, the network cannot know when all nodes have participated. Instead, we artificially increase the communication that occurs during a round to increase the probability for consensus. Below, we explain a scenario where, even in the presence of an update in the network, the update is not propagated throughout the network, and then how we circumvent that scenario.

Even if a node is infective, it will only transmit after it has received a packet originating from the initiator. An example would be a network topology with a single path, where the initiator is at one end, and a node with an update is at the other. The nodes in between will receive a packet from the initiator, but if the initiator does not have an update, they will not transmit in the next slot. The only way that communication progresses on the path, in that case, is if a node in between triggers a retransmit timeout.

To counter scenarios where nodes with updates have a hard time receiving a packet, we enforce the initiator to always have an update at the start of each round. This is done by letting each node maintain two versions of the CH list. Each node merges received updates to a multi-round persistent copy. However, to always have an update in each round, all nodes maintain another copy of the CH list which is reset every round. The initiator initiates communication using the copy that is reset every round and includes an update in the form of incrementing the consecutive cluster round count. The update propagates throughout the network and nodes with new updates add that information to the packet once they receive the first update.

5.3.5 Completion Flags during Cluster Head Rounds

Since we use the CH list to assign flag indices during CH rounds, the list has to be consistent across all nodes, which means that we can never have more CHs in the network than the maximum size of this list. It is not possible for nodes to discard CHs from their list that, for example, are outside of their competition radius since in that case, some parts of the network would not consider that CH when calculating the progress of a round.

This problem is a limitation on our implementation. It should be entirely possible to modify the Join service so that it can run for the CHs, eliminating the need for

the CH list to be consistent across all nodes. Nodes could then filter out the CHs that are outside of their competition radius. However, due to time limitations, we have not explored this solution any further.

6

Evaluation

In this chapter, we evaluate our clustering implementation. We begin by defining the topologies and the metrics we use. We continue with an evaluation of our clustering parameters, where we determine what configurations of the parameters yield the most reliable and stable network, for different topologies. Next, we compare our clustering implementation to the A^2 system. We end this chapter with a discussion on the evaluation results, how our limitation on fault tolerance affected our results, and running different applications in a clustered network.

6.1 Evaluation Setup

We run several different tests to evaluate our clustering implementation. In this section, we describe our test setup, the network topologies we run our tests on, and the metrics we use.

6.1.1 The Cooja Simulator

Cooja [9] is a simulator for ContikiOS [8], the operating system that Chaos and A^2 are implemented on [3, 4]. In Cooja, we create networks with 50 and 200 nodes placed randomly in topologies of different sizes, to get both dense and sparse networks. We start with 50 nodes in a 100x100 m^2 area and increase the side of the square in steps of 300 meters up to 2500x2500 m^2. We do the same for 200 nodes but only up to 1300x1300 m^2. We only have one network topology per area and number of nodes. We simulate each test for 30 minutes, which gives us 600 rounds of data.

We run most of our tests using 50 nodes for two reasons. First, 50 nodes are enough to see that the Clustering service creates several clusters. Second, running these simulations takes a significant amount of time, and it does not scale well when increasing the number of nodes. Simulating 30 minutes with 50 nodes takes 2-3 hours, and 200 nodes takes 30-50 hours on our hardware. However, when comparing our clustering process to the A^2 system, we also simulate 200 nodes to see the effects of scaling up the network. Moreover, we chose the different sizes from the fact that

in Cooja, a network with an area of 100x100 m^2 ensures that is is a 1-hop network and at 2500x2500 m^2 the network is at least a 7-hop network.

6.1.2 The Flocklab Testbed

The Flocklab testbed [10] consists of 27 nodes deployed both indoors and outdoors at ETH Zurich. Flocklab provides several services that make it easy to measure and evaluate the code that is running on the nodes. Our tests log data to the serial port of an observer node, Flocklab aggregates the data and sends it to us at the end of each test.

6.1.3 Metrics

To define the metrics we use, we make a distinction between rounds which execute coordination and rounds which execute applications. Coordination is rounds in which the network is set up to enable applications to run; coordination includes the Cluster, Join, and Demote services. We also differentiate between static and dynamic coordination. Static coordination is scheduled at fixed round intervals to, for example, cluster the network. Dynamic coordination, on the other hand, is scheduled by nodes as needed during the application phase, to handle faults in the network or new nodes. Applications are programs that run with the goal of providing some value or perform some calculation; Some examples of applications are finding the maximum or mean of all values proposed by nodes in the network, or performing a two-phase commit.

We consider four metrics when evaluating clustering on A^2: stability, reliability, latency, and energy usage. Reliability is previously defined by Landsiedel et al. [3] as *"the percentage of rounds in which all nodes reach completion"*. However, in our case, clusters may reach completion independent from each other. Therefore, we measure reliability in a way which reflects that some clusters can succeed while others fail during a round. Furthermore, we introduce the metric stability, since we want to measure how much time is spent by the network running dynamic coordination. The amount of dynamic coordination in a network should be kept to a minimum. Moreover, both latency and energy usage is measured in previous work conducted by Landsiedel et al. [3]. We define all metrics for one execution of a network, that is, one simulation in Cooja or one run on the Flocklab testbed.

Stability is a measurement of how often topology changes occur. We say that a node is stable if it is executing an application and unstable if it is executing dynamic coordination. Stability during one round is the proportion of nodes that are stable. We give the following definition for the stability for one test.

Definition

Let s_i be the set of stable nodes in round i, n the set of all nodes, and R the set of all rounds which do not execute static coordination.

$$Stability = \frac{\sum_{i \in R} |s_i|}{|n| * |R|}$$

Reliability is a measurement of how well an application runs on the network; thus, it is only measured during application rounds. Reliability during one round is the proportion of stable nodes that successfully execute an application. We give the following definition for the reliability for one test.

Definition

Let a_i, $a_i \subset s_i$, be the set of stable nodes that succeeds with an application in round i, and R_a the set of all rounds in which an application is executed.

$$Reliability = \frac{\sum_{i \in R_a} |a_i|}{\sum_{i \in R_a} |s_i|}$$

Latency is a measurement of how long it takes for an application to terminate. For a stable node in one round, latency is the number of slots from the start of the round until the node powers down. We give the following definition for the latency for one test.

Definition

Let l_{ij} be the latency for node j in round i.

$$Latency = \frac{\sum_{i \in R_a} \sum_{j \in s_i} l_{ij}}{\sum_{i \in R_a} |s_i|}$$

Energy usage is a measurement of the average amount of energy used by a node per Energest time unit. Energest is the built-in energy estimation module in Contiki and we measure energy usage in all rounds, both during coordination and applications. We give the following definition for the energy usage for one test.

Table 6.1: *The parameters we evaluate and their default values*

Parameter	Value
Competition Radius	1
Minimum cluster size	4
Nodes per cluster ratio	10
Round re-synchronisation threshold	3

Definition

Let e_i be the energy used by node i in one test, and let T be the running time of the test in Energest time units.

$$Energy\ usage = \frac{\sum_{i \in n} e_i}{T * n}$$

6.1.4 Limitations

Due to time constraints we impose several limitations on the evaluation of our clustering implementation. We only run tests in the Cooja simulator and on the Flocklab testbed. Running simulations gives us the freedom to choose topologies, but the simulations do not model the network perfectly. Flocklab, on the other hand, contains real nodes but is relatively small with only 27 nodes in a static configuration. We only compare our implementation to A^2, and do not consider any other WSN protocols.

Furthermore, we only simulate two different network sizes, 50 nodes and 200 nodes. We only simulate 50 nodes when searching for parameter values, and include networks with 200 nodes when we compare our clustering process to A^2.

To limit the scope of our parameter evaluation, we use the values listed in Table 6.1 for the parameters that we are not currently evaluating.

Lastly, we only evaluate clustering using the Max application. The Max application is the most straightforward application currently implemented in the A^2 system. We discuss how to apply different applications and the implications of doing so in Section 6.4.3.

6.2 Clustering Parameters

In this section, we experimentally search for optimal values to the parameters in our clustering algorithm. The parameters are *round re-synchronisation threshold*, *competition radius*, *minimum cluster size*, and *nodes per cluster ratio*. We run each parameter configuration three or six times and plot the reliability and stability of each test for each network topology.

We only look at the reliability and stability of these tests since the purpose is to find which parameters yield the best results, in terms of running the application. However, we include the latency results in Appendix A for completeness; we note that there are no significant differences in latency for different values of these parameters, and will not comment on these results any further.

We use a different number of topologies depending on which parameter we evaluate. When evaluating competition radius, we test on all network topologies we describe in Section 6.1.1 since this parameter is dependent on the density of the network. However, when we test the parameters minimum cluster size and nodes per cluster ratio, we use topologies from 100x100 m^2 to 1300x1300 m^2, using competition radius 1. We argue that increasing the competition radius while simultaneously increasing the area of the networks will not give any new results. By excluding the network topologies with a larger area, we could repeat each configuration for these tests a total of six times, instead of three.

6.2.1 Round Re-synchronisation Threshold

The *round re-sync threshold* parameter is not directly related to the clustering process. Rather, it is a setting that controls a fundamental part of the A^2 system. Round re-sync threshold is a numeric parameter that determines how many rounds a node will spend receiving no packets before associating with the network again.

We evaluate three different values of the round re-sync threshold (1, 2, and 3), for each value of re-sync threshold we evaluate three different values of competition radius (1, 2, and 3), which gives us a total of 9 test configurations. We use different values of competition radius for these tests because we do not want a suboptimal competition radius to affect the results regarding the re-sync threshold, especially for sparser networks.

We make two observations from the round re-sync threshold results, shown in Fig. 6.1. First, the reliability is drastically increased when the re-sync threshold is greater than 1. Second, the stability increases as the re-sync threshold increases.

The reason reliability is low when the re-sync threshold is 1 (Fig. 6.1a), is that a clustered network is not compatible with such a low threshold. The cause of this problem is that the network has a single initiator during CH rounds, which is the CH with the lowest ID. Furthermore, we allow clusters to schedule and run the Join

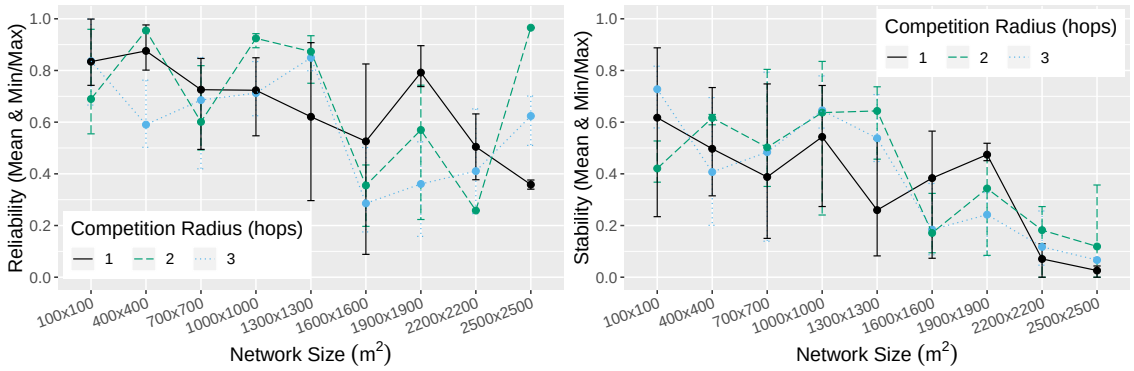
service during CH rounds, meaning those clusters will not participate with their value, neither will they forward packets during CH rounds.

The problem, which often occurs, is that the cluster with the lowest ID runs the Join service during a CH round, in that case, there is no initiator present for the CH round. Without an initiator, all other clusters that participate in the CH round, will increase their resynchronisation counter by one, reach the resynchronisation threshold, and start associating. Furthermore, as we discuss in Section 6.4, our limitation regarding fault tolerance further exacerbates this problem.

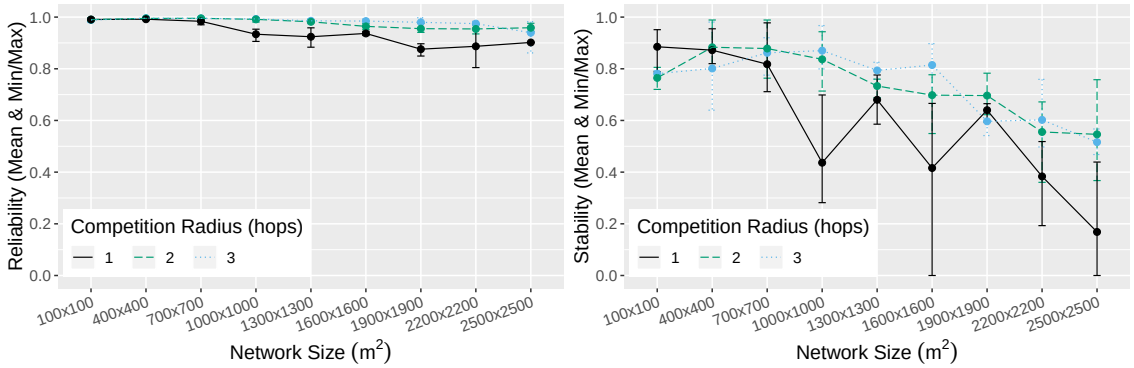
Usually, these scenarios only affect the stability of the network. However, in this case, the reliability is also affected since the network never has a chance to continue normal execution. Since the re-sync threshold is set too low, there is a high probability that every round has some portion of nodes associating, and some portion of nodes running the max application but failing since the only initiator is running the Join service.

Furthermore, the stability increases with the re-sync threshold. The increase in stability when increasing the parameter from 1 to 2 follows from the same reasoning as for the reliability. However, the stability continues to increase when going from 2 to 3 as well. With an increase in the re-sync threshold, there is a lower chance that a node will re-associate with the network since it has a higher chance of receiving at least one valid packet before it reaches the threshold.

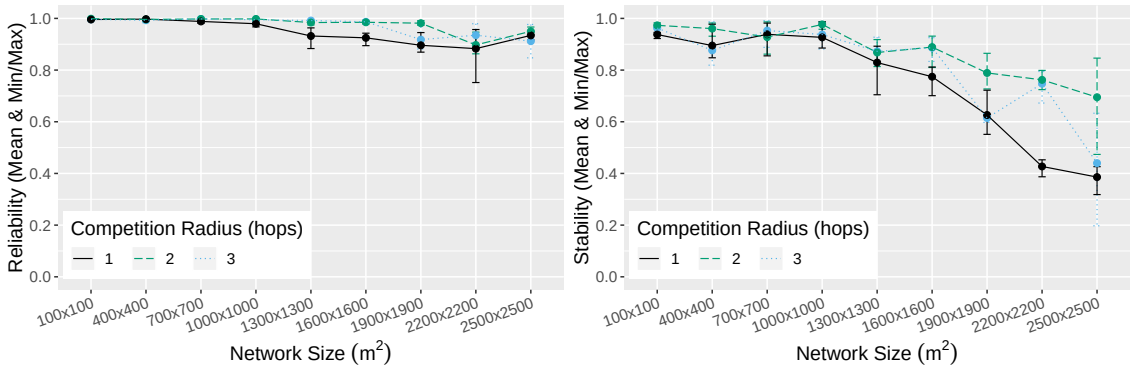
From this evaluation, we conclude that out of the values we tested, using a re-sync threshold of 3 gives the best results. Even though Fig. 6.1 imply a trend of higher stability with an increasing re-sync threshold. However, we did not test greater values of this parameter due to time constraints, we discuss implications of increasing it more in Section 6.4.2. All further evaluation of our clustering implementation uses a re-synchronisation threshold of 3.



(a) *Re-synchronisation threshold 1.*



(b) *Re-synchronisation threshold 2.*



(c) *Re-synchronisation threshold 3.*

Figure 6.1: *Resynchronisation threshold tests for different values of Competition radius. Both reliability and stability increases as the re-sync threshold increases.*

6.2.2 Competition Radius

Competition radius determines the distance, in hops, from which a node can choose a CH. For this parameter, we evaluate the values 1, 2 and 3, which ensures that we test the minimum value of 1 but also not get too sparse clusters. We show the average, minimum, and maximum reliability and stability for each network size and each competition radius in Fig. 6.2. Note that this figure is the same as Fig. 6.1c because we set the re-synchronisation threshold to 3 for all tests. Consequently,

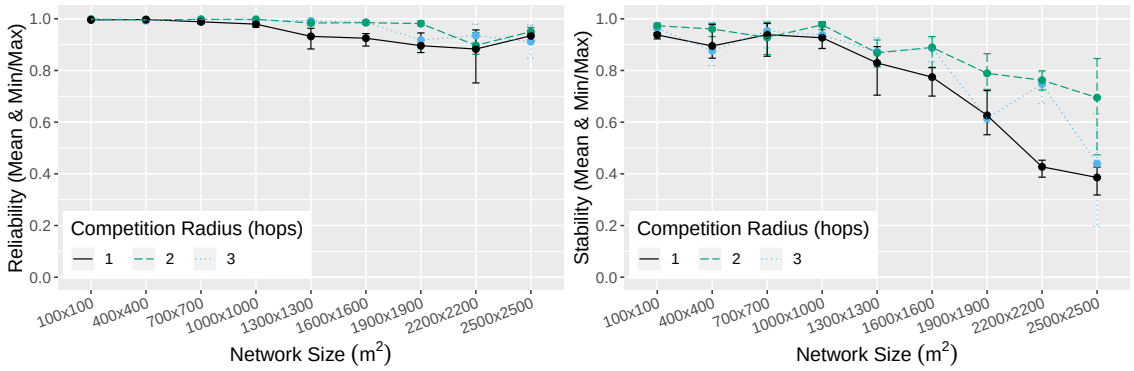


Figure 6.2: *The reliability and stability for different network sizes. Each network size has been tested with competition radius 1, 2, and 3.*

both of these tests use the same parameter values, so we did not rerun the test.

We see that both reliability and stability decreases as the size of the networks increase. We expect to see this decrease since clustering sparse networks will make them even sparser, which hinders communication. However, we see that higher values of competition radius are better for these networks since larger clusters are created, which does not impact communication as much as small clusters. Furthermore, the differences in stability and reliability for the denser networks (100x100, 400x400, and 700x700) is noise. These networks are 1-hop networks, and thus the competition radius value does not affect the clustering.

6.2.3 Minimum Cluster Size

Minimum cluster size determines the smallest cluster that is allowed to exist after the clustering process has converged on a set of clusters. All clusters with fewer nodes are removed, and the affected nodes join other clusters. We evaluate this parameter using three different values, 2, 4, and off; off means that no clusters are considered too small. We did not test any larger values for this parameter since we only use 50 nodes for the parameter tests, and a value larger than 4 would mean that more than 10% of the nodes in the network could be in a cluster and it would be removed, which we consider too aggressive. We show the reliability and stability for these tests in Fig. 6.3.

When we look at the reliability for this parameter, we see that there is no significant difference for any topology or parameter value. However, we see a small variance in stability for larger networks. Looking at the data on how many cluster heads were created and demoted for these tests, which we can see in Fig. 6.4, we see two interesting results. First, using higher values of this parameter show no significant difference in the number of CHs created by the clustering process. Second, there was one test which demoted many more CHs than the others, the value 2 for the 100x100 m^2 topology. The high number of demoted CHs resulted in the CH count being close to the average. In that test, 32 CHs were demoted while normally 1-2 CHs are demoted. This recovery demonstrates that using this parameter can help

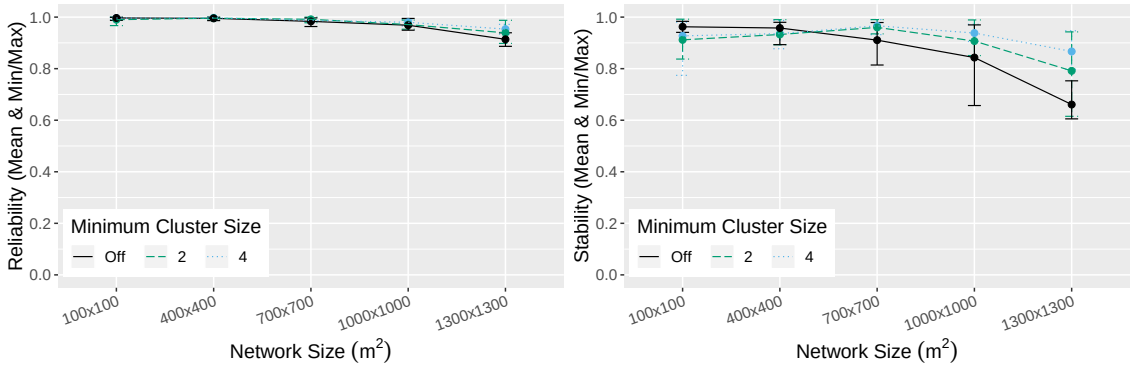


Figure 6.3: *The reliability and stability for different network sizes. Each network has been tested with minimum cluster size values off, 2, and 4.*

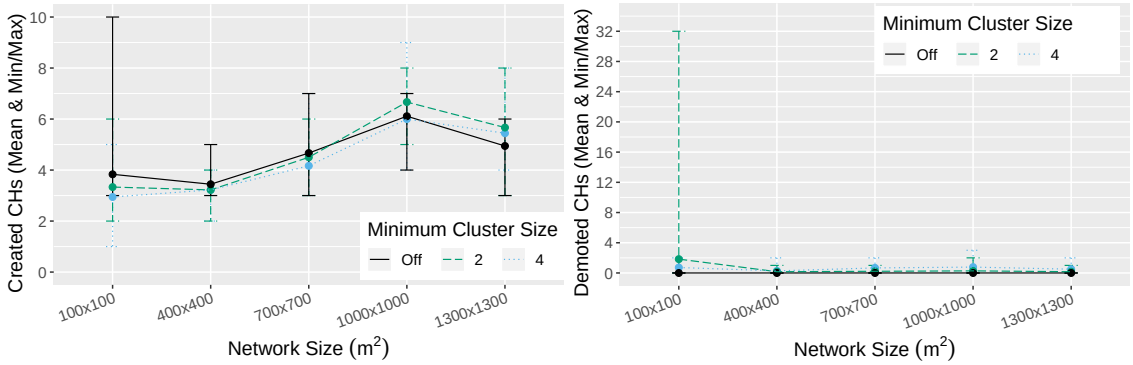


Figure 6.4: *Shows the number of CHs after the clustering process has finished, and the number of demoted CHs.*

the clustering process handle faults in the Clustering service.

6.2.4 Nodes per Cluster Ratio

The *nodes per cluster ratio* parameter determines if nodes may become CHs even if they have heard from another CH within the competition radius. The parameter describes a ratio of how many neighbours a node needs to have relative to the number of CHs the node has heard. We test the values 5, 10, 15, and off for this parameter. The value off means a CH will never announce itself if it has heard from another CH within the network's competition radius. The results can be seen in Fig. 6.5.

The most significant results for the nodes per cluster ratio parameter is the decrease in stability and reliability in the 1000x1000 and 1300x1300 m^2 networks. With more clusters in networks with a larger area, we see more clusters failing with communication, causing nodes to try and re-synchronise, which is worsened by our limitation on fault tolerance. Furthermore, the two deviations in stability for 100x100 m^2 in the case of 10 and 15 nodes per cluster ratio are both due to noise.

Additionally, we see a small but constant decrease in reliability for the parameter value 5. This decrease is because the Clustering service creates too many clusters

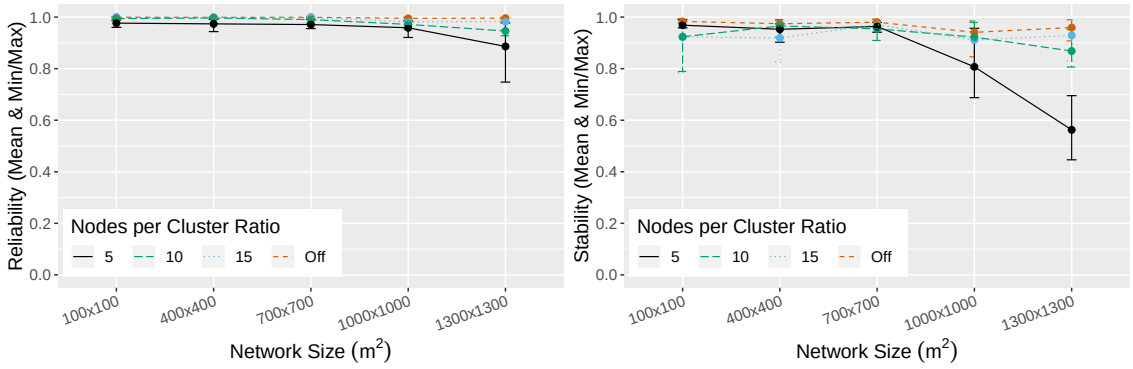


Figure 6.5: *The reliability and stability for different network sizes. Each network size has been tested with max node count off, 5, 10, and 15.*

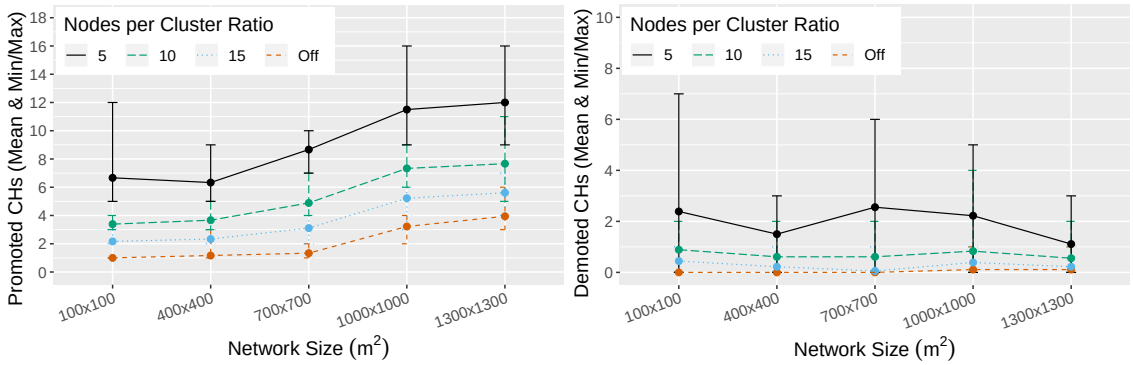


Figure 6.6: *The number of elected CHs before the Demote service runs, and the number of demoted CHs.*

which are only demoted or otherwise removed, as can be seen in Fig. 6.6. Since demoting a CH may cause a fault, increasing the number of demotions will increase the probability of a fault to occur. Usually, these scenarios only affect the stability of the network, but in this case, the reliability is also affected since the CHs could not agree on which CHs was demoted and thus never agree on the correct maximum value in the CH rounds.

Nonetheless, we noticed that more nodes announced themselves as CH with a smaller nodes per cluster ratio value as seen in Fig. 6.6; this shows that the parameter works as intended but requires careful consideration when configured, and needs to be adapted to the network topology.

6.3 Comparing A^2 with Clustered A^2

In this section, we show the results of evaluating our implementation of clustered A^2 and compare with the results from our evaluation of the original A^2 system. We show results for stability, reliability, latency, and estimated energy usage for all tests we run. We use different parameter values for some network topologies during this comparison. The most significant change is that we increase the competition

radius as the network sizes increases. We list all parameter values for these tests in Appendix B.1.

6.3.1 Reliability and Stability

We show reliability and stability for the evaluations with 50 nodes in Fig. 6.7a and for 200 nodes in Fig. 6.7b. Looking at the reliability for both of these, we see almost no difference for most network topologies, except with 50 nodes for 2200x2200 and 2500x2500 m^2, where the reliability drops when using clustering. The reason it drops is presented in detail Section 6.2.2, but essentially, too small clusters are created for these sparse networks.

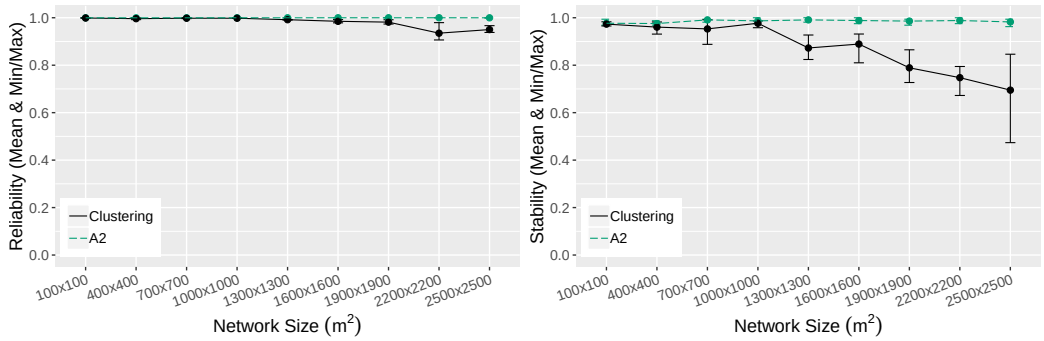
However, the stability results have a higher variance. In the networks with 50 nodes, the clustered networks have lower stability, there are some tests which achieve similar levels of stability, but on average it is lower or much lower. In networks with 200 nodes, on the other hand, clustering achieves better results, the average stability is still lower than for the A^2 system, but there are some tests where clustering outperforms A^2 slightly.

The reason A^2 drops in stability for 200 nodes, is that when one node loses connection with the network, the whole network has to spend on average, two or three rounds running the Join service. The clustering implementation, on the other hand, should only have to run the Join service locally in the affected cluster, which impacts stability less. However, since the clustering implementation lacks fault tolerance (discussed in Section 6.4), the stability results for clustering are worse than they could be.

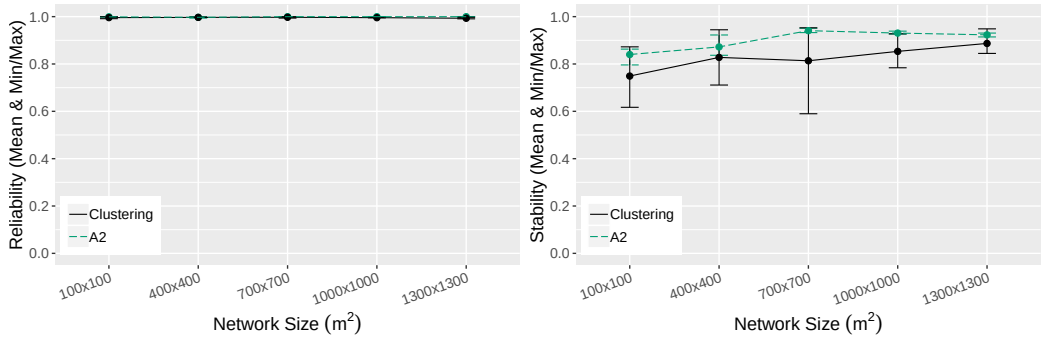
Furthermore, the stability has a decreasing trend in the 50 node networks while it is increasing in the 200 node networks. The reason the stability has an increasing trend in 200 node networks is that the small networks are dense and therefore generate more interference, which increases the number of faults that occur. However, we have not tested larger networks than 1300x1300 m^2 using 200 nodes, and it is possible that the stability will show a similar trend as the 50 nodes network when the area of the networks is increased further.

6.3.2 Latency

We show the mean latency and standard deviation for each test we run in Fig. 6.8. Looking at the tests using 50 nodes (Fig. 6.8a) the latency for the clustering implementation is on average a little lower for network topologies smaller than $2200x2200m^2$, after that the A^2 system outperforms our implementation. However, for all network topologies our clustering implementation has a higher standard deviation. We see a similar trend for 200 nodes (Fig. 6.8b), the standard deviation is still high; however, our clustering implementation consistently has a lower mean latency for all topologies tested.



(a) *Networks with 50 nodes.*



(b) *Networks with 200 nodes.*

Figure 6.7: *Reliability and stability comparison between A^2 with clustering and original A^2.*

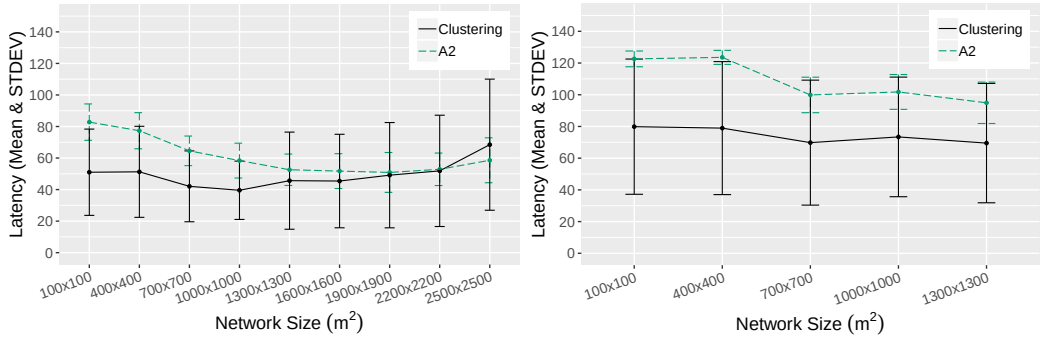
If we compare the latency between 50 nodes and 200 nodes, we see that it increases as the number of nodes increases. Consequently, we get lower latency when applying clustering since each cluster only contains a proportion of all nodes in the network.

6.3.3 Energy

In Fig. 6.9, we show the mean energy usage per node per Energest time unit. We observe in Fig. 6.9a that energy usage for clustering increases with network area. We also observe that stability (Fig. 6.7b) has an inverse relationship to energy usage, because an unstable node consumes more energy than a stable node. Both associating with the network and running the Join service requires more energy than the max application. They take more energy since the Join service takes longer time than the max application, and when a node associates it does not sleep.

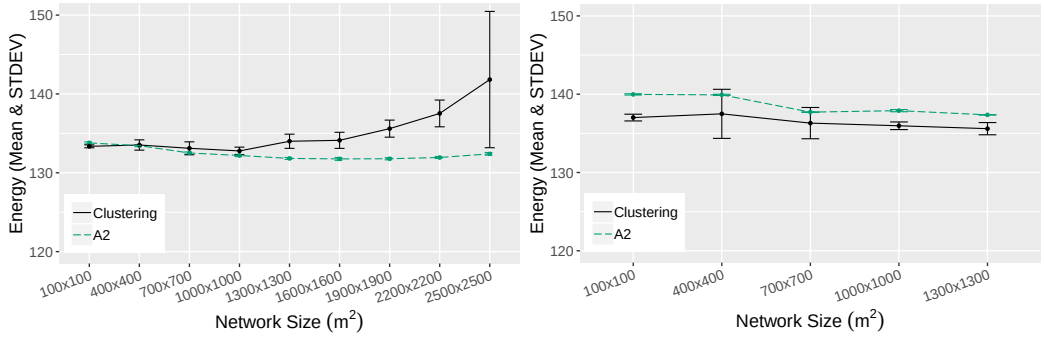
Comparing 50 to 200 nodes we notice a higher energy usage on average for 200 nodes, which we expect to see since more nodes require more communication and longer rounds, which leads to higher energy consumption. Clustering shows a small advantage with 200 nodes Fig. 6.9b, compared to A^2; each cluster can reach consensus in fewer slots than the whole network can, which is why the average energy usage is lower on all topologies.

Finally, the energy usage differs from the other metrics in that it is measured during



(a) *Networks with 50 nodes.* **(b)** *Networks with 200 nodes.*

Figure 6.8: *Latency comparison between A^2 with clustering and original A^2.*



(a) *Networks with 50 nodes.* **(b)** *Networks with 200 nodes.*

Figure 6.9: *Energy comparison between A^2 with clustering and original A^2.*

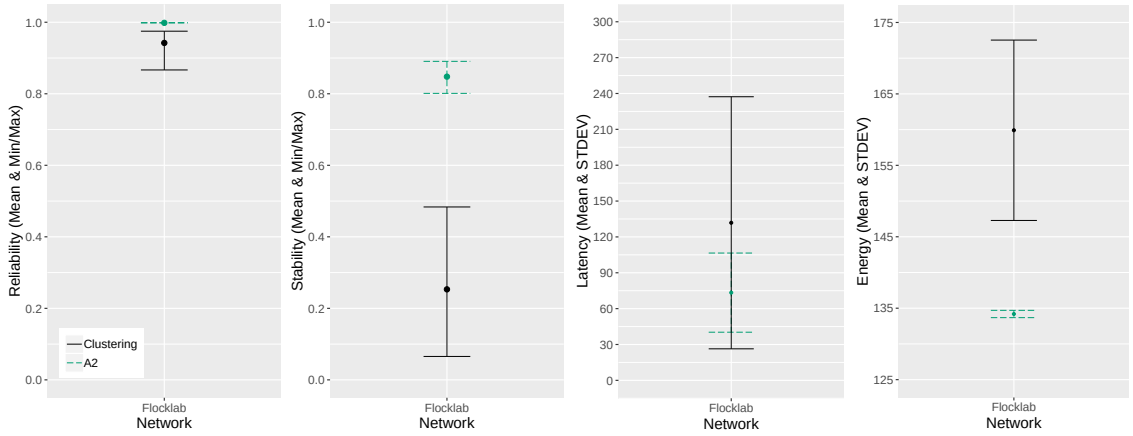
both the coordination and application phase. It is measured in all rounds because the energy results for only the applications could be misleading since the energy usage by the clustering process could outweigh the benefit of clustering the network.

6.3.4 Flocklab

We present the results from our tests on the Flocklab testbed in Fig. 6.10, which we get by running our clustering implementation and the A^2 system four times each. We see that A^2 outperforms our implementation in all metrics, achieving higher reliability and stability, lower latency, and lower energy consumption.

Flocklab is a small and relatively sparse network with only 27 nodes, and it is also affected by external interference, which the simulations do not model. All of these factors contribute to lower stability, for both A^2 and the clustering implementation.

When clustering the Flocklab network what often happens is that some node loses connection and need to associate with the network, to handle this the Join service is scheduled. Because no test run created more than two CHs, approximately half of the nodes begins to run the Join service, this makes it even harder for the other nodes to communicate, forcing more nodes to associate, creating a compounding effect



(a) *Reliability.* **(b)** *Stability.* **(c)** *Latency.* **(d)** *Energy.*

Figure 6.10: *The results of running A^2 and our clustering implementation on the Flocklab testbed. We show the mean and min/max for stability and reliability, and the mean and standard deviation for latency and energy consumption.*

of more and more nodes associating with the network. Nonetheless, running the clustering implementation on the Flocklab testbed demonstrates that the clustering process works when running on real nodes.

6.4 Discussion

In this section, we discuss insights from our results. Furthermore, we discuss how our limitation on fault tolerance affected the stability of the networks. We also discuss running other applications in a clustered network, and end with some comments regarding the scalability and energy efficiency of a clustered network.

6.4.1 Stability and Fault Tolerance

From the results we get from comparing A^2 to our implementation, we see that our clustering implementation achieves similar reliability in almost all cases. However, the stability has high variance and is, in some cases, much lower. The primary reason for these stability results is that we did not consider fault tolerance for our clustering implementation. In this section, we will describe and discuss what scenarios lead to lower stability and how they relate to fault tolerance.

One of the most common fault tolerance issues is that a node loses connection to the network and has to re-associate with it. For a node, this process should look as follows.

1. A node does not receive any packet for some rounds equal to *round re-synchronisation threshold*.

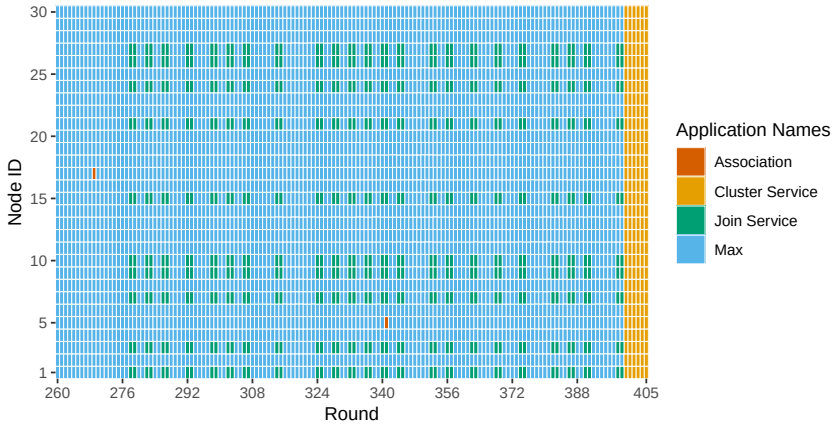


Figure 6.11: *Example of what happens when a node requests the Join service to be scheduled, blue is the correct application, green is the Join service, and yellow is the Clustering service. The first cluster repeatedly schedules the Join service without any effect.*

2. The node switches to association, listening for a packet with which it can synchronise.
3. The node sets the join flag in outgoing packet headers, requesting that the initiator schedule the Join service.
4. The current initiator schedules the Join service for its cluster.

By analysing the tests from the comparison of our implementation to the A^2 system, we see that stability issues always occur in step 4 of this process. We performed this analysis by looking at the outcomes of tests in the simulator; we do not attach any details of this here since it is too extensive.

Properly scheduling the Join service is complicated since we change both which node is the initiator and which nodes communicate while switching between cluster and cluster head rounds. The most common scenario, exemplified in Fig. 6.11, is that a node, in this case node 17, sets its join flag during a CH round, this happens first in round 270 in the example, and forces the first CH to schedule the Join service. The problem is that it does not matter which cluster node 17 has joined, it is always the first cluster that schedules the Join service. Consequently, since node 17 never gets the chance to join, because the wrong cluster is running the Join service, this scenario continues until a reclustering is scheduled.

Furthermore, if a CH loses connection to the network and has to associate, additional problems can occur. What happens depends on which CH associates. First, if the CH with the lowest ID associates, the network cannot continue normal execution, since that CH is the initiator during CH rounds. Causes and solutions to this problem are discussed more thoroughly in Section 5.3.3. Second, if another CH associates, the network will continue to execute the application. However, that node neither contributes its value nor sets its flag during CH rounds, which means the other nodes will never know that they have reached completion and cannot shut

down early during CH rounds. This scenario does not impact the stability of the application, but it increases the latency and energy usage.

Finally, that our clustering implementation works in the absence of faults is supported by the reliability and stability results in Fig. 6.7. Since the reliability metric does not include scenarios which require fault tolerance, it expresses the fact that clustering achieves good results during normal operation of the network. Additionally, from the stability results for 200 nodes, Fig. 6.7b, we see that there are some tests which achieve better stability than A^2, demonstrating that our clustering implementation has the potential to achieve better stability than the A^2 system in larger networks.

6.4.2 Clustering Parameters

The results we see in the parameter evaluation varies greatly, both when looking at a single test but also when looking at the different evaluations of the parameters combined. From the parameter evaluation, we can give some general remarks about each of the parameters.

As we see in the resynchronisation threshold evaluation (Section 6.2.1), both the stability and reliability increase significantly as the re-sync threshold increases. These results suggest that we should increase the re-sync threshold indefinitely. We did not evaluate this parameter further due to time constraints.

However, there are other reasons why indefinitely increasing this parameter will not yield the desired results. Re-sync threshold controls the number of rounds a node will wait while receiving no packets until it re-synchronises with the network again. Thus this parameter only fixes the symptom and not the cause of the problem, which is that a node has a bad connection to its cluster. For example, with a high re-sync threshold a node might stay in a suboptimal cluster since it receives enough packets to reset the re-sync threshold but not enough packets to complete the application running in the cluster, which will affect reliability.

For competition radius, we see that it is a parameter useful for creating reliable clusters since it had the most significant impact on the stability of a test for larger network areas.

For minimum cluster size, the results we got suggest that this parameter has a small impact on the overall stability. However, we see no correlation between the overall number of CHs and the stability, suggesting that these results either have some other underlying cause or the difference is due to noise. However, this parameter should be used, since it prevents clusters with only one node to form.

Last, the nodes per cluster ratio parameter worked as we expected. With a lower value of this parameter, we saw that the clustering process created more CHs. However, this parameter requires careful consideration since it can cause too many CHs to be created for sparse networks, as we see in Section 6.2.4.

6.4.3 Running Other Applications in a Clustered Network

Throughout this thesis, we limit our evaluation to the Max application because our focus was to implement clustering on the A^2 system. However, other applications might benefit from clustering in different ways, but in some cases, they might not be able to fully utilise the hierarchical communication medium. Additionally, more complex applications might require more adaptions to run in a clustered network. Consequently, the reliability of a clustered network depends on the application that is executing. In this section, we discuss how other applications run in a hierarchical network to highlight some issues that they possess, and to show that clustering is not limited to only the Max application. A typical application for a WSN is to perform data aggregation in the form of calculating a sum, mean or median, or agreeing on a common value using, for example, a 2 or 3 phase commit protocol.

Different applications have different requirements on the aggregation function, and there exist multiple categories for the approach and aggregation function used, as defined by Fasolo et al. [27]. The approach to aggregating data is either with or without size reduction. With size reduction, a node can merge data from packets it receives and then transmits the combined data. Without size reduction, the data points are either unrelated or cannot be combined in a meaningful way; however, a node can still combine the data points into a single packet to reduce overhead. Furthermore, Fasolo et al. classified aggregation functions into duplicate sensitive and duplicate insensitive functions. Duplicate sensitive aggregation functions change the result if some information is considered multiple times; in a duplicate insensitive aggregation function, duplication of information does not affect the result.

Calculating a sum in a clustered network requires few adaptations for an implementation for a non-clustered network, and it can take advantage of the hierarchical communication. Each CH would be able to calculate its cluster's sum of proposed values using a size reduction approach. CHs then repeat the same process during the CH round sharing the aggregated packet with all other CHs. However, one difference when calculating a sum compared to a maximal value is that a CH needs to maintain a list of all contributed values and which node contributed with what value since it is duplicate sensitive. If the nodes were to add values together, then the same value could be added to the sum multiple times. Since A^2 has a restriction on the packet size, the number of nodes that can propose values is limited. Clustering the network would directly help with this since smaller parts of the network calculate separate sums, which means the list of proposed values is smaller.

Furthermore, calculating the median in a clustered network require more resources than calculating a sum, since all values have to be known to at least one node. Clustering affects the size of the required list in each cluster in the same way as when calculating the sum. However, the CH cannot perform a size reducing intermediary calculation as in the sum application. The CHs will instead share the lists between each other, merging all lists from all clusters, which would result in a list with a length equal to the number of nodes in the network. Therefore, calculating the median in a clustered network compared to a non-clustered network does not scale

better with regards to packet size. However, an approach based on aggregation without size reduction would still reduce the overhead required to calculate the median.

However, calculating the mean can make efficient use of a clustered network, since a size reduction approach can be used. A mean calculation requires two things: the sum of all proposed values, and the number of proposed values. Because a CH already knows the size of its cluster, they can perform a mean calculation in the same way as the sum calculation, with the extension that the CHs calculate the sum of the cluster sizes. At the end of the CH round, all CHs know the sum of all proposed values and the sum of all cluster sizes and can, therefore, calculate the mean.

Two other applications, which are more complicated than mean, sum and median, are two and three phase commit. Adapting these to run on a clustered network is not as straightforward. These applications can take advantage of clustering in all phases, running some of the collection and dissemination in parallel in every cluster. However, coordination is required between the CHs when switching between phases. The complexity lies in switching which node is the initiator when switching between inter- and intra-cluster communication; especially if we were to implement a solution which executes the whole commit protocol in one round. We discuss the problems with switching between initiators in a round in Section 5.3.3.

In conclusion, clustering should increase the performance and scalability for many applications running on a WSN. Even applications which require a global view of the network can benefit from hierarchical communication. However, further work is required to implement these applications to evaluate the effectiveness of our clustering implementation.

6.4.4 Achieving Better Scalability

One of the primary goals of this thesis is to increase the scalability of the A^2 system. We do not have a metric for measuring scalability directly, however, using the combined information from all metrics we argue that we achieved our goal. From the latency and energy usage results, we saw that our solution achieved a lower latency and energy usage for networks with 200 nodes. Additionally, since we cluster the network before running the Join service, the restriction caused by the flags field is not applied until after the clusters are created. Thus, we increase the theoretical maximum number of nodes in the network, and the restriction is now local to every cluster.

However, there are some caveats to achieving better scalability. First, we measure the latency results per round. We do not take into account that a clustered network requires two complete rounds to agree on a global maximum value, while A^2 only requires one round. A clustered network requires two rounds because we schedule the inter- and intra-cluster communication in separate rounds. Second, even though our implementation has lower stability than A^2 in almost all cases, we achieve better

latency and lower energy usage; this is good since we achieve those results even in the presence of more faults.

Furthermore, low stability could imply less scalability as well. Our results suggest that the stability decreases as the number of nodes increase, however, we have shown that this is caused by our limitation on fault tolerance and is not inherent to clustering the network.

Despite these caveats, we argue that the results are promising. From the discussion on fault tolerance we conclude that the lower stability for A^2 with clustering can primarily be attributed to limitations on our implementation since some fault tolerance measures present in the A^2 system are not enabled when clustering the network. If future work target this limitation and implement these fault tolerance measures, our clustering implementation has a high probability of achieving equal or better stability while running more scalable and energy efficient networks.

7

Conclusion

Improving scalability and decreasing energy usage in protocols for Wireless Sensor Networks is a research question with much focus. A typical way to accomplish this is to cluster the network. By partitioning the network into separate clusters, the network spends less energy on communication and can, therefore, scale to more nodes. In this thesis, we have introduced clustering to A^2, a system that enables distributed consensus in low powered wireless networks. The clustering implementation is based on the HEED algorithm, which maximises network lifetime by taking into account the residual energy of nodes when electing cluster heads. We have implemented a clustering scheme in the A^2 Synchrotron.

We have evaluated our implementation in the Cooja simulator and on the Flocklab testbed using the metrics stability, reliability, latency, and energy consumption. The results showed that our clustering implementation achieved similar reliability but with higher variance in stability. However, for the larger networks we evaluated, with 200 nodes, our clustering implementation achieved both lower latency and better energy consumption than the A^2 system. We determined that the variance in stability for our clustering implementation was due to our limitation regarding fault tolerance. Future work should, among other things, investigate and implement fault tolerance mechanisms.

7.1 Future Work

In this section, we list and discuss possible goals for future work. With a focus on implementing fault tolerance measures, which we specified as a limitation, and further evaluation to reveal how clustering affects the A^2 system more thoroughly.

7.1.1 Sleep Schemes

Sleep schemes control which nodes wake up each round and which nodes do not. Since forwarders only route packets during CH rounds, sleep schemes are applicable for those nodes. Furthermore, since we showed in Chapter 6 that a clustered network requires less energy than a normal network, sleeping nodes would further decrease the energy consumption of the network proportional to the percentage of nodes put

to sleep; this, however, could be in exchange for reliability. Further work would research ways of determining which nodes can be put to sleep without affecting the stability or reliability of the network.

7.1.2 Transmission Power

In this thesis, all nodes have used the same transmission power. It is conceivable to use a stronger transmission power for CHs that increases with a greater competition radius, enabling all other nodes to sleep during CH rounds. However, it would put a high requirement on dynamic clustering since when normal nodes sleep, CHs will continue to use energy. Dynamic clustering would enable CHs to schedule the Clustering service which should elect new CHs and evenly distribute the energy required by the network.

7.1.3 Fault Tolerance

As we discussed in Section 6.4.1 and specified in our limitation, we did not consider fault tolerance for our implementation. As such, there are several faulty scenarios which our implementation cannot handle. The most important one is the scheduling of the Join service. To properly schedule the Join service locally in each cluster, without affecting the other nodes during CH rounds, would increase stability significantly.

7.1.4 Measuring Throughput

In addition to the metrics we used in this thesis, it would be interesting to measure the throughput the network can handle, and if there is any difference when clustering is applied. Furthermore, performing these evaluations would give us data on how the clustering performs when the packet size is increased. The max application only used a small amount of the maximum packet size, which could affect the results.

7.1.5 Using Other Clustering Algorithms

In our work, we based our clustering implementation on the HEED algorithm. However, we have not analysed the effectiveness of the algorithm itself, and it is possible that another algorithm would produce better results when applied to the A^2 system. Further work could investigate what happens using a different clustering algorithm, with a focus on different classes of algorithms, such as deterministic or centralised algorithms.

Bibliography

[1] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[2] M. A. Mahmood, W. K. G. Seah, and I. Welch, "Reliability in wireless sensor networks: A survey and challenges ahead," *Computer Networks*, vol. 79, no. December, pp. 166–187, 2015.

[3] O. Landsiedel, F. Ferrari, and M. Zimmerling, "Chaos: versatile and efficient all-to-all data sharing and in-network processing at scale," *Proceedings of the 11th ACM Conference on Embedded Netwoked Sensor Systems - SenSys '13r*, pp. 1–14, 2013.

[4] B. A. Nahas, S. Duquennoy, and O. Landsiedel, "Network-wide Consensus Utilizing the Capture Effect in Low-power Wireless Networks," *ACM SenSys*, pp. 1–14, 2017.

[5] M. M. Afsar and M. H. Tayarani-N, "Clustering in sensor networks: A literature survey," *Journal of Network and Computer Applications*, vol. 46, pp. 198–226, 2014.

[6] O. Younis, M. Krunz, and S. Ramasubramanian, "Node clustering in wireless sensor networks: Recent developments and deployment challenges," *IEEE Network*, vol. 20, no. 3, pp. 20–25, 2006.

[7] O. Younis and S. Fahmy, "HEED: A Hybrid, Energy-Efficient, Distributed clustering approach for ad hoc sensor networks," *IEEE Transactions on Mobile Computing*, vol. 3, no. 4, pp. 366–379, 2004.

[8] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - A lightweight and flexible operating system for tiny networked sensors," *Proceedings - Conference on Local Computer Networks, LCN*, pp. 455–462, 2004.

[9] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with COOJA," *Proceedings - Conference on Local Computer Networks, LCN*, pp. 641–648, 2006.

[10] R. Lim, F. Ferrari, and M. Zimmerling, "FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems," *Proceedings of the 12th ...*, pp. 153–165, 2013.

[11] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[12] D. D. V. Nikolaos A. Pantaziz, "A Survey On Power Control Issues In Wireless Sensor Networks," *IEEE Communications Surveys and Tutorials*, vol. 9, no. 4, pp. 86–107, 2007.

[13] Moteiv, *TMote Sky: Ultra low power IEEE 802.15.4 compliant wireless sensor module*, 2 2006. Rev. 1.0.2.

[14] J. Lee, W. Kim, S.-j. Lee, D. Jo, J. Ryu, T. Kwon, and Y. Choi, "An Experimental Study on the Capture Effect in 802.11a Networks," *Proceedings of the ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WinTECH)*, p. 19, 2007.

[15] I. X. S. (e-book collection) and I. X. (e-book collection), *IEEE Std 802.15.4f-2012 (Amendment to IEEE Std 802.15.4-2011): IEEE Standard for Local and metropolitan area networks– Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 2: Active Radio Frequency Identification (RFID) System Physical*. S.l.: IEEE, 2012.

[16] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson, "Epidemic algorithms for replicated database maintenance," *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing - PODC '87*, pp. 1–12, 1987.

[17] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking, "Randomized rumor spreading," *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 565–574, 2000.

[18] P. T. Eugster, R. Guerraoui, A. M. Kermarrec, and L. Massoulie, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[19] M. Perillo, Z. Cheng, and W. Heinzelman, "An analysis of strategies for mitigating the sensor network hot spot problem," *MobiQuitous 2005: Second Annual International Conference on Mobile and Ubiquitous Systems -Networking and Services*, pp. 474–478, 2005.

[20] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella, "Energy conservation in wireless sensor networks: A survey," *Ad Hoc Networks*, vol. 7, no. 3, pp. 537–568, 2009.

[21] E. Ever, R. Luchmun, L. Mostarda, A. Navarra, and P. Shah, "UHEED - an unequal clustering algorithm for wireless sensor networks," *Sensornets 2012*, 2012.

[22] C. Li, M. Ye, G. Chen, and J. Wu, "An energy-efficient unequal clustering mechanism for wireless sensor networks," *2nd IEEE International Conference on Mobile Ad-hoc and Sensor Systems, MASS 2005*, vol. 2005, pp. 597–604, 2005.

[23] S. D. Muruganathan, D. Ma C.F, I. Rolly, Bhasin, and A. O. Fapojuwo, "A Centralized Energy-Efficient Routing Protocol for Wireless Sensor Networks," *IEEE Radio Communicatins*, no. March, pp. 8–13, 2005.

[24] W. B. Heinzelman, A. P. Chandrakasan, S. Member, and H. Balakrishnan, "An Application-Specific Protocol Architecture for Wireless Microsensor Networks," *IEEE Transactions on Wireless Communications*, vol. 1, no. 4, pp. 660–670, 2002.

[25] D. Chronopoulos, *Extreme Chaos : Flexible and Efficient All-to-All Data Aggregation for Wireless Sensor Networks*. M.s thesis, Delft University of Tehcnology, 2016.

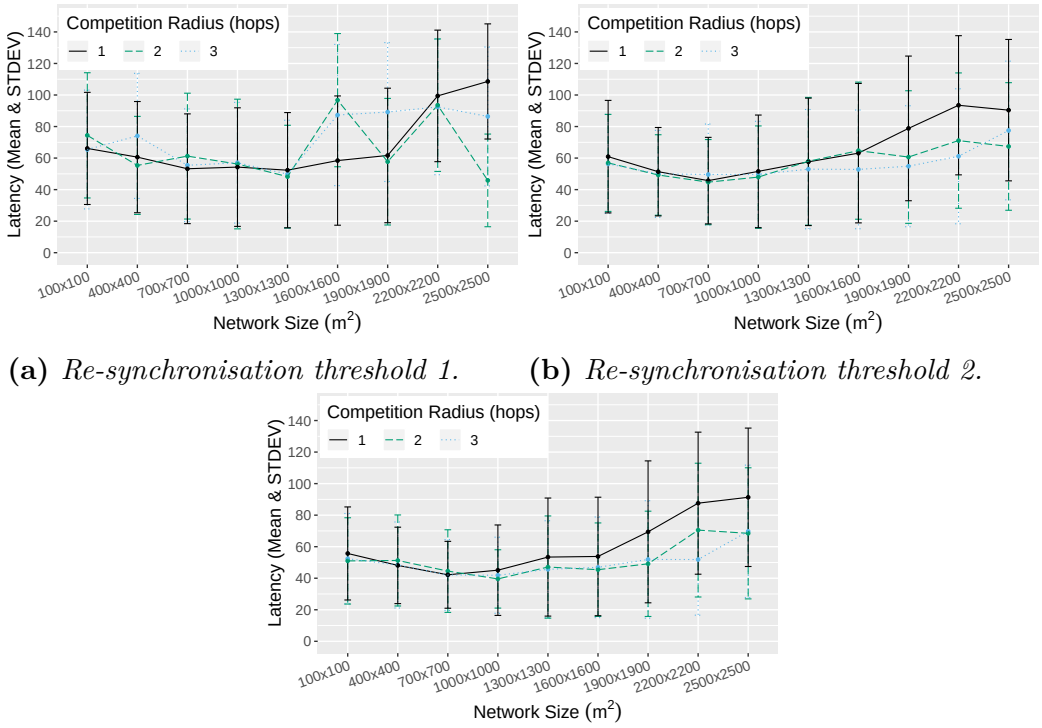
[26] A. L. Liestman and D. Richards, "Toward optimal gossiping schemes with conference calls," *Discrete Applied Mathematics*, vol. 7, no. 2, pp. 183–189, 1984.

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

A

Appendix 1

A.1 Re-synchronization Latency Results

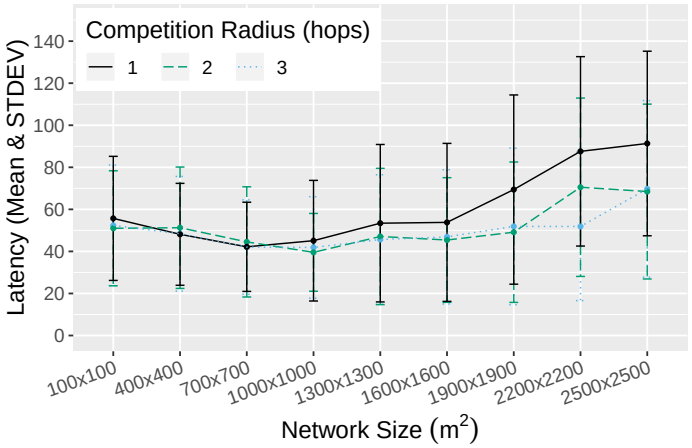


(a) *Re-synchronisation threshold 1.* **(b)** *Re-synchronisation threshold 2.*

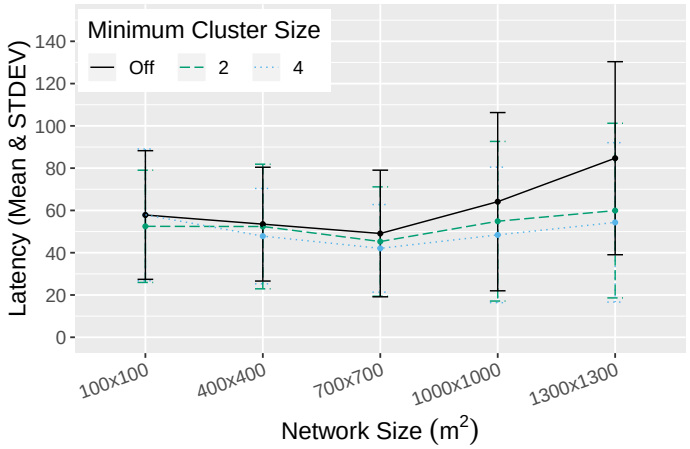
(c) *Re-synchronisation threshold 3.*

Figure A.1: *Competition radii tests for different values of resynchronisation threshold.*

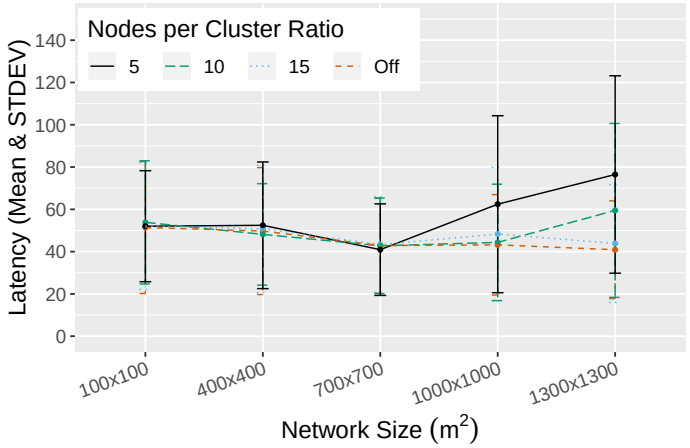
A.2 Parameter Latency Plots



(a) *Competition radius.*



(b) *Minimum cluster size.*



(c) *Nodes per cluster ratio.*

Figure A.2: *The latency results for the parameter tests.*

B

Appendix 2

B.1 Parameter Values for the A^2 Comparison

Table B.1: *List of topologies with 50 nodes and the competition radius used for each topology.*

Competition Radius	Topologies
2	700x700, 1300x1300, 2200x2200
3	100x100, 400x400, 1000x1000, 1600x1600, 1900x1900, 2500x2500

Table B.2: *List of topologies with 200 nodes and the nodes per cluster ratio used for each topology.*

Nodes Per Cluster Ratio	Topologies
10	100x100, 400x400
30	700x700, 1000x1000, 1300x1300