

# A Self-stabilizing Link-Cluster Algorithm in Mobile Ad Hoc Networks

Doina Bein<sup>1</sup>    Ajoy K. Datta<sup>1</sup>    Chakradhar R. Jagganagari<sup>1</sup>    Vincent Villain<sup>2</sup>

<sup>1</sup> *School of Computer Science, University of Nevada Ls Vegas*  
*{siona,datta,jagganag}@cs.unlv.edu*

<sup>2</sup> *LaRIA, Université de Picardie Jules Verne, France*  
*villain@laria.u-picardie.fr*

## Abstract

We design a self-stabilizing cluster routing algorithm based on the link-cluster architecture of wireless ad hoc networks. The network is divided into clusters. Each cluster has a single special node, called clusterhead that contains the routing information about inter and intra-cluster communication. The proposed algorithm assumes that all nodes have unique IDs. The algorithm achieves two tasks. First, the set of special nodes (clusterheads) is elected such that it models the link-cluster architecture: any node belongs to a single cluster, is within two hops of the clusterhead, and knows the direct neighbor on the shortest path toward the clusterhead. Second, the routing tables are maintained by the clusterheads to store information about nodes both within and outside the cluster. There are two advantages of maintaining routing tables only in the clusterheads. First, as no two neighboring nodes are clusterhead (as per the link-cluster architecture), there exists no consistency problems. Second, since other nodes are responsible for forwarding only, they use less power. So, when the CH runs out of power, some neighboring node will be available to take on the task.

A self-stabilizing system [3] has the ability to automatically recover to normal behavior in case of transient faults, without a centralized control. The MANET can start in some arbitrary state and with no knowledge of the network topology, but still eventually selects a set of clusterhead nodes (as specified by the link-cluster architecture) in a constant amount of time  $(2(\text{time\_period} + 2) + n(\text{time\_period} + 1))/2$  time units, where  $n$  represents the total number of nodes in MANET). Then in these special nodes, routing tables are built with information about shortest paths for intra-cluster routing and shortest paths for inter-cluster routing (based on on-demand set of nodes).

**Keywords:** Cluster routing, clusterhead election, link-cluster architecture, self-stabilization.

## 1. Introduction

Mobile ad hoc networks (MANET) consist entirely of mobile nodes that communicate on-the-move without central management. In MANET, every node acts both as a host (generates user and application traffic) and a router (carries out network control and routing protocols). In this paper, we model MANET as undirected unweighted networks (the distance is measured in number of hops).

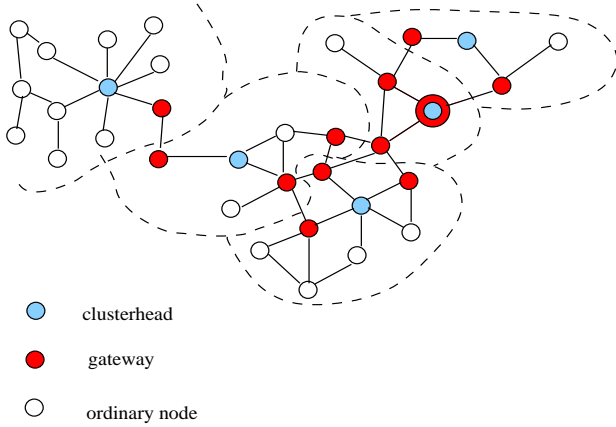
A wireless ad hoc network consists of nodes that move freely and communicate with each other using wireless links. A way to support communication between nodes in a wireless ad hoc network is to select a set of special nodes that can act as a backbone for the entire communication (called *clusterheads*). Thus, the nodes are grouped into *clusters*, and each cluster contains a single *clusterhead* responsible for managing the routing [12, 9]. A *gateway* is a node that has at least two direct links to nodes that belong to different clusters. The nodes that are neither clusterheads nor gateways are called the *regular* nodes.

For example, in Figure 1, an ad hoc network is divided into five cluster based on link-cluster architecture. The clusterheads are shaded. Observe that a clusterhead can act as a gateway as well.

As nodes can move freely and new nodes can join or leave, the set of backbone nodes need to change to reflect the topology changes in the network. The selection of backbone nodes must be fast, but also should require as little communication as possible, since mobile nodes are often powered by batteries.

In link-cluster architecture, a non-clusterhead node is within two hops from its clusterhead. There are two advantages of this model. Scalability is improved since a reduced number of mobile nodes participate in some routing algorithm, hence a low routing-related control overhead. The chance of interference via coordination of data transmissions is lower.

We use the concept of *self-stabilization* ([1, 3, 4, 8, 6]) to design a self-stabilizing link-cluster routing protocol, called



**Figure 1. Ad hoc network divided in 5 clusters**

*CCA.* A *self-stabilizing* system has the ability to automatically recover to normal behavior in case of transient faults. Regardless of the system starting state (initial state of the nodes and initial messages in the channels) and without a centralized control, a set of clusterhead nodes is selected (as specified by the link-cluster architecture) and in these special nodes build correct routing tables with shortest paths for intra-cluster routing and shortest paths for inter-cluster routing (based on on-demand set of nodes). Being self-stabilizing, our algorithm can deal with the topology changes as well.

**Related Work.** For a survey of routing protocols for clustered architecture networks in MANET refer to [12, 10]. One way to improve the routing efficiency is to route packets alternately between clusterheads and gateways. That is, a packet will be routed via pairs of (clusterhead, gateway), and finally reaches its destination clusterhead. This routing scheme is called *CGSR* [2]. In [2], various token schedule schemes are used to improve the routing efficiency. *CGSR + PTS* scheme gives upstream nodes (the neighbors from which a packet has been received) higher priority to get a permission token to forward packets to clusterhead nodes in such a way that the packets are sent with least delay. On the other hand, some heuristics can be used for gateways to improve packet delivery from clusterheads to gateways, rather than using random scheduling. In *CGSR + PTS + GCS* scheme, packets are transmitted through clusterheads and gateways alternately. Higher priority is given to upstream clusterhead of a gateway after this gateway transmits a packet to its down-stream clusterhead. The gateway must switch its code to hear the upstream clusterhead in order to receive a packet after it sends out a packet to its downstream clusterhead. Similarly, the gateway will switch its code to the downstream clusterhead in order to receive the permission token to forward the packet after it receives a packet

from its upstream clusterhead.

A novel access scheme for clustered environments using the link-cluster architecture called two-hop polling (2HP) is proposed in [5]. Two-hop polling manages to utilize inter-cluster links leading to better connectivity and throughput.

The cluster size selection to achieve an optimum and efficient routing is studied in [11].

**Contribution.** The disadvantage of the existing schemes ([9, 2, 12, 5, 10, 11]) is that they are not stabilizing, i.e., they all work assuming correct initialization. Even if the models are fault-tolerant (i.e., the scheme may adapt to free movement of nodes, joining and/or leaving of existent nodes), they cannot cope with situations where due to a wrong initialization, some nodes are taken as clusterheads while they are not, or having two clusterheads adjacent to the same link.

Certain nodes in the ad hoc network are elected as clusterheads. There are two basic selections of the clusterheads: lowest-ID algorithm and highest-connectivity (degree) algorithm [7]. In our algorithm, when two neighboring nodes are potential candidates to become clusterheads (or are already clusterheads), the one with the higher ID is chosen as the clusterhead and the other node becomes a regular node.

We propose a self-stabilizing cluster routing algorithm for MANET based on link-cluster architecture. The algorithm selects the clusterheads, and then builds in those nodes routing tables for nodes within and outside the cluster. For intra-cluster routing, the shortest paths are maintained. For inter-cluster routing, we implement routing on-demand (the shortest paths are maintained only for the nodes that need to send packets). The algorithm is self-stabilizing: it copes with wrong initialization, and it adapts to free movement of nodes, joining and/or leaving of existent nodes.

**Outline of the paper.** In Section 2, we describe our model and self-stabilization. Section 3 contains the self-stabilizing link-cluster algorithm. Because of lack of space, simulations (some illegal configurations followed by how our algorithm corrects each of these situations) and the proof of correctness are omitted. The paper ends with concluding remarks in Section 4.

## 2. Preliminaries

We use the asynchronous message-passing system model. The asynchronous systems are the most common systems, and the hardest to design algorithms for. Every node can execute its code at its own pace, and the message delivery can take an arbitrary time. We assume an upper bound on the message delivery time, called *timeout*, after which the message is considered lost. In order to compute the time complexity, we assume an upper bound (called, a *time unit*) on the message transmission time over a link.

Each node starts with a unique ID, and knows and distinguishes its direct neighbors. The variable  $N_i^1$  representing the one-hop neighborhood of node  $i$  is maintained by an underlying local topology maintenance protocol that adjusts its value in case of topological changes in the network due to failures of nodes, links, or both.

All nodes execute the same distributed program (*uniform*). The program is a finite set of *guarded actions* of the form:  $\langle label \rangle :: \langle guard \rangle \rightarrow \langle statement \rangle$

A statement can be executed if and only if its guard, a boolean expression, evaluates to *true*. The selected statement is executed in one atomic step. If a process has at least one true guarded command, then it is called *enabled*.

We consider a *distributed daemon*: In every execution step, if one or more processes are enabled, then the daemon chooses at least one (possibly more) of these enabled processes to execute. Once the process is selected, then non-deterministically one of its enabled actions is selected and its statement is executed. We assume a *weakly fair* daemon: A continuously enabled process will be eventually chosen by the daemon.

Each node has a local state defined by its current variables. The global state of the system (configuration) is the union of the local state of its nodes and the messages on the links. An execution  $e$  is a maximal sequence of configurations,  $e = c_1, c_2, \dots$  such that  $\forall i \geq 1, c_i \in \mathcal{C}$ , and  $c_i$  is reached from  $c_{i-1}$  by executing some guarded action, or  $c_i$  is a terminal configuration (no nodes are enabled).

Given  $\mathcal{C}$ , the set of all possible states, and a predicate  $\mathcal{P}$  over  $\mathcal{C}$ , we denote by  $\mathcal{L}_{\mathcal{P}} \subseteq \mathcal{C}$  the set of all *legitimate states with respect to  $\mathcal{P}$* , or simply, the set of all *legitimate states*.

A set  $S \subseteq \mathcal{C}$  is called *closed* if any execution starting from a state in  $S$  reaches a state in  $S$ . Let  $C_1$  and  $C_2$  be subsets of  $\mathcal{C}$ .  $C_2$  is a closed attractor for  $C_1$  if and only if for any initial state  $c_1$  in  $C_1$ , for any execution  $e$  in  $E_{c_1}$  ( $e = c_1, c_2, \dots$ ), there exists  $i \geq 1$  such that for any  $j \geq i$ ,  $c_j \in C_2$ , and  $C_2$  is closed.

We now define self-stabilization.

**Definition 1 (Self-Stabilization)** *A system  $S$  is called self-stabilizing if and only if there exists a predicate  $\mathcal{P}$  such that  $\mathcal{L}_{\mathcal{P}}$  is a subset of legitimate states and  $\mathcal{L}_{\mathcal{P}}$  is a closed attractor for  $\mathcal{C}$ .*

### 3. Self-stabilizing Link-Cluster Algorithm (LCA)

For short, CH means clusterhead node.

In our algorithm we assume and we implement that a non-clusterhead node belongs to a single cluster. The clusterhead routing table contains entries regarding the nodes in its cluster (or clusterhood) and nodes outside its clusterhood. The routing table is updated whenever some clusterhead

election takes place and whenever changes occur related to paths existent in the routing table. The gateways and regular nodes have no routing table. The only routing information they have is a variable indicating the neighbor on the shortest path toward their clusterhead.

For intra-cluster routing, each CH keeps data in its routing table about its clusterhood. This data is collected in Module *Election* (Algorithm 3.1) using *CLREQ* messages. These messages are periodically sent by a non-CH to check the status of its own CH and the path to it.

For inter-cluster routing, either before sending a packet or a block of packets to a destination that is outside the cluster or at specific intervals of time, *ARP* messages are broadcast by the sender (a simple flooding). The *ARP* messages are small in size and are designed solely for path discovery and maintenance. When such messages are received by some non-CH, it broadcasts to its neighbors. When such messages are received by a CH, in Module *Routing* (Algorithm 3.2), the path from the sender to the current node updates the (*Macro Update*). Once the destination is reached or the CH of the destination, an *ACK* message is sent back to the sender containing the shortest path currently detected inside the message. When the *ACK* message reaches the sender, it can now send the data packets following that path.

Every node  $i$  uses several variables:  $c.i$  (the clusterhead ID of node  $i$ ),  $d.i$  (the distance to  $c.i$ ; the set of values is  $\{0, 1, 2\}$ ) and  $n.i$  (the neighbor of  $i$  toward  $c.i$ ).

The distributed program, executed in every node, is divided into two modules. Module *Election* (Algorithm 3.1) contains the actions related to selection of CHs, and creating and/or updating entries in the routing tables in each CH regarding intra-cluster routing. Module *Routing* (Algorithm 3.2) is responsible for creating and/or updating entries in the routing tables in each CH. The non-CHs (gateways or ordinary nodes) do not have a routing table. The only routing data they have is the variable  $n.i$  indicating the neighbor toward the node's CH.

The routing table of a CH will contain entries for its clusterhood, and all other CHs. An entry will contain the destination ID regardless of it is part of the current cluster (if it shares the same CH) or not. If the destination is inside the same cluster, a node maintains the complete path, the length (in number of hops) of the path, and the neighbor on the path toward the destination. If the destination is in another cluster, a node keeps only the neighbor on the (shortest) path, the length of the path, and the neighbor on the path. The macro *Update(sender, dest, ch, path, length)* is responsible for maintaining the routing tables. The parameter  $ch$  represents the CH of the sender  $sender$ . If  $dest == ch$ , the sender and the destination  $dest$  are in the same cluster.

### 3.1. Election

Module *Election* (Algorithm 3.1) contains guards regarding the election of clusterheads. When selecting the CHs, such that eventual topology changes are reflected, three conditions have to be respected: each node belongs to a single CH, each non-CH is within two hops to its CH, and there are no adjacent CHs.

Messages *CL\_ANN*, *CL\_REQ* and *CL\_REJ* contain the following fields: *sender* (sender ID), *dest* (destination ID), *hops* (either the number of hops the message went or the number of hops the message went - 1).

CHs broadcast periodically *CL\_ANN* messages to the nodes within two hops neighborhood. Based on those messages, nodes select or change clusterheads (set *c.i* variable), updates their *n.i*, *d.i* variables, CHs give up their CH status (and broadcasts *CL\_REJ* messages).

Also non-CHs broadcast periodically *CL\_REQ* messages to make sure that their CH are still CH and if the CH is at two hops distance, it still exists (if it is at one hop distance, it can be easily verified by condition  $c.i \in N_i^1$ ).

Action *E.01* periodically (*time\_period*) checks the clusterhead of node *i*. The value of *time\_period* is dependent on the time unit of the network, and has to be at least four time units, such that there is enough time for a message to travel back and forth at the distance of two hops. In case the node sees itself as a clusterhead (condition  $c.i == ID_i$ ), then *i* sets its *n.i* to  $\perp$ , *d.i* to 0, and an announcement message *CL\_ANN* is broadcast (reaching only the nodes within two hops).

If the node is not a clusterhead and the node's clusterhead is not within the two hops neighborhood, then the node elects itself as a clusterhead. The subsequent actions (Actions *E.02-07*) refer to announcement, request and/or rejection messages exchanged only between direct neighbors. Whenever a message is received, the sender is checked to be other than the node itself. For the sake of simplicity of the code, we omit the following test: if condition  $sender == ID_i \vee dest \neq ID_i$  is *true* then discard the message, otherwise accepts it and processes it further. Instead we characterize the message as "proper" (if condition  $sender \neq ID_i \wedge dest == ID_i$  is *true*).

If a *CL\_ANN* message is received by some node (Actions *E.02* and *E.03*), then it must have originated from the node's own clusterhead or from another clusterhead. If the message came from the node's clusterhead, then *n.i* is set to *nb* (the node always stores in *n.i* the direct neighbor from which a message was received). If the message has traveled less than two hops, then it is relayed further. If *i* is a clusterhead and the message came from a clusterhead with a higher ID, then *i* gives up its clusterhead status and broadcasts *CL\_REJ* messages to announce its decision (Action *E.02*). If the current node is a clusterhead but it did not

---

#### Algorithm 3.1 Module *Election*

---

##### Predicates:

$is\_CH(i) \equiv (c.i == ID_i \wedge n.i == \perp \wedge d.i == 0)$   
 $better\_CH(i) \equiv (c.i == ID_i \wedge (n.i == c.i \vee sender > ID_i))$   
 $some\_CH(i) \equiv (c.i \neq ID_i \wedge ((c.i \in N_i^1 \wedge is\_CH(c.i)) \vee (n.i \in N_i^1 \wedge c.(n.i) == c.i)))$   
 $no\_CH(i) \equiv (c.i \neq ID_i \wedge (c.i \notin N_i^1 \vee c.i \neq c.(n.i)))$

##### Actions:

*E.01* *time\_period*  $\longrightarrow$   
 if ( $c.i == ID_i$ ) then  
   if ( $n.i \neq \perp$ ) then  $n.i = \perp$   
   if ( $d.i \neq 0$ ) then  $d.i = 0$   
   send *CL\_ANN*(*sender*, *j*, 1) to all  $j \in N_i^1$   
 else  
   if *some\_CH*(*i*) then send *CL\_REQ*(*ID<sub>i</sub>*, *c.i*, 0) to *n.i*  
   else  $c.i = n.i = ID_i$

*E.02* rcv proper *CL\_ANN*(*sender*, *dest*, 0) from *nb*  $\longrightarrow$   
 if ( $sender \neq nb$ ) then discard it  
 else  
   if ( $sender == c.i$ ) then  
     if ( $n.i \neq c.i \vee d.i \neq 1$ ) then  $n.i = sender; d.i = 1$   
     send *CL\_ANN*(*sender*, *j*, 1) to all  $j \in N_i^1 \setminus nb$   
   else  
     if *better\_CH*(*i*)  $\vee$  *no\_CH*(*i*) then  
       // I have no CH, or I am a self-elected, not announced  
       // CH, or I am CH but but my neighbor with higher ID  
       // is CH also, so I give up and become a non-CH  
        $c.i = sender; n.i = nb; d.i = 1$   
       send *CL\_REJ*(*ID<sub>i</sub>*, *j*, 0) to all  $j \in N_i^1$

*E.03* rcv proper *CL\_ANN*(*sender*, *dest*, 1) from *nb*  $\longrightarrow$   
 if ( $sender == c.i$ ) then  
   if ( $n.i \notin N_i^1$ ) then  $n.i = nb$   
 else  
   if ( $c.i == ID_i \wedge n.i == ID_i$ )  $\vee$  *no\_CH*(*i*) then  
     // I have no CH, or I am a self-elected, not announced  
     // CH, and I found a CH within two hops, so I give up  
     // and become a non-CH  
      $c.i = sender; n.i = nb; d.i = 2$

*E.04* rcv proper *CL\_REJ*(*sender*, *dest*, 0) from *nb*  $\longrightarrow$   
 if ( $sender == c.i$ ) then  
    $c.i = n.i = ID_i$   
   send *CL\_REJ*(*sender*, *j*, 1) to all  $j \in N_i^1 \setminus nb$

*E.05* rcv proper *CL\_REJ*(*sender*, *dest*, 1) from *nb*  $\longrightarrow$   
 if ( $sender == c.i$ ) then  $c.i = n.i = ID_i$

*E.06* rcv proper *CL\_REQ*(*sender*, *dest*, 0) from *nb*  $\longrightarrow$   
 if ( $ID_i == dest$ ) then  
   if  $\neg is\_CH(i)$  then send *CL\_REJ*(*ID<sub>i</sub>*, *sender*, 0) to *nb*  
   else *Update*(*sender*, *ID<sub>i</sub>*, *ID<sub>i</sub>*, *nb*, 1)  
 else  
   if ( $dest == c.i \wedge dest \in N_i^1 \wedge n.i == dest$ ) then  
     send *CL\_REQ*(*sender*, *dest*, 1) to *n.i*  
   else send *CL\_REJ*(*ID<sub>i</sub>*, *sender*, 0) to *nb*

*E.07* rcv proper *CL\_REQ*(*sender*, *dest*, 1) from *nb*  $\longrightarrow$   
 if ( $ID_i \neq dest \vee \neg is\_CH(i)$ ) then  
   send *CL\_REJ*(*ID<sub>i</sub>*, *sender*, 0) to *nb*  
 else *Update*(*sender*, *ID<sub>i</sub>*, *ID<sub>i</sub>*, *nb*, 2)

---

announce yet ( $n.i == ID_i$ , and when it announces it,  $n.i = \perp$ ), then the current node gives up its clusterhead status (Action E.03).

If a *CL\_REJ* message is received by some node (Actions E.04 and E.05), and the message has originated from the node's clusterhead, then since it has no current clusterhead, the node elects itself as a clusterhead. If the message has traveled less than two hops, then it is relayed further.

If a *CL\_REQ* message is received by some node (Actions E.06 and E.07), then a certain node *dest* (that is seen by *sender* as its clusterhead) must be checked whether it is (still) a clusterhead or not. If the destination is a clusterhead, then the routing table is updated using Macro *Update*. As the link-cluster architecture requires at most two hops between a clusterhead and a non-clusterhead that belongs to the cluster, these messages can travel at most two hops.

If the message had traveled one hop, Action E.06 is responsible for checking whether the message had reached the destination (the sender's clusterhead) or not. If not, then the message has to travel one or more hop. In that case, this intermediate node must share the same clusterhead as the sender, it must have the clusterhead as its direct neighbor and have its *n.i* set to the clusterhead as well. If not, a *CL\_REJ* message is sent back.

If the message traveled two hops, Action E.07 checks whether the message reached the destination (the sender's clusterhead) or not. If not, then the clusterhead is more than two hops away so a *CL\_REJ* message is sent back.

The time between a node *i* elects itself as the clusterhead and the announcement done by a broadcast (at *time.period*) can be used by *i* to select some other node *j* as a clusterhead instead of *i* itself as long as *j* is within two hops from *i*. Since Action *E.01* is executed periodically at some intervals, if a node *i* decides to self-nominate as a clusterhead (by setting both *c.i* and *n.i* to itself), if before Action *E.01* is enabled and selected, some *CL\_ANN* messages reached the node coming from clusterheads within two hops (Action E.02 or E.03), then *i* may elect the node whose message reached *i* first as a clusterhead. In this way, the total number of clusterheads is reduced.

Predicate  $is\_CH(i) \equiv (c.i == ID_i \wedge d.i == 0 \wedge n.i == \perp)$  is evaluated to *true* when the node is an announced CH.

Predicate  $better\_CH(i) \equiv (c.i == ID_i \wedge (n.i == c.i \vee sender > ID_i))$  is evaluated to *true* when: the node *i* is an announced clusterhead and there is a clusterhead with higher ID in its one hop neighborhood, or the node has self-nominated without announcing yet.

Predicate  $some\_CH(i) \equiv (c.i \neq ID_i \wedge ((c.i \in N_i^1 \wedge is\_CH(c.i)) \vee (n.i \in N_i^1 \wedge c.(n.i) == c.i)))$  is evaluated to *true* when the node *i* is not a clusterhead, and there is clusterhead in one or two hop neighbor.

Predicate  $no\_CH(i) \equiv (c.i \neq ID_i \wedge (c.i \notin N_i^1 \vee n.i =$

$\perp \vee c.i \neq c.(n.i)))$  is evaluated to *true* when node *i* is not a clusterhead and what he sees as a clusterhead is not a direct neighbor, he has no proper *n.i* variable or if *n.i* is a neighbor of *i* it does not share the same clusterhead as *i*.

### 3.2. Routing

Module *Routing* (Algorithm 3.2) contains guards regarding routing. For intra-cluster routing, the shortest paths are maintained. For inter-cluster routing, we implement routing on-demand (the shortest paths are maintained only for the nodes that need to send packets).

---

#### Algorithm 3.2 Module *Routing* Self-Stabilizing Clusterhead Routing

---

**Actions:**

```

R.01 rcv ARP(sender, dest, ch, path) from neighbor nb →
    if ( $c.i == ID_i$ ) then // the node is CH
        if  $dest == ID_i$  then Update(sender, dest, ch, path) and
            send an ACK(sender, dest, ch, path, path) to nb
        else
            if  $ID_i \in path$  then discard /* a loop */
            else
                 $path \leftarrow path + ID_i$ 
                if ( $dest \in routing\ table$ ) then send
                    ARP(sender, dest, ch, path) to the neighbor
                    on the shortest path
                else send ARP(sender, dest, ch, path) to
                    all my neighbors except nb
            else // the node is either gateway or ordinary node
                if  $dest == ID_i$  then
                    send ROUTE(sender, dest, ch, path) to n.i
                else
                    if  $ID_i \in path$  then discard it // a loop
                    else
                         $path \leftarrow path + ID_i$ 
                        send ARP(sender, dest, ch, path) to all  $j \in N_i^1 \setminus nb$ 

R.02 rcv ROUTE(sender, dest, ch, path) from neighbor nb →
    if ( $c.i == ID_i$ ) then // the node is CH
        if  $dest == ID_i$  then Update(sender, dest, ch, path) and
            send an ACK(sender, dest, ch, path, path) to nb
        else // the node is either gateway or ordinary node
            if  $ID_i \in path$  then discard it // a loop
            else
                 $path \leftarrow path + ID_i$ 
                send ROUTE(sender, dest, ch, path) to n.i

R.03 rcv ACK(sender, dest, ch, path, route) from neighbor nb →
    if  $ID_i$  is the last ID in path then
        if ( $c.i == ID_i$ ) then Update(sender, dest, ch, route)
            delete  $ID_i$  from path
        if ( $path \neq empty\ string$ ) then
             $nb = last\ ID\ in\ path$ 
            if ( $nb \in N_i^1$ ) then
                send ACK(sender, dest, ch, path, route) to nb

```

---

Messages *ARP* and *ROUTE* contain the following fields: *sender* (sender ID), *dest* (destination ID), *ch* (the CH of the sender), *path* (the path the message went on).

Message *ACK* contains the following fields: *sender* (sender ID), *dest* (destination ID), *ch* (the CH of the sender), *path* (the path the message has to travel to reach the destination), *route* (the complete path between the sender and destination). Two fields are used to represent path: *path* contains only the partial path, from the current node to the destination, while *route* contains the complete path. When a message passes through an intermediate node, the node is deleted from the partial path such that the next intermediate node becomes available for routing. The complete path is processed by the CH of the destination cluster if it is a good path (either the only one or the shortest one).

A node sends an *ARP* message when it needs to find out the shortest path toward a destination. If the destination is not a CH but a gateway or a regular node, then the information received is sent to the CH of the destination node to be incorporated in its routing table (as a *ROUTE* message). If the destination is a CH, or the CH has received a *ROUTE* message from one of its cluster 'clients', the *path* is stored for future use. If the path contained by the message is the only path or the shortest one, an *ACK* message is sent back to the sender by the CH of the destination node.

The macro *Update(sender, dest, ch, path, length)* uses the reverse of the *path* to see whether you can store it as a path to *sender* (the sender) or *ch* (the CH of the sender), if it is a good path – either the only one or currently the shortest one.

Whenever Macro *Update(sender, dest, ch, path)* is called, some information will be checked to see whether it can be added or it can improve the current information in the routing table. The reverse of the *path* represents a complete path back to the *sender* and/or the clusterhead of the sender *ch*. If it is a good path — either the only one or the shortest one — it is added. If not, it is ignored.

## 4. Conclusion

We have presented a self-stabilizing a self-stabilizing cluster routing algorithm for MANET based on link-cluster architecture. The algorithm selects the clusterheads, and then builds in those nodes routing tables regarding nodes inside and outside the cluster.

The proposed protocol guarantees that starting in an arbitrary configuration and in finite number of steps, the network is divided into clusters using link-cluster architecture. Each cluster has a single special node, called *clusterhead* that holds the routing information regarding inter and intra-cluster communication. The protocol ensures that any node belongs to a single cluster and is within two hops from the clusterhead. For intra-cluster routing, the shortest paths are maintained. For inter-cluster routing, we implement routing on-demand (the shortest paths are maintained only for the nodes that need to send packets). The algorithm is self-

stabilizing: it copes with wrong initialization, and it adapts to arbitrary movement of nodes, and joining and/or leaving of existent nodes.

## References

- [1] A. Arora and M. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.
- [2] C. Chiang, H. Wu, W. Liu, and M. Gerla. Routing in clustered multihop, mobile wireless networks with fading channel. *ACM Baltzer Journal of Wireless Networks*, 1:255–265, 1995.
- [3] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
- [4] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Selected Writings of Computing: A Personal Perspective*, pages 41–46, 1982.
- [5] G. Dimitriadis and F. Pavlidou. Two-hop polling: An access scheme for clustered, multihop ad hoc networks. *International Journal of Wireless Information Networks*, 10(3):149–158, 2003.
- [6] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
- [7] M. Gerla and J.-C. Tsai. Multicluster, mobile, multimedia radio network. *ACM-Baltzer Journal of Wireless Networks*, 1:225–265, 1995.
- [8] M. Gouda. *Elements of network protocol design*. John Wiley & Sons, Inc., 1998.
- [9] T. Johansson and L. Carr-Motyckova. On clustering in ad hoc networks. *First Swedish National Computer Networking Workshop (SNCNW 2003)*, 2003.
- [10] B. Lee, C. Yu, and S. Moh. Issues in scalable clustered network architecture for mobile ad hoc networks. *Handbook of Mobile Computing*, 2004.
- [11] P. Mansinthorn and A. B. Bogobowicz. Network decomposition in wireless communication. *3rd Annual Communication Networks and Services Research Conference (CNSR 2005)*, pages 243–248, 2005.
- [12] M. Steenstrup. Cluster-based networks. *Chapter 4 of Ad Hoc Networking*, 2001.