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

Andreas Larsson
Computer Science and Engineering
Chalmers University of Technology
Email: larandr@chalmers.se

Philippas Tsigas
Computer Science and Engineering
Chalmers University of Technology
Email: tsigas@chalmers.se

***Abstract*—Wireless Ad-hoc networks are distributed systems that often reside in error-prone environments. Self-stabilization lets the system recover autonomously from an arbitrary system state, making the system recover from errors and temporarily broken assumptions. Clustering nodes within ad-hoc networks can help forming backbones, facilitating routing, improving scaling, aggregating information, saving power and much more. We present a self-stabilizing distributed (k,r)-clustering algorithm. A (k,r)-clustering assigns k cluster heads within r communication hops for all nodes in the network while trying to minimize the total number of cluster heads. The algorithm assumes a bound on clock frequency differences and a limited guarantee on message delivery. It uses multiple paths to different cluster heads for improved security, availability and fault tolerance. The algorithm assigns, when possible, at least k cluster heads to each node within $O(r\pi\lambda^3)$ time from an arbitrary system configuration, where π is a limit on message loss and λ is a limit on pulse rate differences. The set of cluster heads stabilizes, with high probability, to a local minimum within $O(r\pi\lambda^4 g \log n)$ time, where n is the size of the network and g is an upper bound on the number of nodes within $2r$ hops.**

I. Introduction

Starting from an arbitrary system state, self stabilizing algorithms let a system stabilize to, and stay in, a consistent system state [1]. There are many reasons why a system could end up in an inconsistent system state of some kind. Assumptions that algorithms rely on could temporarily be invalid. Memory content could be changed by radiation or other elements of harsh environments. Battery powered nodes could run out of batteries and new ones could be added to the network. It is often not feasible to manually configure large ad-hoc networks to recover from events like this. Self-stabilization is therefore often a desirable property of algorithms for ad-hoc networks. However, the trade off is that self-stabilization often comes with increased costs. A self-stabilizing algorithm can never stop because it is not known in advance when temporary faults occur. Nevertheless, as long as all assumptions hold, it can converge to stable result, or, after convergence, stay within a set of acceptable states. Moreover, there are often overheads in the algorithm tied to the need to recover from arbitrary system states. They can be additional computations, larger messages, larger data structures or longer required times to achieve certain goals.

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

improved scaling of a large network. Nodes in a cluster could take turns doing energy-costly tasks to reduce overall power consumption.

Clustering is a well studied problem. Due to space constraints, for references to the area in general, we point to the survey of the area with regard to wireless ad-hoc networks by Chen, Liestam and Liu in [2] and the survey by Abbasi and Younis in [3] for wireless sensor networks. In this paper we focus on self-stabilization, redundancy and security aspects. One way of clustering nodes in a network is for nodes to associate themselves with one or more cluster heads. In the (k,r)-clustering problem each node in the network should have at least k cluster heads within r communication hops away. This might not be possible for all nodes if the number of nodes within r hop from them is smaller than k. In such cases a best effort approach can be taken for getting as close to k cluster heads as possible for those nodes. The clustering should be achieved with as few cluster heads as possible. To find the global minimum number of cluster heads is in general computationally hard, and algorithms usually provide approximations. The (1,r)-clustering problem, a subset of the (k,r)-clustering problem, can be formulated as a classical set cover problem. This was shown to be NP complete in [4]. Assuming that the network allows k cluster heads for each node, the set of cluster heads forms a total (k,r)-dominating set in the network. In a *total* (k,r)-dominating set the nodes in the set also need to have k nodes in the set within r hops, in contrast to an ordinary (k,r)-dominating set in which this is only required for nodes not in the set.

There is a multitude of existing clustering algorithms for ad-hoc networks of which a number is self-stabilizing. Johnen and Nguyen present a self-stabilizing (1,1)-clustering algorithm that converges fast in [5]. Dolev and Tzachar tackle a lot of organizational problems in a self-stabilizing manner in [6]. As part of this work they present a self-stabilizing (1,r)-clustering algorithm. Caron, Datta, Depardon and Larmore present a self-stabilizing (1,r)-clustering in [7] that takes weighted graphs into account. Self-stabilization in systems with unreliable communications was introduced in [8]. In [9] a self-stabilizing (k,1)-clustering algorithm, that can cope with message loss, is presented.

There is a number of papers that do not have self-stabilization in their settings. Fu, Wang and Li consider the (k,1)-clustering problem in [10]. In [11] the full (k,r)-clustering problem is considered and both a centralized and a distributed algorithm for solving this problem are presented. Wu and Li also consider the full (k,r)-clustering in [12].

Other algorithms do not take the cluster head approach. In [13], sets of nodes that all can communicate directly with each other are grouped together without assigning any cluster heads. In this paper malicious nodes that try to disturb the protocol are also considered, but self-stabilization is not considered.

A. Our Contribution

We have constructed a self-stabilizing (k, r)-clustering algorithm for ad-hoc networks that can deal with message loss, as long as at least one out of π consecutive broadcasts are successful, and that uses unsynchronized pulses, for which the ratios between pulse rates are limited by a factor λ. The algorithm makes sure that, within $O(r\pi\lambda^3)$ time, all nodes have at least k cluster heads (or all nodes within r hops if a node has less than k nodes within r hops) using a deterministic scheme. A randomized scheme complements the deterministic scheme and lets the set of cluster heads stabilize to a local minimum. It stabilizes within $O(r\pi\lambda^4 g \log n)$ time with high probability, where g is a bound on the number of nodes within $2r$ hops, and n is the size of the network.

We presented the first distributed self-stabilizing (k,r)-clustering in [14]. There, the system settings

assumed perfect message transfers and lock step synchronization of the nodes. The current article is a further development of that work and the main idea of the algorithm is the same. The unreliable communication media, the unsynchronized nodes and the introduction of a veto mechanism to speed up convergence, all have made the current algorithm quite different, yet clearly related to the one in [14]. We present correctness proofs for quick selection of enough cluster heads (k cluster heads within r hops when possible) and that the set of cluster heads converges towards a local minimum and stays at that local minimum. This includes an upper bound on the time it takes, with high probability, for that convergence to happen. Furthermore, we also present experimental results on the convergence of the algorithm and how it copes with changes to the topology.

B. Document Structure

The rest of the paper is organized as follows. In section II we introduce the system settings. Section III describes the algorithm. Section IV gives the overview of the proofs of the algorithm. We discuss experimental results, security and redundancy in Section V.

II. System Settings

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

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

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

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

III. Self-stabilizing Algorithm for (k, r)-clustering

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

The basic idea of the algorithm is for cluster heads to constantly broadcast the fact that they are cluster heads and for all nodes to constantly broadcast which nodes they consider to be cluster heads. The set of considered cluster heads consists

```
1 on pulse:
2   timer ← (timer + 1)  mod T
3   if estate = SLEEP ∧ ∃t s.t. (i, JOIN, t) ∈ smem then state ← HEAD
4   if state = HEAD then (newheads, newslaves) ← ({i}, ∅)
5   else (newheads, newslaves) ← (∅, {i})
6   for each j ∈ {k |k ≠ i ∧ ∃ ki ≠ JOIN,t s.t. (k,ki,t) ∈ smem} do handlestate(j)
7   (heads, slaves) ← (newheads, newslaves)
8
9   /* Escaping */
10  if state ∈ {HEAD, ESCAPING}
11    estate ← updateestate()
12    if estate = INIT ∧ state = HEAD ∧ |heads| > k
13      state ← ESCAPING
14      < heads, slaves>  ← < heads \ {i}, slaves ∪ {i}>
15  if state = ESCAPING ∧ estate = SLEEP
16    if ∃t s.t. (i, JOIN, t) ∈ smem then state ← HEAD
17    else state ← SLAVE
18  if state = SLAVE then < estate,estart>  ← < SLEEP,-1>
19
20   /* Add heads */
21  if |heads| < k
22    heads ← heads ∪ {smallest(k -|heads|, slaves)}
23    slaves ← slaves \ heads
24
25  /* Join and send state */
26  for each j ∈ heads
27    if j ≠ i then sendset ← pruneset(sendset ∪ {< j, JOIN, r, π> })
28    else state ← HEAD
29  smem ← stepmem(smem)
30  < sendset,data>  ← stepset(pruneset(sendset ∪ {< i, state, r, π> }))
31  LBcast(< i,data> )
32
33 function updateestate:
34  if timer = 0 then estart ← uniformlyrandom({0, 1, . . ., T-T_cool-1})
35  if estart ∈ [0,T-T_cool-1]
36    if timer ∈ [estart, estart] then return INIT
37    else if timer ∈ [estart+1, estart+T_flood-1] then return FLOOD
38  return SLEEP
```

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

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

One could imagine an algorithm that in a first phase adds cluster heads and thereafter in a second phase removes cluster heads that are not needed. To achieve self-stabilization however, we cannot rely on starting in a predefined system state. Recovery from an inconsistent system state might start at any time. Therefore, in our algorithm there are no

```
40 function handlestate(j):
41   js ← prioritystate(j,smem)
42   if js = HEAD
43     newheads ← newheads ∪ {j}
44     sendset ← pruneset(sendset ∪ {< j, JOIN, r, π> })
45   else if js = ESCAPING ∧ j ∈ heads
46     if |heads| ≤ k
47       newheads ← newheads ∪ {j}
48       sendset ← pruneset(sendset ∪ {< j, VETO, r, π> })
49     else heads ← heads \ {j}
50   newslaves ← (newslaves ∪ {j}) \ newheads
51
52 function prioritystate(j,mem):
53   if ∃ t s.t. (j, HEAD, t) ∈ mem
54     return HEAD
55   if ∃ t s.t. (j, ESCAPING, t) ∈ mem
56     return ESCAPING
57   return SLAVE
58
59 function stepmem(mem):
60   newmem ← ∅
61   for each < j,js,ttk>  in mem
62     ttk ← min(ttk,κ)-1
63     if ttk > 0
64       newmem ← prunemem(
65             newmem ∪ {< j,js,ttk> })
66   return newmem
67
68 function stepset(set):
69   < newset, newdata>  ← < ∅, ∅>
70   for each < j,ji,ttl,ttf>  in set
71     < ttl, ttf>  ← < min(ttl,r), min(ttf,π)-1>
72     if ttf > 0 ∧ ttl > 0 then
73       newset ← pruneset(newset ∪ {< j,ji,ttl,ttf> })
74     if ttf ≥ 0 ∧ ttl > 0 then
75       newdata ← newdata ∪ {< j,ji,ttl> }
76   return < newset, newdata>
77
78 on LBrecv(< k, infoset> ):
79   for each < j,ji,ttl>  ∈ infoset
80     ttl ← min(ttl,r))
81     if ji = VETO
82       if j = i ∧ state = ESCAPING
83         state ← HEAD
84     else if (j ≠ i ∧ ji ≠ JOIN) ∨ (j = i ∧ ji = JOIN)
85       smem ← prunemem(smem ∪ {< j,ji,κ> })
86     if j ≠ i ∧ ttl > 1
87       sendset ← pruneset(sendset ∪ {< j,ji,ttl-1,π> })
```

Figure 3. Pseudocode for the self-stabilizing clustering algorithm (2/2).

phases and the mechanism for adding cluster heads runs in parallel with the mechanism for removing cluster heads and none of them ever stops.

At each pulse a node sends out its state (the algorithmic state, i.e., which role it takes in the algorithm) and forwards the states of others. A cluster head node normally has the state HEAD and a non cluster head node always has state SLAVE. If a node p_i in any pulse finds out that it has less than k cluster heads it selects a set of other nodes

Constants, and variables:
i : **Constant** *id of executing processor.*
T, T_{cool}, T_{flood}, κ : *Constants derived from r, k,* λ *and* π. *See Definition 4.*
state $\in$ {HEAD, ESCAPING, SLAVE} : *The state of the node.*
timer : *Integer. Timer for escape attempts.*
estart : *Integer. The escape schedule.*
estate $\in$ {SLEEP, INIT, FLOOD} : *State for escape attempts.*
heads, slaves : *Sets of Id:s tracking what nodes have which role.*
smem, sendset, data : *Infotuple sets for keeping and forwarding state data.*

External functions and macros:
LBcast(*m*) : *Broadcasts message m to direct neighbors.*
LBrecv(*m*) : *Receives a message from direct neighbor.*
smallest(*a*,*A*) : *Returns the* **min**(|*A*|,*a*) *smallest id:s in A.*
pruneset(*A*): *maxt* $\leftarrow$ {< *j,ji,ttl,ttf*> $\in$ *A* : *ttl* $= \max_\tau$ {τ : < *j,ji,*τ*,ttf*> $\in A$}}
return {< *j,ji,ttl,ttf*> $\in$ *maxt* : *ttf* $= \max_\phi$ {ϕ : < *j,ji,ttl,*ϕ> $\in A$}}
prunemem(*A*): **return** {< *j,ji,*ξ> $\in$ *A* : $\xi = \max_x$ {*x* : < *j,ji,x*> $\in A$})}

Figure 1. Constants, variables, external functions and macros for the algorithm in Figures 2 and 3.

that it decides to elect as cluster heads. Node p_i then elects established cluster head nodes and any newly elected nodes by sending a *join* message to them. Any node that is not a cluster head becomes a cluster head if it receives a join addressed to it.

We take a randomized approach for letting nodes try to drop their cluster head responsibility. Time is divided into periods of T pulses. A cluster head node p_i picks uniformly at random one pulse out of the $T - T_{cool}$ first pulses in the period as a possible starting pulse, $estart_i$, for an escape attempt. If p_i has more than k cluster heads in pulse $estart_i$, then it will start an escape attempt. When starting an escape attempt a node sets it state to ESCAPING and keeps it that way for a number of pulses to make sure that all the nodes in G_i^r will eventually know that it tries to escape. A node $p_j \in G_i^r$ that would get fewer than k cluster heads if p_i would stop being a cluster head can veto against the escape attempt. This is done by continuing to regard p_i to be a cluster head and send a VETO back to p_i. If p_j, on the other hand, has more than k cluster heads it would not need to veto. Thus, by accepting the state of p_i as ESCAPING, p_j will not send any join to p_i. After a number of pulses all nodes in $\hat{G}_i^r$ will have had the opportunity to veto the escape attempt. If none of them objected, at that point p_i will get no joins and can set its state to SLAVE.

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

The pseudocode for the algorithm is described in Figures 2 and 3 with accompanying constants, variables, external functions and macros in Figure 1. At each pulse of a node the lines 1-31 are executed resulting in a message that is broadcast at some time before the next pulse of that node. When a message is being received, the lines 78-87 are executed.

IV. Correctness

In Section IV-A we show some basic results that we use further on. In Section IV-B we will show that within $O(r\pi\lambda^3)$ time we will have $|C_i^r| \geq k_i$ for any node p_i. First we show that this holds while temporarily disregarding the escaping mechanism, and then that it holds for the general case in Theorem 1. In Section IV-C we will show that a cluster head node p_i can become slave if it is not needed and if it tries to escape undisturbed by other nodes in G_i^{2r}. We continue to show that the set of nodes converges, with high probability, to a local minimum in $O(r\pi\lambda^4 g \log n)$ time in Theorem 2.

Definition 1. *When all system assumptions hold from a point s in time and forward, we say that "we have a legal system execution from s". We denote a pulse of p_i with Γ_x^i for some integer x. Consecutive pulses of p_i have consecutive indices, e.g., Γ_x^i,*

Γ^i_{x+1}, Γ^i_{x+2}, *etc. We denote the time between* Γ^i_x *and* Γ^i_{x+1} *with* γ^i_x.

Definition 2. *We define the set of* states *as* $\{SLAVE, HEAD, ESCAPING\}$. *An* infotuple *is a tuple* (j, js, ttx) *or* (j, js, ttl, ttf), *where* js *is a either a state or one of* $\{VETO, JOIN\}$ *and is said to be* for node p_j *regardless if* p_j *is the original sender or final receiver of the infotuple. The* ttx *field can either be a* ttl, *the number of hops the info is to be forwarded, or a* ttk, *the number of pulses for which the infotuple should be kept in* $smem$ *before being discarded. A* ttf *field denotes the number of resends that is left to be done for that particular tuple.*

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

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

Definition 3. *A node* p_i *is said to* handle *a state* σ *for a node* p_j *in a pulse* Γ^i_x *when the* $handlestate$ *function is called with parameter* j *at line 6 and the subsequent call to the* $prioritystate$ *with* j *as a parameter returns* σ, *setting* $js_i = \sigma$ *at line 41.*

A. Basic properties

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

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

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

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

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

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

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

Lemma 2. *Assume that we have a legal system execution from time* $s - r(\pi - 1)\lambda$ *and consider a node* p_i *that has a pulse* Γ^i_x *at time* s. *Now, assume that* p_i *has* (k, σ, r, π) *in* $sendset_i$ *just*

before executing line 30 in Γ_x^i and consider a node $p_j \in G_i^r$, $p_j \neq p_i$ and a time interval $\hat{I} = [s - r(\pi-1)\lambda, s + r(\pi+1)\lambda]$.

First, if (k, σ, τ') is $relevant_j$, there exist a pulse $\Gamma_y^j \in \hat{I}$ so that (k, σ, τ') is received in γ_{y-1}^j, for a $\tau' \geq 1$.

Second, if (k, σ, τ') is $memorable_j$, then $(k, \sigma, \kappa) \in smem_j$ in Γ_y^j just before executing line 2.

Lemma 3. *Assume that we have a legal system execution from time $s - r(\pi-1)\lambda$ and consider a node p_i that has a pulse Γ_x^i at time s. If $p_j \in G_i^r$, then $i \in slave_j \cup head_j$ from time $s + r(\pi+1)\lambda$ and forward.*

Lemma 4. *Assume that we have a legal system execution from time s and consider a node p_i that has a pulse Γ_x^i at time s and that has (i, σ, r, π), where σ is a state, in $sendset_i$ just before executing line 30 in Γ_x^i. Assume that p_i do not add (i, σ, r, π) to $sendset_i$ in the time interval $I = (s, s + a)$, for an $a > s + r(\pi+1)\lambda + (\kappa-1)\lambda$. In other words, p_i does not execute line 30 with $state_i = \sigma$ in any pulses or between any pulses in the time interval I. Consider any node p_j in the network. Under these assumptions, the last pulse Γ_y^j in the time interval I for $p_j \in \hat{G}_i^r$ such that p_j receives (i, σ, τ) for any τ in γ_{y-1}^i, can not be later than $s + r(\pi+1)\lambda$. Furthermore, for any pulse of p_j in the time interval $(s + r(\pi+1)\lambda + (\kappa-1)\lambda, s + a)$, and between those pulses, $(i, \sigma, \xi) \notin smem_j$ for any ξ.*

Lemma 5. *Let $\hat{s} = s - r(2\pi+1)\lambda^2$ and assume that we have a legal system execution from time $\hat{s}$, and consider a node p_i that has a pulse Γ_x^i at time s and that has (k, σ, r, π) in $sendset_i$ just before executing line 30 in Γ_x^i. Assume that $(k, \sigma, \cdot)$ is $memorable_j$. Then, there exists a pulse $\Gamma_y^j \in I = [s + r(\pi+1)\lambda - 1, s + r(\pi+1)\lambda + \lambda - 1]$ in which $\exists \xi > 0$ such that $(k, \sigma, \xi) \in smem_j$.*

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

Third, if $\Sigma_{\sigma'}$ holds for all σ' that have a higher priority than σ, then p_j handles σ for p_i in Γ_y^j (see Definition 2).

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

Corollary 1. *Assume that we have a legal system execution from time $s - \lambda$ and consider a node p_i that has a pulse Γ_x^i at time s. Let $\chi = \lceil r(2\pi+1)\lambda^2 + r(\pi+1)\lambda + 2\lambda \rceil$. Now assume that a node p_i, in each of the pulses Γ_x^i–$\Gamma_{x+\chi-1}^i$, adds (i, σ, π) to*

$sendset_i$ *and does not add* (i, σ', π) *to* $sendset_i$ *for any* $\sigma \neq \sigma'$*. Then, for any node* $p_j \in G_i^r$*,* p_j *has a pulse* Γ_y^j *at a time* $\hat{s} = s + r(2\pi + 1)\lambda^2 + r(\pi + 1)\lambda - 1 + t$ *for a* $t \in [0, 2\lambda]$*, in which* p_j *handles* σ *for* p_i*. Furthermore,* $\hat{s}$ *happens between the execution of the pulses* Γ_x^i *and* $\Gamma_{x+\chi-1}^i$

B. Getting Enough Cluster Heads

This section builds up a case showing that the algorithm will elect enough cluster heads. We show how cluster heads are elected in Lemmas 6, 7 and 8. In Lemma 9 we take a look at how the escape mechanism works. Finally, in Theorem 1, we put it all together and show that within $O(r\pi\lambda^3)$ time from starting a legal execution, each node p_j in the network will get k_j cluster heads within r hops.

Definition 5. *For a node* p_i *to be a* cluster head *is equivalent to* $state_i \in \{HEAD, ESCAPING\}$*. For a node* p_i *to be a slave is equivalent to* $state_i = SLAVE$*. For a node* p_j*, we define* C_j^r *as the set of cluster heads in* G_j^r*. Furthermore, we define* H_x *to be the set of cluster heads in the network at time* x*.*

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

Lemma 6. *Assume that we have a legal system execution from time* s*. Assume that, for all nodes in the network, their* $state$ *can never be ESCAPING, their* $estate$ *is always SLEEP and lines 10-18 are never executed. Under these assumptions, any node* p_i *will have* k_i *cluster heads within* r *hops from time* $s + (3r\pi + 2)\lambda$ *and forward.*

Now we consider the full escape mechanisms and show that a node that receive joins remains or becomes a cluster head.

Lemma 7. *Assume that we have a legal system execution from time* $s - \lambda$ *and consider a node* p_i *that has a pulse* Γ_x^i *at time* s*. If* p_i *has* $(i, JOIN, \xi) \in smem_i$*, for some* $\xi > 0$ *before executing line 2, then* p_i *is a cluster just before executing line 4 in* Γ_x^i *and will stay cluster head throughout the rest of* Γ_x^i *and throughout* γ_x^i*. Furthermore, if node* p_i *is a cluster head just before executing line 2 in* Γ_x^i*, then it is a cluster head throughout the entire* Γ_x^i*.*

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

Lemma 8. *Assume that we have a legal system execution from time* s*. Assume that a node* $p_j \neq p_i$ *wants a node* $p_i \in G_j^r$ *to be cluster head as soon as it knows about it and is never willing to let it escape. In other words, (1) if* $i \in slaves_j$ *and* $i \notin heads_j$ *after line 7 in a pulse* Γ_y^j*, we assume that the condition in line 21 holds in* Γ_y^j *and that* i *is added to* $heads_j$ *at the execution of line 22, and (2) the condition in line 46 would hold when* $handlestate$ *is called with* i *as a parameter in any pulse of* p_j*.*

Under these assumptions, there exists a pulse Γ_x^i *such that* $state_i \neq SLAVE$ *after executing line 3 in* Γ_j^i*, and such that* $state_i \neq SLAVE$ *in any* $\Gamma_{x'+1}^i$ *or* $\gamma_{x'}^i$ *for any* $x' \geq x$*.*

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

Definition 6. *A node* p_i initiates an escape attempt *in a pulse* Γ_x^i *if the condition holds in line 12 and lines 13–14 are executed in* Γ_x^i.

Lemma 9 shows that the escape mechanism works, that a cluster head that is not needed can escape that responsibility.

Lemma 9. *Assume that we have a legal system execution from time* $s - T_{cool}$ *and consider a node* p_i *that initiates an escape attempt in a pulse* Γ_x^i *at time* s.

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

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

Theorem 1 shows that, within time $T_{cool} + (5r\pi + 4)\lambda \in O(T_{flood}\lambda) = O(r\pi\lambda^3)$ from an arbitrary configuration, all nodes p_i have at least k_i cluster heads within r hops and that the set of cluster heads in the network can only stay the same or shrink from that point on. From Corollary 1 we get the factor $O(\lambda^2)$ time for a node to know that it has reached out. This theorem introduces another factor of $O(\lambda)$ because a node needs to be sure that another node has finished something as discussed previously.

Theorem 1. *Assume that we have a legal system execution from time* s. *Then any node* p_j *will have* k_j *cluster heads from time* $s + T_{cool} + (3r\pi + 2)\lambda$ *and onward. Moreover, a node that is not in* H_t *for a time* $t \geq s + T_{cool} + (5r\pi + 4)\lambda$ *can not be in* $H_{t'}$ *for a* $t' \geq t$ *and consequently* $|H_{t'}| \leq |H_t|$ *for any* $s + T_{cool} + (5r\pi + 4)\lambda \leq t \leq t'$.

C. Convergence to a Local Minimum

Lemma 9 shows that a cluster head node that is not needed can escape the cluster head responsibility if it does not interfere with escape attempts by other nodes. This section shows that the set of cluster heads converges to a local minimum. We first show that an unneeded cluster head node can escape, with high probability (Lemma 10) in $O(T\lambda) = O(gr\pi\lambda^4)$ time. The extra λ is due to the usual reason and $T \in O(gr\pi\lambda^3)$. The factor g is a bound on the number of nodes that could interfere with a given escape attempt. It is part of T to give a node a constant probability of escaping in its one try in a period of T time (given that it is not needed as a cluster head).

Lemma 10. *Assume that we have a legal system execution from time* s *and consider a node* p_i *that is a cluster head. Assume that* $|C_j^r| > k$ *holds for all nodes* $p_j \in \hat{G}_i^r$ *from time* $s+T_{cool}+(5r\pi+4)\lambda$ *and as long as* p_i *remains a cluster head. Then, node* p_i *will have* $state_i =$ *SLAVE after time* $s+T_{cool}+(5r\pi+4)\lambda+(\beta+1)T\lambda$ *with at least probability* $1 - 2^{-\beta}$.

Theorem 2, shows that with high probability the entire network reaches a local minimum within $O(r\pi\lambda^4 g \log n)$ time.

From Theorem 1 we got that all nodes p_i have at least k_i cluster heads within r hops in $T_{cool} + (3r\pi + 2)\lambda$ time after an arbitrary configuration.

Theorem 2 shows that with at least probability $1-2^{-\alpha}$ the set of cluster heads in the network stabilizes to a local minimum within $s+T_{cool}+(5r\pi+4)\lambda+(\alpha+\log n+1)T\lambda$ time. The factor $O(\log n)$ is multiplied by the result from Lemma 10 because we go from probabilistic guarantee that one specific node gets an uninterrupted escape attempt to that all cluster heads get such an attempt. And the number of cluster heads is bounded by the number, n, of nodes in the network.

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

V. Discussion

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

In addition, we performed experiments on recovery from small changes to the topology from a converged system state. The convergence times from a newly started network ("Start") is compared in Figure 4 with the convergence times after a change to a initially converged network. We investigate 10% added nodes ("Add"), 10% removed nodes ("Remove") or 10% moved nodes ("Move").

We can see that the least obtrusive change to the topology is removed nodes. The chance is good

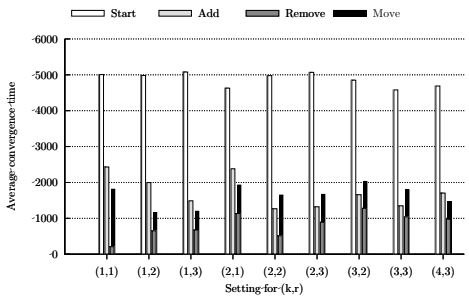

Figure 4. Convergence times from a fresh start, after 10% node additions, after 10% node removes and after 10% node moves.

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

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

The multiple paths can also give applications higher security if some nodes in the network can be compromised. If there is at least one path of at most r hops between a node p_i and a node p_j that is not passing through any compromised nodes then the flooding makes sure that node p_i and p_j gets to know about each other. Moreover, if p_j wants p_i to be cluster head then the compromised nodes cannot stop that. If nodes add information to the

messages about the paths they have taken during message forwarding then the nodes get to know about the multiple paths. With this knowledge they can in an application layer use as diverse paths as possible to communicate with their cluster heads. Thus even if a compromised node is on the path to one cluster head and drops messages or do other malicious behavior there can be other cluster heads for where there is no compromised nodes on the chosen paths.

Consider a compromised node p_c that can lie and not follow protocol. First assume that p_c cannot introduce node id:s that does not exist (Sybil attacks, [15]) or node id:s for nodes that are not within G_c^r (wormhole attacks, [16]) and that p_c cannot do denial of service attacks. Then p_c can make any or all nodes within G_c^r become and stay cluster heads by sending joins to them or having them repeatedly go on and off cluster head duty over time by alternating between sending joins and letting the node escape. Consider a node p_i that is a cluster head and has a path to a node p_j of length $\leq r$ hops that does not pass through p_c. In this situation p_c can not give the false impression that p_i is not a cluster head as HEAD takes precedence over ESCAPING that takes precedence over SLAVE at message receipt. If p_c on the other hand is in a bottleneck between nodes without any other paths between them then it can lie about a node p_ℓ being a cluster heads and refuse to forward any joins to p_ℓ. Now if we assume that p_c is not restricted in what id:s it can include in false messages it can convince a node p_ℓ that nodes not in G_ℓ^r are cluster heads. In the worst case it can eventually make p_ℓ rely exclusively on non-existent cluster heads with paths that all go through p_c. In any case the influence by a compromised node p_c is contained within G_c^{2r} as the maximum ttl of a message is r and is enforced at message receipt.

VI. Conclusions

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

VII. Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 257007.

This work, in a shortened form, appeared in Andreas Larsson and Philippas Tsigas. “Self-stabilizing (k,r)-clustering in clock rate-limited systems.” In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2012)*, Springer, 2012.

References

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

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

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

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

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

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

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

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

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

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

[11] M. A. Spohn and J. J. Garcia-Luna-Aceves, “Bounded-distance multi-clusterhead formation in wireless ad hoc networks,” *Ad Hoc Netw.*, vol. 5, no. 4, pp. 504–530, 2007.

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

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

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

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

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