Journal of King Saud University – Computer and Information Sciences (2017) 29, 378–398

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

A logical structure based fault tolerant approach to handle leader election in mobile ad hoc networks

Bharti Sharma [a,*], **Ravinder Singh Bhatia** [b], **Awadhesh Kumar Singh** [b]

[a] *DIMT, Kurukshetra, India*
[b] *NIT, Kurukshetra, India*

Received 11 January 2015; revised 4 March 2015; accepted 15 March 2015
Available online 31 October 2015

KEYWORDS

Mobile ad hoc network;
Leader election;
Clustering;
Ring formation

Abstract We propose a light weight layered architecture to support the computation of leader in mobile ad hoc networks. In distributed applications, the leader has to perform a number of synchronization activities among participating nodes and numerous applications; hence, it is a stressed node and consequently prone to failure. Thus, fast and fault tolerant leader election is a major concern and popular area of research in distributed computing networks, in general, and wireless ad hoc networks, in particular. In the present article, we have proposed a fault tolerant leader election approach. More importantly, the nodes elect the leader quickly on the basis of local information only. The illustration includes suitable examples. The correctness proof and performance evaluation has also been presented.
© 2015 The Authors. Production and hosting by Elsevier B.V. on behalf of King Saud University. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

1.1. Background

A mobile ad hoc network (MANET) is easily installable and thus, a very economical and popular computing environment; however, it is highly constrained and challenging too. A portable node possesses limited computing power processors, small stable storage, thin battery for energy backup, and a short communication range. They can communicate only via message passing over wireless links. Nodes that are not in the transmission range of each other can communicate via message relay. The nondeterministic mobility pattern incurs concurrent and arbitrary topological changes. The topological changes become more frequent because of the dynamism in wireless links and limited availability of bandwidth. A highly performance retarding offshoot of this phenomena is the variable message delay. Hence, the distributed algorithms developed for static domain are not directly implementable in MANETs.

Nowadays, there are many popular distributed applications that are executed in MANETs, e.g. agreement problem, internode communication, data exchange, service request, data aggregation, key distribution, group communication, and privilege grant. In the execution of any such applications, especially in MANET-like fault prone environment, the leader is a critical component that is responsible to coordinate such activities.

* Corresponding author.
E-mail addresses: bharti_kanhiya@yahoo.co.in (B. Sharma), rsibhatia@yahoo.co.in (R.S. Bhatia), aksinreck@rediffmail.com (A.K. Singh).
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2015.03.001
1319-1578 © 2015 The Authors. Production and hosting by Elsevier B.V. on behalf of King Saud University.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Thus, it is very useful to have a leader to make critical decisions in an easy way. Therefore, finding a leader (a.k.a. coordinator) is a popular research challenge in distributed systems community.

1.2. The problem statement

The leader election is a very simple form of symmetry breaking and it is a classic synchronization problem of distributed computing. An algorithm for choosing a unique process to deliver a particular service is called an *election algorithm* and the elected process is called *leader*. For example, in a central-server architecture based mutual exclusion algorithm, the 'server' is chosen from among the processes. It is necessary that all the processes agree on the choice. Afterward, if the process that delivers the service of the server fails or wishes to stop providing the service then another election process instance is executed to choose a substitute leader. We say that a process *initiates the election* if it takes an action that begins the execution of a particular instance of the election algorithm. An individual process does not initiate more than one election process instance at a time; however, in principle, several processes could initiate concurrent election process instances. At any point in time, a process p_i is either a *participant* – meaning that it is involved in some execution of the election algorithm – or a *non-participant* – meaning that it is not currently involved in any election. An important requirement for the choice of the elected process is to be unique, even if multiple processes call elections concurrently. For example, more than one process could notice independently that a leader process has failed, and they initiate elections. In short, computing a leader is to eventually elect a distinguished node from a given set of nodes.

1.3. Related work and motivation

A detailed discussion on leader election problem and protocols can be found in many popular books (Ghosh, 2010; Attiya and Welch, 2004; Garg, 2004; Raynal, 2013). In order to solve the problem, several distributed algorithms have been proposed in the literature that include a few classic protocols (Chang and Roberts, 1979; Hirschberg and Sinclair, 1980; Garcia-Molina, 1982) and an interesting array of recently published protocols (Boukerche and Abrougui, 2007; Derhab and Badache, 2008; Haddar et al., 2008; Dagdeviren and Erciyes, 2008; Ingram et al., 2009; Raychoudhury et al., 2014; Raz et al., 2004; Sharma and Singh, 2011; Shirmohammadi et al., 2009; Singh and Sharma, 2011a,b; Vasudevan et al., 2003, 2004; Malpani et al., 2000; Jain and Sharma, 2012; Subathra et al., 2012). In general, the protocols developed for the conventional distributed systems fail to work efficiently when implemented in cellular and ad hoc network based modern computing systems. For instance, the leader election protocol for MANETs needs to be message efficient because the nodes possess limited energy. The message overhead can be controlled by imposing a cost-effective logical structure on the physical network. However, most of the above listed protocols don't exploit this idea. Furthermore, the energy of the leader node dissipates faster than ordinary nodes, due to the coordination overheads that the leader handles. It makes the leader node susceptible to crash. Secondly, the absence of a leader leads to reduced utilization of nodes' resources, application discontinuity, performance degradation, or aborts. We intend to prevent these problems and thus our motivation is not to develop yet another leader election protocol, rather to design an election protocol that can choose the leader in a faster way. In this paper, we propose a logical structure based light weight protocol to support the leader election in MANETs. Also, our protocol quickly finds a replacement in the event of leader failure.

2. The protocol concept

2.1. The system settings

We make the following assumptions about the system architecture. In our illustration, the network is assumed to have a finite number of nodes distributed in a geographic region. A node has a universal identifier UID (called ID, for short) represented by a binary sequence that is unique and constant throughout the network lifetime. A node's ID may be, for example, its MAC/IP address or CPU ID (Zeng et al., 2010). The unique ID assignment is a non-trivial problem and beyond the scope of this paper. Also, the nodes are assigned certain weights depending on the quality related attributes, like node's residual power, computational capability, speed etc. The node IDs are used as a tie-breaker among equal weight nodes. The nodes may crash and recover autonomously. The communication links between nodes are assumed to be bidirectional FIFO and guarantee a message delivery if the corresponding sender and receiver remain connected during the message propagation. The message propagation delay is nondeterministic however bounded. Each node has a sufficiently large receive buffer; hence, it never suffers buffer overflow.

We propose a layered architecture, as shown in the following Fig. 1, to support leader election in MANETs.

- At the lowest layer, a clustering algorithm divides the MANET into balanced clusters, using a modified version of the previously designed merging clustering algorithm (MCA) (Dagdeviren et al., 2005) and it constructs a heap of cluster nodes.
- The second layer implements the ring formation (RF) algorithm which constructs the virtual ring of cluster heads. Further, the ring members are represented in the form of a heap.
- Finally, our leader election (LE) algorithm finds the leader out of the cluster heads.

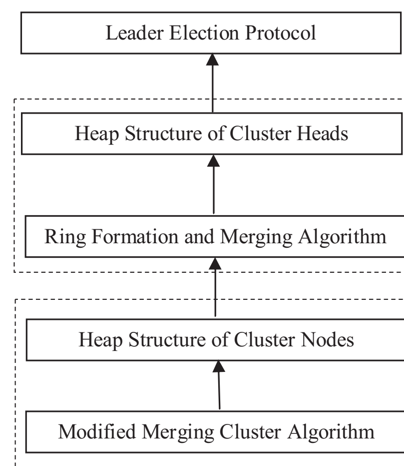

Figure 1 The layered architecture.

2.2. Clustering using modified merging cluster algorithm

Consider a MANET represented by an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of connecting edges. The merging clustering algorithm (MCA) (Dagdeviren et al., 2005) finds clusters in a MANET, by merging the clusters to form higher level clusters, using the technique mentioned in GHS (Gallager et al., 1983), the seminal minimum spanning tree (MST) algorithm. Such construction is a highly non local activity that needs coordination, and thus message exchanges, among far away nodes in the system (Basagni et al., 2004; Han and Jia, 2007). However, our algorithm performs the clustering operation without using MST like construction. Hence, our approach requires less coordination and has low message overhead. Secondly, we maintain an upper as well as a lower bound on the cluster size during the clustering operation, which leads to the balanced cluster formation. Third, we have modified the message structure of MCA (Dagdeviren et al., 2005) in such a way that the exchange of information about all the nodes in a cluster is in the form of a max heap. This results in selecting a new cluster head in constant time in the event of cluster head failure. Therefore, we named our algorithm as modified merging cluster (MMC) algorithm.

2.3. Ring formation and merging algorithm

The objective is to construct a single ring of all the cluster heads. Each cluster head arranges itself in a ring. Thus, during the ring formation process, multiple rings may be formed simultaneously; however, at any time, each node can participate only in a single ring formation process. Hence, in case, multiple rings are formed, the rings contain a mutually exclusive set of nodes. Subsequently, these rings are merged together to form a composite ring. Our ring formation method is inspired from Yen and Chi (2004), Dagdeviren and Erciyes (2006). When a node belonging to any ring sends a merge request to another node belonging to a different ring, it considers the predecessor, successor and successor's successor of that node. Therefore, any two rings can be merged by modifying the values of the above three variables. Once a unique single ring of all the cluster heads is formed, we construct a max heap of all the members of the ring. This ring is used in our leader election protocol for electing a new leader and the heap of ring members helps in electing a new leader in constant time in the event of leader crash.

2.4. The leader election algorithm

We propose an asynchronous leader election algorithm for unidirectional ring networks. Any node can initiate the election by sending its node ID to its neighbor node. The neighbor node forwards the request further, by appending its node ID to the request, using max heap. Finally, the initiator receives the request back in the form of a max heap with root as leader. Subsequently, the initiator notifies the information about the created max heap and the elected leader to all the nodes in the ring. Now, each node in the ring possesses a max heap structure representing the ring. Therefore, in the case of leader failure, the node in its neighborhood detects the failure, removes the entry of the failed leader and modifies its local max heap accordingly. The node at the root of the modified heap is the new leader and this information is notified all through the ring.

3. The algorithm

In order to implement the heap structure in our algorithm we need some basic functions operating over the heap. The detailed discussion and pseudo code of such functions can be found in Cormen et al. (2001). Some of them have been reproduced below.

Our algorithm uses the following functions, detailed in Fig. 2 below, to perform various operations over heap.

3.1. The clustering algorithm

3.1.1. Data structures

The following data structures are maintained at each node.

(i) *Cluster_leader_id* (integer): It is ID of the node which is leader of the cluster. Each node, including the leader, maintains this variable.
(ii) *Node_id:* Every node in the network is provided a unique integer ID.
(iii) *Heap_array* (array): It is an array of the whole cluster in the form of max heap. Each entry contains *Node_id* and some performance metric on the basis of which max heap is constructed.
(iv) *Heap_size* (integer): It represents the number of elements in the *Heap_array*.
(v) *Cluster_size* (integer): It represents the number of nodes in the cluster. It is used when the number of nodes exceeds the maximum permitted value for the size of a cluster.
(vi) *pm:* It represents the performance metric (a.k.a. node weight) that may be computed based on parameters like computation power, mobility speed, residual battery power, fault tolerance etc.
(vii) *Target_size* (integer): It is the maximum permitted size of a cluster. Each node maintains this variable and while merging two clusters, the size of the composite cluster should not exceed *Target_size*.
(viii) *T_out:* It is a timer.

3.1.2. Messages

(i) *Req_msg* ⟨*Node_id*, *Cluster_size*⟩: A cluster leader node initiates the clustering operation by sending *Req_msg* message to a destination node.
(ii) *Ldr_req_msg* ⟨*Node_id*, *Cluster_size*⟩: On receiving *Req_msg* message from a node outside its cluster, a cluster member node sends *Ldr_Req_msg* message to its cluster leader.
(iii) *Become_mbr* ⟨*Node_id*, *Heap_array*, *Heap_size*⟩: A cluster leader node sends *Become_mbr* message on receiving *Ldr_req_msg* from another leader that has a smaller node ID.
(iv) *Become_ldr* ⟨*Node_id*, *Heap_array*, *Heap_size*⟩: A cluster leader node will send *Become_ldr* message on receiving *Ldr_req_msg* from another leader that has a greater node ID.

```
(1) Parent (i)
return ⌊i/2⌋;
(2) Left (i)
return 2i;
(3) Right (i)
return 2i+1;

(4) Heap_Increase_Id (H, i, id, pm)                /* Increase key of a node
H[i] = <id , pm>
While i > 1 && H[Parent(i)] < H[i]
    Exchange H[i] with H[Parent(i)]
i = Parent(i)

(5) Max_heap_insert (H, id, pm)                    /* Insertion of a new node
H.heapsize = H.heapsize + 1;
H[H.heapsize] = -∞
Heap_Increase_Id (H, H.heapsize, id, pm)

(6) Heap_remove_CH (H)                             /* Remove cluster head
H[1] = H[H.heapsize]
H.heapsize = H.heapsize - 1
Max_Heapify (H, 1)

(7) Max_Heapify (H, i)
l = Left(i)
r = Right(i)
if l <= H.heapsize && H[l] > H[i]
    largest = l
else largest = i
if r <= H.heapsize && H[r] > H[largest]
    largest = r
if largest ≠ i
    exchange H[i] with H[largest]
    Max_Heapify (H, largest)

(8) Heap_remove_node (H, i)                        /* Remove any node other than CH
H[i] = H[H.heapsize]
H.heapsize = H.heapsize - 1
Max_Heapify (H, i)

(9) Merge_Heap (H, H1)
If H.heapsize < H1.heapsize
    For every node i in H
    Max_heap_insert (H1, H[i].id, H[i].pm)
Else
    For every node i in H1
    Max_heap_insert (H, H1[i].id, H1[i].pm)
```

Figure 2 Basic heap operations.

(v) *Ldr_ACK* ⟨*Node_id*, *Cluster_size*, *Heap_array*, *Heap_size*⟩: A node sends *Ldr_ACK* message on receiving *Become_mbr* message and the successful completion of all operations.

(vi) *Mbr_ACK* ⟨*Heap_array*, *Heap_size*⟩: A node sends *Mbr_ACK* message on receiving *Become_ldr* message and the successful completion of all operations. The receiver of *Mbr_ACK* message becomes a member of the cluster.

(vii) *Reorganise_cluster* ⟨*Node_id*, *Cluster_size*, *Heap_array*, *Heap_size*⟩: A node will multicast a *Reorganise_cluster* message on receiving *Ldr_ACK* message. The leader of a cluster computes a new level and multicasts the *Reorganise_cluster* message to its cluster members to update their cluster level information along with the complete heap information.

(viii) *Reorganise_cluster_ACK:* A node sends *Reorganise_cluster_ACK* message in response to *Reorganise_cluster* message.

(ix) *Reject_msg:* A node sends this message to the sender of *Req_msg* if the merging operation is not possible because the sum of sizes of both clusters is greater than *Target_size*.

3.1.3. The modified merging cluster algorithm

Initially, each node is assumed to be the head of its own cluster that contains a single member. Each node in a cluster maintains heap array of the max heap structure of its cluster nodes. The sender node appends the heap array of its own cluster with each *Become_mbr*, *Become_ldr*, *Ldr_ACK* and *Mbr_ACK* message. After sending *Become_ldr*, *Become_mbr*,

Ldr_ACK or *Mbr_ACK*, the message *Reorganize_cluster* that contains the heap array of the latest merged cluster, is sent to every member node in the earlier cluster. If a cluster-head receives a heap array appended with any of the above messages, except *Reorganize_cluster*, it merges its own heap array with the received one. Similarly, if an internal node receives a *Reorganize_cluster* message, it merges its own heap array with the received one and modifies its heap size accordingly.

3.1.4. Procedures

The various procedures used in the clustering algorithm are detailed below in Fig. 3.

3.1.5. The pseudo code

The following Fig. 4 contains the pseudo code of the clustering algorithm.

3.2. Ring formation and merging

We have the following assumptions:

(a) The channel is FIFO.
(b) No node in the ring crashes during merge operation.
(c) The network is reliable.

3.2.1. Data structures

(i) *rc:* It is an integer representing the ring count, i.e. the number of nodes present in the ring. It is equal to 1 for a cluster head that is not part of any ring. In our illustration, we refer the local ring count as *l_rc* and foreign ring count as *f_rc*.
(ii) *CH_heap:* It represents cluster head heap array, i.e. max heap of the performance metric and node ID of all cluster heads present in the ring. It is an array where each element contains a doublet of *pm* and *Node_id*.
(iii) *pred:* It is integer pointing to the predecessor of the current node in the ring.
(iv) *succ:* It is integer pointing to the successor of the current node in the ring.
(v) *s_succ:* It is integer pointing to the successor's successor of the current node in the ring.
(vi) *Node_id:* Every node in the network is provided a unique integer ID.
(vii) *pm:* It represents the performance metric (a.k.a. node weight) that may be computed based on parameters like computation power, mobility speed, residual battery power and fault tolerance.
(viii) *nmop:* It is a Boolean flag. It is true if no merging operation is in progress, else it is false.

3.2.2. Messages

(i) *Setf* ⟨*i.pred*, *i.succ*, *i.s_succ*, *rc*, *CH_heap*⟩: A node sends this message to another node belonging to a different ring during the ring merging operation. First three items represent the intended value of predecessor, successor and successor's successor, respectively, of the destination node on merging two rings. Also, two more items, contained in the message, are *CH_heap* and *rc* of the source node. If a non-cluster-head receives this message, it forwards the same to its cluster head. The symbol '_' in any field, namely *i.pred*, *i.succ*, or *i.s_succ* represents that the intended value of predecessor, successor or successor's successor of the destination node will remain unchanged.
(ii) *Setl* ⟨*i.pred*, *i.succ*, *i.s_succ*⟩: It is a message that is sent by a node to its predecessor or successor in the same ring. The items contained represent the same thing as in *Setf* message.
(iii) *Blk* ⟨*pm*, *Node_id*, *f*⟩: This message is used to block the destination node from any merging operation. The flag *f* signifies that the receiving node needs to forward it to its predecessor, in case, it is 1; no, otherwise.

Send_Req():
```
{
    If T_out = 0
        Then send Req_msg to all neighbors
}
```

Send_Ldr_Req():
```
{
    If received Req_msg and (Node_id != Cluster_leader_id)
        Then send Ldr_Req_msg to Cluster_leader_id
}
```

Receive_Req_Msg(): The leader node on receiving *Req_msg* message or *Ldr_req_msg* message performs this procedure.
```
{
    If received message is Req_msg or Ldr_req_msg
    Then if Cluster_ldr_id of receiver > Cluster_ldr_id of sender
        Then send Become_mbr
    Else
        send Become_ldr
}
```

Figure 3 Basic procedures.

```
(i) On receiving Become_mbr message by a node: /* reorganize cluster */
{
    Extract foreign Heap_array;
    Change its Cluster_leader_id;
    Send Ldr_ACK to the sender of Become_mbr;
    Send Reorganise_cluster to nodes in local Heap_array;   /* Call Max_Heap_Insert */
    Merge local Heap_array with foreign Heap_array;          /* Call Merge_Heap */
}
(ii) On receiving Become_ldr message by a node
{
    Extract foreign_heap_array;
    Send Mbr_ACK to sender of Become_ldr;
        Send Reorganise_cluster to nodes in local Heap_array;
        Merge local Heap_array with foreign Heap_array; /* Call Merge_Heap */
}
(iii) On receiving Mbr_ACK or Ldr_ACK message by a node
{
    Extract foreign Heap_array;
    Change Cluster_leader_id if received Mbr_ACK;
    Send Reorganise_cluster to nodes in local Heap_array;
    Merge local Heap_array with foreign Heap_array;          /* Call Merge_Heap */
}
(iv) On receiving Reorganise_cluster message by a node
{
    Extract foreign Heap_array;
    Merge foreign Heap_array with local Heap_array;          /* Call Merge_Heap */
    If (new Cluster_leader_id ≠ Cluster_leader_id) then send Reorganise_cluster_ACK to Cluster_leader_id;
    Else send Reorganize_cluster_ACK to new Cluster_leader_id; set Cluster_leader_id = new
Cluster_leader_id;
}
```

Figure 4 Pseudo code.

(iv) *Blk_ack:* On receiving *Blk* message, a node checks whether it is also involved in any merged operation. If not, it sends a *Blk_ack* message. Otherwise, it checks whether the node weight received in the message is greater than it own. If so, it sends *Blk_ack* message; else, do nothing.

(v) *Rej:* A node sends *Rej* message to the initiator of merge operation, if the node could not receive all *Blk_ack* messages before time out. If the node itself was the initiator, there is no need to send any message.

(vi) *Idn:* This token holds the identity of the foreign node with which the merge operation is taking place. It is '?' if no merging operation is in progress. It is used only when a node initiates merge operation.

(vii) *Time_out:* If *Time_out* expires and no message is received, the node starts the merge operation, provided, a foreign node is present in the neighborhood.

(viii) *Blk_tm:* It is the time duration a node waits for *Blk_ack* messages after sending *Blk* messages.

3.2.3. The working of ring formation and merging algorithm

Our algorithm is inspired from a recently published algorithm for maintaining a ring structure in MANETs (Yen and Chi, 2004). When a cluster-head sends a request to a destination node to merge rings it appends its heap-array of cluster-heads with the request. If an internal node receives a request to merge rings it forwards the same to its cluster-head. The requester cluster-head sends a *Blk* message to its predecessor and successor to block them from engaging in any other merging operation. The cluster-head that received the request for merging rings sends a *Blk* message to its predecessor which in turn forwards it to its predecessor leading to blocking of two predecessor nodes from engaging into any other ring merging operation. The first reply sent by the cluster-head (if not a reject), which received the request to merge rings, to the cluster-head that made the request, contains the heap-array of the ring of cluster-heads present in its own ring. The cluster-head A, with ring count equal to 1 signifies that there's no other node in its cluster and it has its $pred$ = A, $succ$ = A and s_succ = A. Likewise, a cluster with ring count equal to 2 has only two nodes 'A' and 'B' having the entries as follows:

$$Pred_{\mathrm{A}} = \mathrm{B}, \quad succ_{\mathrm{A}} = \mathrm{B}, \quad s_succ_{\mathrm{A}} = \mathrm{A}$$

$$Pred_{\mathrm{B}} = \mathrm{A}, \quad succ_{\mathrm{B}} = \mathrm{A}, \quad s_succ_{\mathrm{B}} = \mathrm{B}$$

It is worth noting that the requests received at a cluster-head to change either of the data structures *pred*, *succ* and *s.succ* from its ring member and non-ring member are not handled in the same way. The detailed pseudo code is given in Appendix I.

In order to enable the structural modifications to progress concurrently, the ring structure is modified in a localized and mutually exclusive manner. The modified merging clustering (MMC) and ring formation (RF) modules isolate the upper layers from the communication link failures. Though, the architecture is robust in the case of updating the failed links, the node crashes are not covered. Nevertheless, due to time out mechanism, if a node crashes before the exchange of all messages related to current instance of ring formation process, the participant neighbors of the crashed node abort the 'in progress' instance and initiate a new one involving the remaining neighbors that are ready to participate in the ring formation process further. The crashes in the ordinary nodes are easy

to handle. However, the crashes of dominator nodes may amend the network topology greatly. In the course of topology changes, a participant node may leave the computation due to forced termination or power off. This might result in an incorrect ring structure. To deal with, an additional mechanism needs to be introduced. By the addition of a fault tolerance module, orthogonal to this architecture, it is possible to recover from the node crashes. This practical issue is beyond scope of the paper and intended for future study.

4. The complexity analysis

The proposed protocol consists of the following three layers.

1. Modified merging cluster (MMC) algorithm
2. Ring formation (RF) algorithm
3. Leader election (LE) algorithm

Theorem 1. The message complexity of the protocol is $O(n)$ where n is the number of nodes.

Proof. The message complexity for the protocol is the sum of the message complexities of the above three algorithms and the complexity of termination detection of the first two algorithms.

Assume, there are n nodes in the network. For each merge operation of two clusters, three messages (one *Req_msg*, one *Become_ldr* or *Become_mbr*, and one *Ldr_ACK* or *Mbr_ACK*), K number of *Reorganise_cluster* messages, and K number of *Reorganise_cluster_ACK* messages are required, where K is the maximum permissible size of a cluster. Thus, total number of required messages would be $(3 + 2K)$.

The average number of clusters formed at the end of MMC execution will be n/K, where n is the total number of nodes in the network. Now, let us choose K such that $K^2 = n$ and hence the number of clusters would also be K. The merging of two clusters needs a single iteration of the MMC algorithm. Similarly, the merging of three clusters needs two iterations of the MMC algorithm. Thus, merging of K clusters would require $(K - 1)$ iterations of the MMC algorithm. Therefore, the total number of messages sent would be the product of the number of messages sent in single iteration and the number of iterations, i.e. $(3 + 2K) * (K - 1) \approx 2n$. Hence, the message complexity would be $O(n)$.

According to above assumption the number of cluster heads (or clusters) will be K. Also, the number of iterations of ring merge operation will be $(K - 1)$. The number of messages sent in single iteration of ring merge operation will be 10 (two *Setf* messages + four *Setl* messages + 6 *Blk* messages). In addition, $2K$ heap multicast messages will also be needed. Hence, in a single iteration, total $(10 + 2K)$ messages are exchanged. Therefore, the total number of messages exchanged in ring formation algorithm will be $(10 + 2K) * (K - 1)$. Thus, the message complexity will be $O(K^2)$, i.e. $O(n)$.

If the leader crashes, the absence of the leader is detected by a node i in its neighborhood if node i doesn't receive the next heartbeat message in the stipulated time. Subsequently, in order to find a new leader, node i reorganizes its heap structure, which does not involve any message exchange because it is a local computation. Subsequently, to propagate the reorganized heap among K cluster heads, total K messages are required. Thus, the message complexity would be $O(K)$.

Now, assuming termination detection requires negligible number of messages, the composite message complexity of the leader election protocol (LEP) can be computed as follows.

$$\begin{aligned} O(\text{LEP}) &= O(\text{MMC}) + O(\text{RF}) + O(\text{LE}) \\ &= O(n) + O(n) + O(K) = O(n) \end{aligned}$$

□

5. The correctness proof

5.1. Modified merging cluster algorithm

Theorem. There can be only one cluster head in any cluster at any time.

Proof. Assume the contrary. Let P and Q be two distinct nodes belonging to the same cluster, say C_i and they have different *Node_id* as their cluster head. For this to happen, the nodes P and Q must have different max heap structures with one of them having a smaller node ID than the other. As each node ID is assumed unique, both the nodes must have different roots in their respective max heap structure. Hence, one of the two heaps must have a larger ID set as child than its parent node, violating the max heap property. Therefore, it is infeasible to have two different cluster heads of the same cluster at any point of time. □

5.2. Ring formation algorithm

Theorem. No two consecutive nodes in the ring can make topological changes concurrently.

Proof. Let, P and Q are two distinct nodes in a ring requesting topological change concurrently. Our protocol ensures that the one with a higher priority can proceed.

By our protocol, P will send a block message *Blk* to Q with its $Node_id_P$ and performance metric pm_P. Similarly, Q will also send a block message *Blk* to P with its $Node_id_Q$ and performance metric pm_Q. Now, there are two possibilities:

1. $pm_p > pm_Q$ or ($pm_p = pm_Q$ and $Node_id_p > Node_id_Q$): In this case, node Q will send an acknowledgement message *Blk_ack* to node P after receiving *Blk* message from node P. Thus, only node P will be able to proceed.
2. ($pm_Q > pm_P$) or ($pm_p = pm_Q$ and $Node_id_Q > Node_id_P$): In this case, node P will send *Blk_ack* message to node Q after receiving *Blk* message from node Q. Thus, only node Q will be able to proceed.

□

Theorem. Any iteration of ring formation algorithm terminates within a finite time.

Proof. In our system model, the message propagation delay has been assumed as finite. Also, the number of sequential

message propagations, involved in any iteration of ring formation, is finite, as shown above in the complexity analysis. Hence, the time taken for the completion of any iteration of the ring formation algorithm would also be finite. □

Theorem. There is unique leader in a ring.

Proof. The proof is similar to the proof of the MMC algorithm. □

6. Illustrating examples

6.1. Ring formation between two isolated nodes

If the value of *Target_size* is 1, there can be only one node in a cluster. Similarly, if the value of *Target_size* is 2, there can be two nodes in a cluster. Assume, there are two isolated nodes A and B and node A detects node B in its neighborhood (Fig. 5.a). Node A sends *Setf* message to node B (Fig. 5.b). Subsequently, node B replies with a *Setf* message to node A. Node B sets its predecessor *pred* = A, successor *succ* = A, and ring count *rc* = 2 (Fig. 5.c). Finally, both nodes A and B merge into a single ring and node A sets its predecessor *pred* = B, successor *succ* = B, and ring count *rc* = 2 (Fig. 5.d).

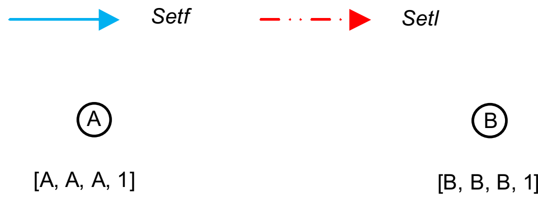

Figure 5.a Isolated nodes.

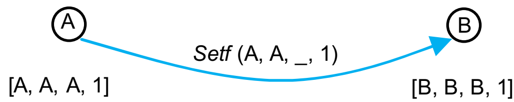

Figure 5.b Ring formation begins.

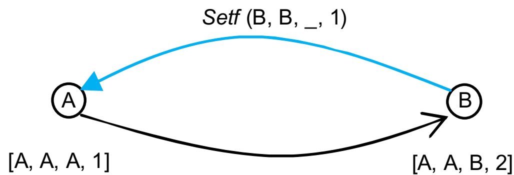

Figure 5.c Ring formation ends.

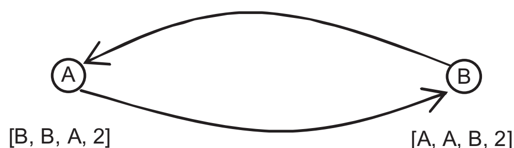

Figure 5.d Ring of two nodes.

6.2. Ring formation when one ring has rc = 1 and the other ring has rc = 2

Assume, there is a ring containing two nodes A and B and an isolated node C in the neighborhood (Fig. 6.a). Initially, node

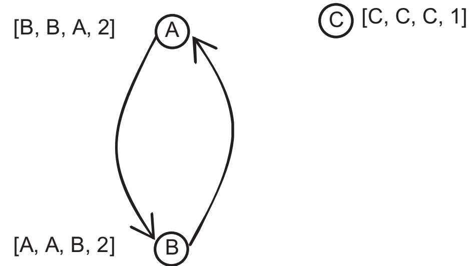

Figure 6.a Ring and isolated node.

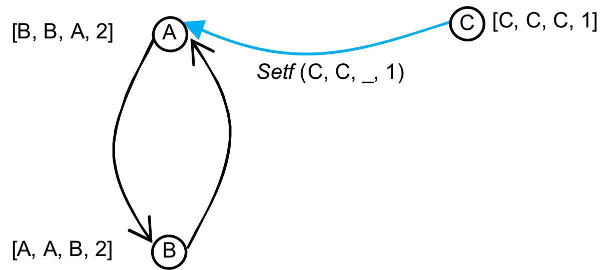

Figure 6.b New ring formation begins.

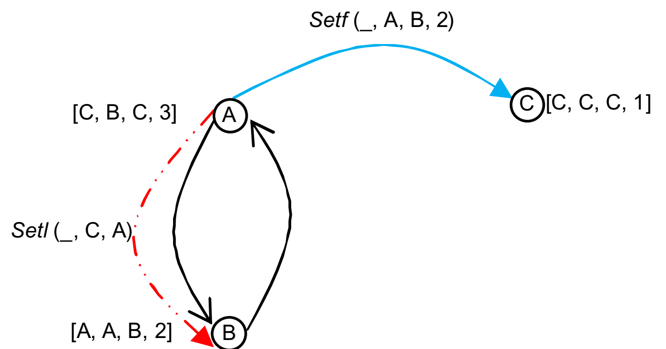

Figure 6.c New ring formation continued.

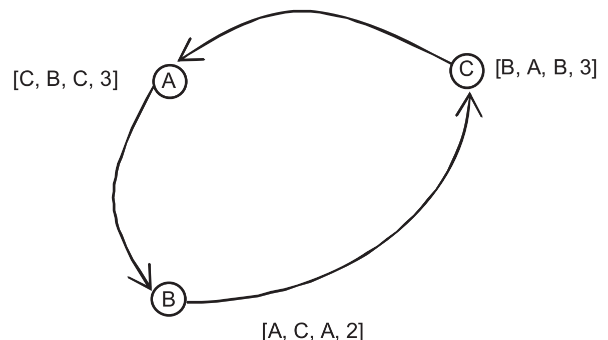

Figure 6.d New ring formation ends.

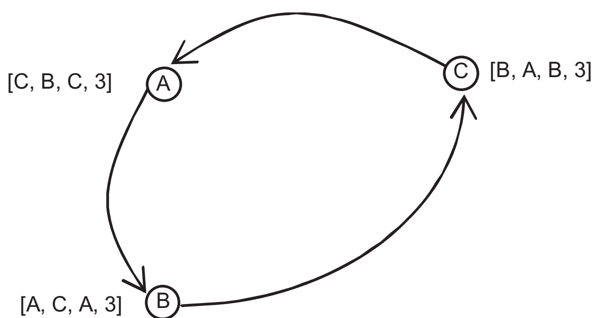

Figure 6.e Ring of three nodes.

C sends *Setf* message to node A (Fig. 6.b). Subsequently, node A replies with a *Setf* message to node C and *Setl* message to node B (Fig. 6.c). Node A sets its predecessor *pred* = C, successor *succ* = B, and ring count *rc* = 3. Now, node A multicasts the foreign heap array, which it received from node C previously, along with the new ring count. Subsequently, node C updates its max heap (Fig. 6.d). Finally, nodes A, B and C merge into a single ring and node B sets its predecessor *pred* = A, successor *succ* = C, and ring count *rc* = 3 (Fig. 6.e).

6.3. Ring formation when one ring has rc = 1 and the other ring has rc = 2

This case (see Fig. 7.a) is similar to the situation explained in the above Section 6.2. However, initially, node A sends *Setf* message to node C (Fig. 7.b). Subsequently, node C replies with a *Setf* message to node A (Fig. 7.c). Node A sends *Setl* message to node B (Fig. 7.d). Node A sets its predecessor

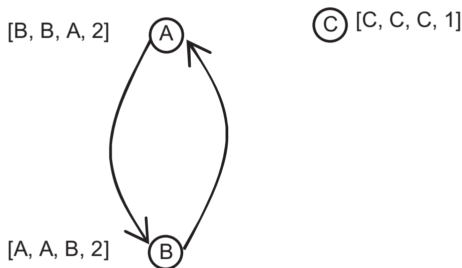

Figure 7.a Ring and isolated node.

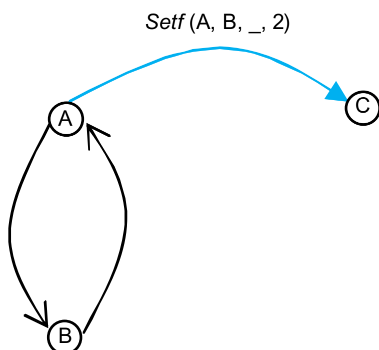

Figure 7.b New ring formation begins.

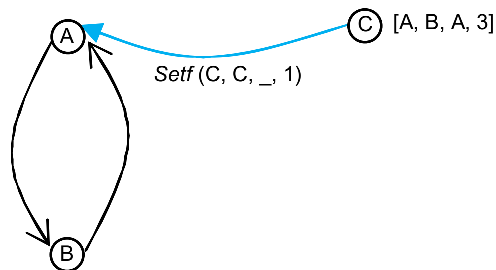

Figure 7.c New ring formation continued.

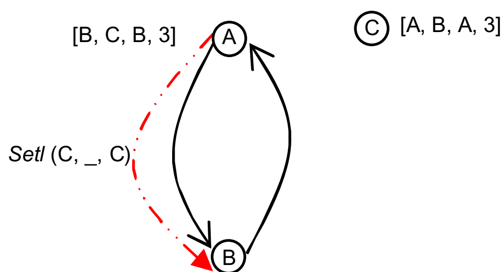

Figure 7.d Intra-ring update.

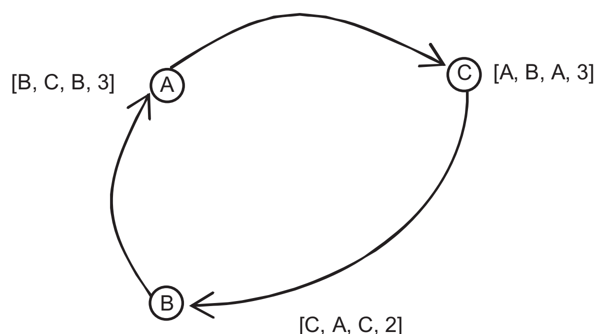

Figure 7.e New ring formation ends.

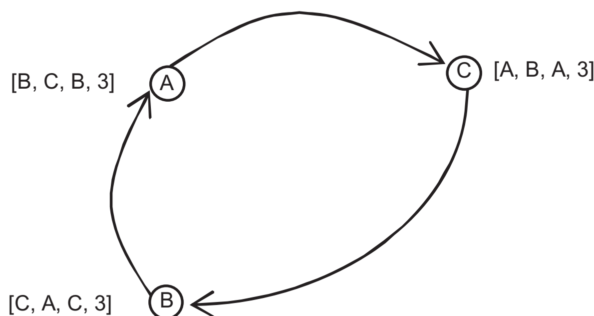

Figure 7.f Ring of three nodes.

pred = B, successor *succ* = C, and ring count *rc* = 3. Now, node A multicasts the foreign heap array, which it received from node C previously, along with the new ring count. Subsequently, node C updates its max heap (Fig. 7.e). Finally, nodes A, B and C merge into a single ring and node B sets its predecessor *pred* = C, successor *succ* = A, and ring count *rc* = 3 (Fig. 7.f).

6.4. Formation of clusters by merging of two isolated nodes

Assume, an isolated node A, having performance metric 100, detects another isolated node B, having performance metric 56, in its neighborhood. In order to initiate the cluster merge operation, node A sends the message *Req_msg* (⟨A, 100⟩, 1) to node B (Fig. 8.a). Now, node B compares the entries in the received *Req_msg* message with its own and finds that its performance metric value is smaller than the one contained in the message. Thus, it sends its heap and heap size along with message *Become_ldr* (⟨B, 56⟩, 1) to node A (Fig. 8.b). On receiving *Become_ldr* message, node A sends *Mbr_ACK* message to node B (Fig. 8.c). Node A merges its local heap with the foreign heap received. Now, both nodes A and B have become members of the same cluster with node A as cluster head and have heap array as [⟨A, 100⟩, ⟨B, 56⟩] and heap size 2.

6.5. Representing heap as an array

A heap can be represented by an array starting with index 1 and the left child of node at index i is at index $2i$, right child at index $2i + 1$ and the parent at index $i/2$ (discard fractional part). The heap shown in the following Fig. 9 can be represented as the following array.

$[\langle 16,1\rangle, \langle 14,4\rangle, \langle 10,2\rangle, \langle 8,7\rangle, \langle 7,8\rangle, \langle 9,12\rangle, \langle 3,6\rangle]$

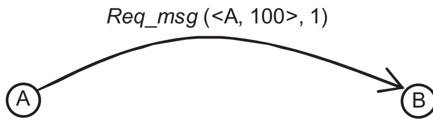

Figure 8.a Cluster merge begins.

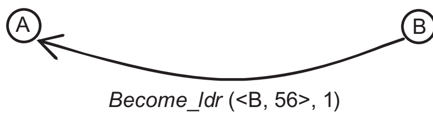

Figure 8.b Leader request.

Figure 8.c Member request.

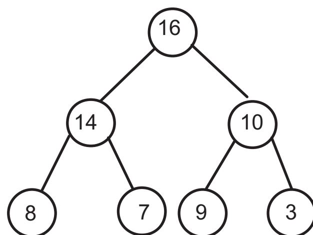

Figure 9 Heap.

Each element is shown as tuple $\langle i, j\rangle$ where the performance metric *pm* of the node is depicted as i and j is the unique *Node_id* that is not shown in the tree structure, for the sake of clarity.

6.6. Merging of two large clusters

Assume, a cluster head node A, having performance metric 16 and heap size 10, detects another cluster head node U, having performance metric 32 and heap size 6, in its neighborhood. In order to initiate cluster merge operation, say, node A sends the message *Req_msg* (⟨A, 16⟩, 10) to node U. Now, node U compares the entries in the received *Req_msg* message with its own and finds that its performance metric value is larger than the one contained in the message. Thus, it sends its heap H_U and heap size along with message *Become_mbr* (⟨U, 32⟩, 6) to node A. On receiving *Become_mbr* message, node A sends its heap H_A and heap size along with message *Ldr_ACK* to node U. Node A multicasts H_U to all its cluster members and node U multicasts H_A to all its cluster members. Thus, each cluster member gets a complete heap array. Now, the merged cluster has cluster head U and heap size 16.

6.7. Merging of two rings with heap

We identify each ring with its heap and leader, e.g. H_D represents the heap with node D as its ring leader. Assume, a ring node B having heap H_D and ring count 4 detects in its neighborhood another node F, from a different ring, having heap H_E and ring count 4 (Fig. 10.a). In order to initiate the ring merge operation, say, node B sends the message *Setf* (B, C, D, 4, H_D) to node F. Now, node F sends the message *Setf* (G, F, E, 4, H_E) to node B and sends the message *Setl* (_, C, D) to node G. Subsequently, node B sends the message *Setl* (_, _, F) to node A and *Setl* (G, _, _) to node C. Also, node G sends the message *Setl* (_, _, C) to node H. Now, node B multicasts foreign heap H_E to all its local ring members and similarly, node F multicasts foreign heap H_D to all its local ring members. Finally, the nodes merge their respective local heap with the foreign heap just received. Now, the merged ring has leader node E and ring count 8.

The sequence of messages that are exchanged during above merge operation is as follows:

Node B detects F in its neighborhood (Fig. 10.a). Node B sends a message *Setf* (B, C, D, 4) to F with its own heap array

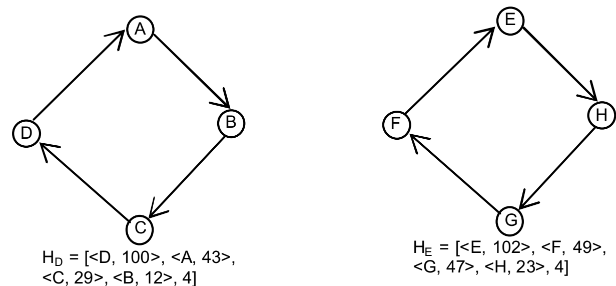

Figure 10.a Two isolated rings.

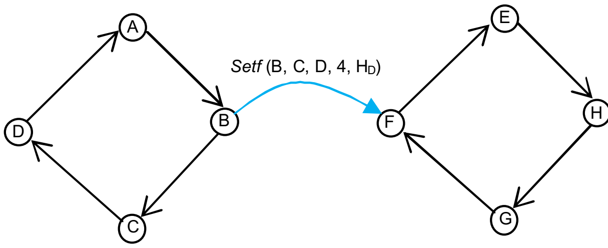

Figure 10.b Ring merge begins.

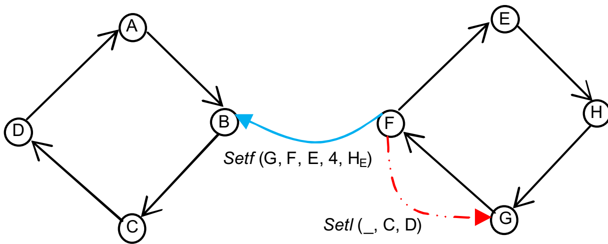

Figure 10.c Ring merge continued.

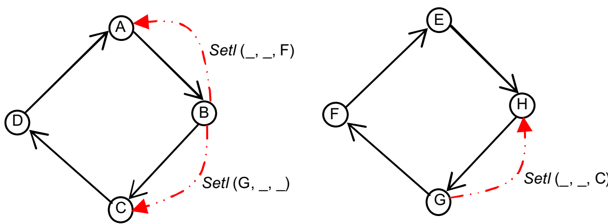

Figure 10.d Ring merge ends.

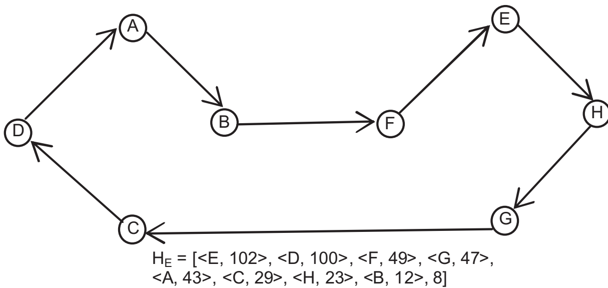

Figure 10.e Composite ring.

(Fig. 10.b). Node F upon receiving *Setf* (B, C, D, 4) sends *Setl* (_, C, D) to G and *Setf* (G, F, E, 4) to B (with its own heap array) and sets its *pred* = B (Fig. 10.c). Node G on receiving *Setl* (_, C, D), sends *Setl* (_, _, C) to H and sets its *succ* = C and *s_succ* = D (Fig. 10.d). Node H on receiving *Setl* (_, _, C) sets its *s_succ* = C. Node B on receiving *Setl* (G, F, E, 4) sends message *Setl* (_, _, F) to A and message *Setl* (G, _, _) to C and sets its *succ* = F and *s.succ* = E (Fig. 10.d). Node A on receiving *Setl* (_, _, F) sets its *s_succ* = F. Node C on receiving *Setl* (G, _, _) sets its *pred* = G. Finally, both B and F multicast the new ring count and the heap-arrays that they received while merging the rings to the members of their respective former rings (Fig. 10.e). It marks the end of current ring merge operation.

6.8. Ring formation process when one ring has rc = 3 and the other ring has rc = 2

We have explained the ring formation process in Sections 6.1–6.3 when an isolated node joins another isolated node or another ring. For the benefit of reader, we illustrate the ring formation process with the help of a more general example where two rings merge to form a bigger composite ring. Assume, node A that is part of a ring detects node D, in its neighborhood, which is part of another ring (Fig. 11.a). Now, node A sends *Setf* message to node D (Fig. 11.b). Subsequently, node D replies with a *Setf* message to node A and also sends *Setl* message to node E in its ring (Fig. 11.c). Now, node A sends *Setl* message to both nodes B and C in its ring (Fig. 11.d). Now, node A sets node D as its successor, updates ring count and multicast its modified heap array to its former ring members B and C (Fig. 11.e). Node D updates ring count and sends its modified heap array to its former ring

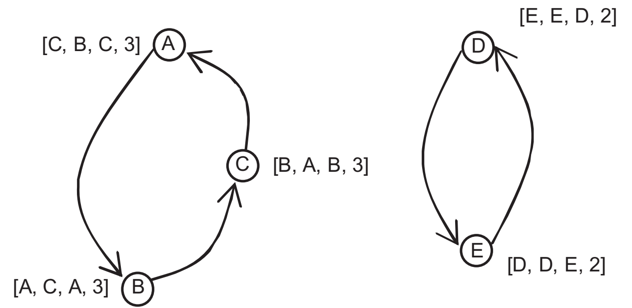

Figure 11.a Two unequal rings.

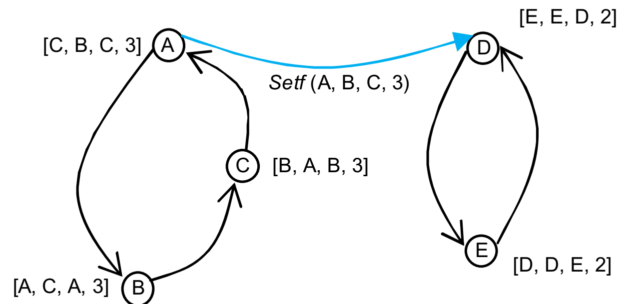

Figure 11.b Ring merge begins.

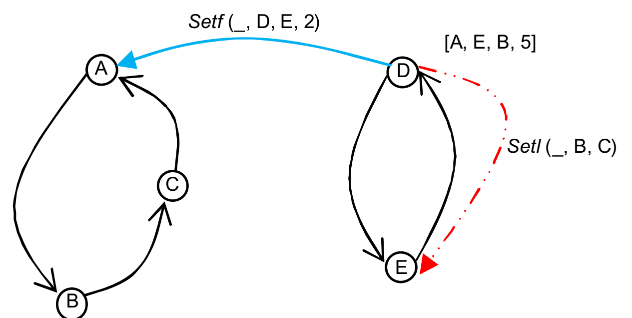

Figure 11.c Ring merge continued.

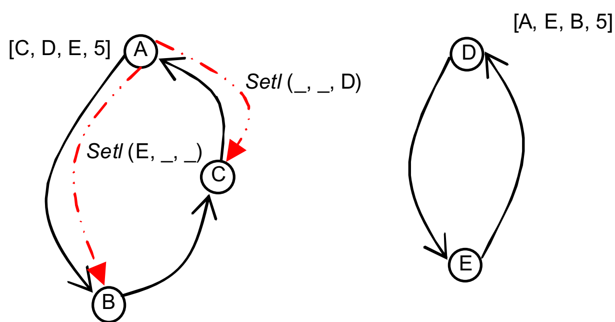

Figure 11.d Intra-ring update.

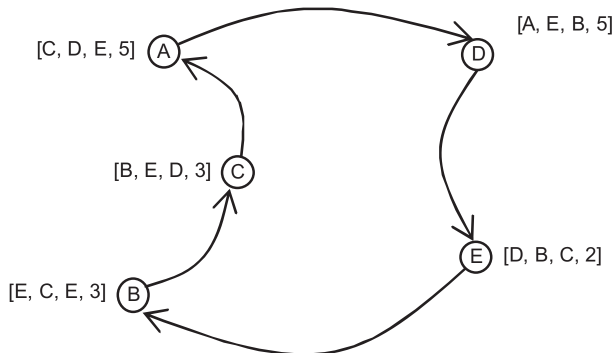

Figure 11.e Ring merge ends.

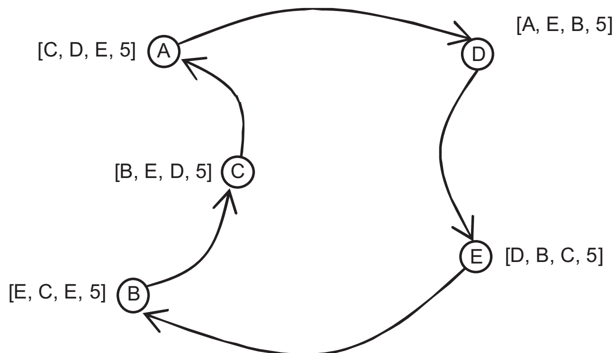

Figure 11.f Ring of five nodes.

member E. Node E sets node B as its successor and updates ring count (Fig. 11.f).

It is clear from the algorithm that no node can participate simultaneously in more than one ring merge operation and any two rings participating in a merge operation are directed ones and thus the virtual edge modification involves at most four nodes out of which at most two nodes are from each participating ring. Therefore, if both participating rings have a ring count $rc > 2$, even then the same procedure can accomplish the ring merge operation.

7. The failure handling

The most important feature of our approach is its failure handling capability. In our approach, we elect a leader from the ring of cluster heads and hence, the leader also acts as head of its cluster. Each cluster head has the heap containing all cluster heads as elements. If a cluster head detects the leader failure, we call it the initiator. The initiator waits for time out so that the cluster, whose head has failed, may find a new cluster head. If such a cluster head is available before time out, then the initiator includes ID of the new cluster head in its heap; otherwise, it removes ID of the failed cluster head from its heap. Subsequently, it reorganizes its heap to compute a new leader. Thus, the new leader is computed without any delay because the mechanism uses only local information and it does not involve any message propagation in the ring. Subsequently, the initiator notifies in the ring the information about the new leader and the reorganized heap.

8. Discussion on the performance

In addition to better failure handling capability, our protocol promises improved performance over other techniques due to several other components. First, in our approach, the overlay network is a ring of cluster heads and enables a non-head node in a cluster to reach its cluster head in a single hop. It is well established that the ring is very efficient under heavy load conditions. Furthermore, compared with other logical architectures, a ring allows conflict-free two-way communications, supports node ordering, and provides cost-free status feedbacks of operations (Banerjee and King, 2009). However, if we model the network as a random graph, then the problem of building a ring architecture for a set of nodes in such a graph is NP-complete. It is also known as the Hamiltonian cycle. Thus, we resort to divide a problem instance into small instances that can be solved in reasonable times. Therefore, we organize the nodes into bounded-sized clusters and the clusters are linked with a ring structure. This topology makes it easy for us to monitor the system, update software, synchronize clocks, detect abnormal conditions, and reorganize the network (Banerjee and King, 2009). Second, each time the leader is lost, the highest weight (a.k.a. priority) node among available ones would become the leader. Thus, we need to implement a priority queue. The low memory overhead of the heap makes it an excellent choice for maintaining a priority queue. It is simple and faster than other implementations because the worst case times for both 'insert' and 'remove' operations are logarithmic in the number of values stored in the priority queue. Hence, heap is the most appealing for this application and works well in common cases. Third, the cluster head holds the inter-cluster control information and performs traffic management. Therefore, it invites frequent topology updates from its cluster members in order to maintain the valuable information about their location and contacts. To a large extent, this information is consistent because its management is quasi-centralized. It is collected locally in the cluster head and then propagated to the other cluster heads using the virtual links and thereby minimizing the extent of mismatch (Shah et al., 2012; Abid et al., 2014) between logical and physical topologies in the event of node mobility/churn/failure or link losses. In fact, the node clustering is a popular practice to alleviate the topological dynamics in MANET. It stabilizes the end-to-end communication links and enhances the network scalability in order to control the routing overhead in large scale MANET. In addition, it restricts the node

count involved in the route establishment process (Gábos and Varga, 2012).

In our illustration, it is assumed that each node has a unique ID that is time/system invariant. Secondly, our merging clustering approach is bottom up in nature. The isolated mobile nodes self-organize into clusters based on their physical proximity and then elect a cluster head. Afterward, the cluster heads connect to each other in a virtual ring and finally the leader is elected. Therefore, the mismatch problem (Shah et al., 2012; Abid et al., 2014) is much less severe as compared to DHT-based protocols (Abid et al., 2014), where each node is assigned a logical unique ID, called LID that identifies a node in the logical namespace and describes the relative position of that node in the logical overlay. The neighborhood is defined on the basis of logical proximity and hence the neighbor nodes in the logical overlay may not be so close in the physical network. Therefore, the DHT-based protocols are more susceptible to the mismatch problem. Though DHT-based protocols have many other good features, the discussion is beyond the scope of the present exposition. The readers may refer (Abid et al., 2014) for an excellent survey and further details on the topic.

9. The simulation analysis

We have simulated our protocol using NS2 simulator for clustered mobile ad hoc networks under different network conditions with varying size and density. We have considered the wireless channel with two-ray ground radio propagation and omni-directional unity-gain antenna model. The WLAN is based on the direct sequence spread spectrum radio technology. We have opted IEEE 802.11 MAC protocol and the AODV protocol for routing. The interface queue gives priority to the routing protocol packets. The nodes have been assumed as randomly distributed in the network area. We assume the random waypoint mobility model and generate random movements for each simulation. The density μ of the network graph can be calculated as $\mu(r) = n\Pi r^2/A$, where n is the number of nodes in the network graph, A is the area of the network graph and r is the transmission range. Thus, $\mu(r)$ represents the average number of nodes within the transmission radius of each node. This allows us to test our algorithm with varying network density by considering the nodes with different transmission range, r. Our values range from $r = 75$, and $\mu(r) = 14$ (sparse network) to $r = 200$, and $\mu(r) = 102$ (dense network). By default, NS2 always uses a random number generator (RNG), named `defaultRNG`, with seed "1" to generate random numbers and thus the results obtained from every run are essentially the same. For the statistical analysis of a system, it is desirable to generate several distinct sets of results. Therefore, we introduce the diversity to each run by seeding different runs with different values $\langle n \rangle$. The simulation time has been set 200 s and the node pause time has been set 2 s.

The simulation experiments have been performed with our election algorithm to compute the election message overhead and the election latency under varying number of nodes, number of clusters and network area. Each plotted point in the graph represents the average value of 20 measurements. Thus, for each x-axis value, we run our simulation against 20 different randomly generated node locations and speeds but maintaining a common density pattern. The mobility rate has been varied from 20 m/s to 300 m/s.

There are many recent existing approaches of leader election, like Dagdeviren and Erciyes (2008), Sharma and Singh (2011), Shirmohammadi et al. (2009) and Singh and Sharma (2011a,b). However, the protocols (Sharma and Singh, 2011; Singh and Sharma, 2011a,b) are flat MANET protocols and the protocol (Shirmohammadi et al., 2009) is designed for

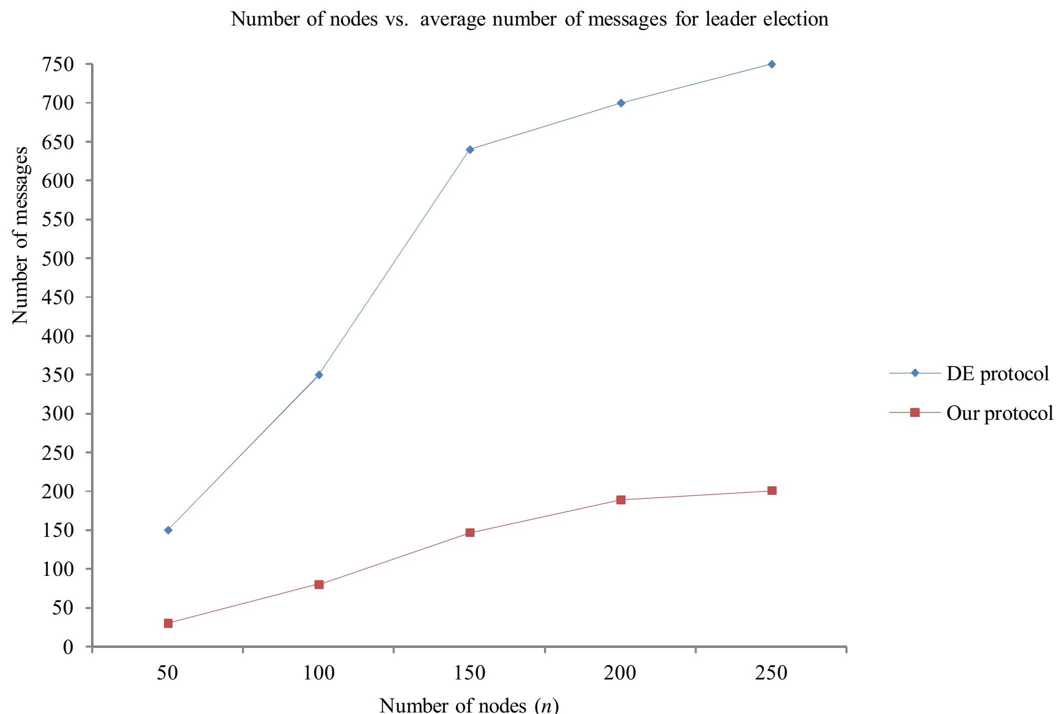

Figure 12 Nodes vs. election messages.

wireless sensor networks where the nodes are mostly static. Thus, we select Dagdeviren–Erciyes (Dagdeviren and Erciyes, 2008) protocol (DE protocol, for short) for comparison. Secondly, it is closest to our protocol because it also uses the multi-level hierarchical approach for leader election in MANETs. Therefore, we have compared the performance of our protocol with the DE protocol. In Fig. 12, on increasing the number of nodes, unlike our protocol, there is steep rise in the number of messages needed for leader election in the case of the DE protocol. The overall message requirement is 76% less in our protocol.

In Fig. 13, DE protocol manifests large election latency as compared to our protocol on varying the network area. In our protocol, the overall reduction in the leader election time has been noted as 82%. Also, the graph of our protocol is almost flat regardless of the network area.

In Fig. 14, the election latency increases exponentially with increase in the number of clusters in the case of the DE protocol. However, it is almost constant in the case of our protocol. The time to elect a new leader is 62.6% less in our case.

In Fig. 15, after the crash of the leader, the new leader information notification time increases faster, in the case of the DE protocol than the same in the case of our protocol, on increasing the number of clusters. The new leader information notification time has been found to be 44.5% less in our case.

In Fig. 16, the gradients of the plots for different mobility rates are very close in our protocol. Thus, we can conclude that the rate of increase of the election message count against the node count remains under control in low, medium and high mobility conditions. In fact, due to high speed node movement, some existing communication links may fail;

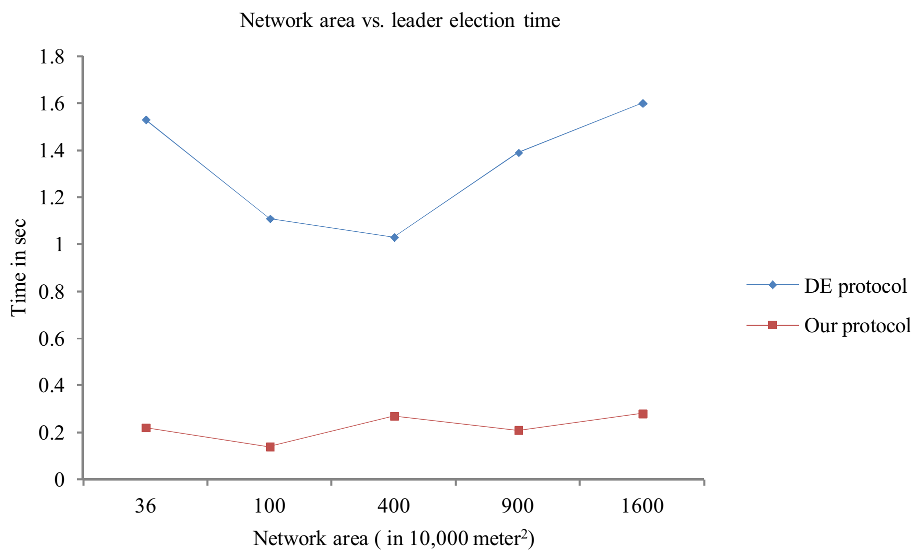

Figure 13 Network area vs. election latency.

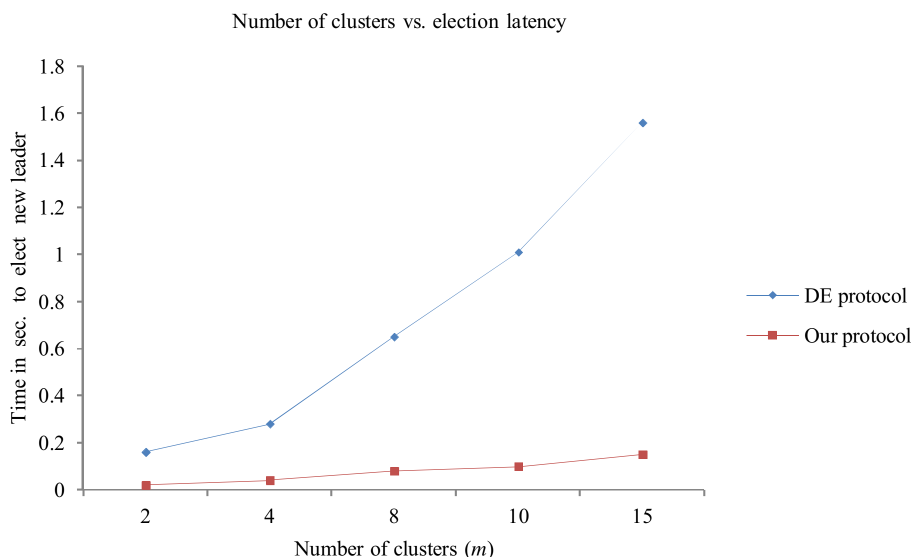

Figure 14 Number of clusters vs. election latency.

nevertheless, many new links may also be created. Secondly, due to clustering, only wide area (inter-cluster) node movement affects our logical structure and thus incurs limited maintenance overhead.

10. An alternative application

Our framework can also be used to perform efficient routing and data distribution. In the literature, there exist many other proposals also to achieve the routing efficiency through the core-based and mesh-based protocols. In the core-based protocol, a subset of nodes in the network is identified as the "core". The core nodes carry out special functions, such as routing paths construction, multicast membership management, and control/data packets propagation. The core is dynamically selected using a distributed algorithm and the number of core nodes is kept small. Every mobile node in the network must be adjacent to at least one core node and picks this core node as its dominator. However, in a mesh-based protocol, a set of interconnected nodes form mesh structures. The mesh structure is more robust than the tree structure when used in dynamic networks because a mesh provides alternate paths when link failure occurs. However, the cost for maintaining mesh structures is normally higher than tree. In a hybrid core-mesh protocol, the topology induced by the core nodes and the mesh connecting them form the virtual backbone that eases out the frequent route discovery and can be shared by both unicast and multicast routing. However, in large networks, the performance retards because the core-mesh maintenance overhead is significantly high (Biswas et al., 2004).

The implementation of mesh networks is easy provided the connection is simple as well as regular and the number of links per node is small. Secondly, the mesh networks render better

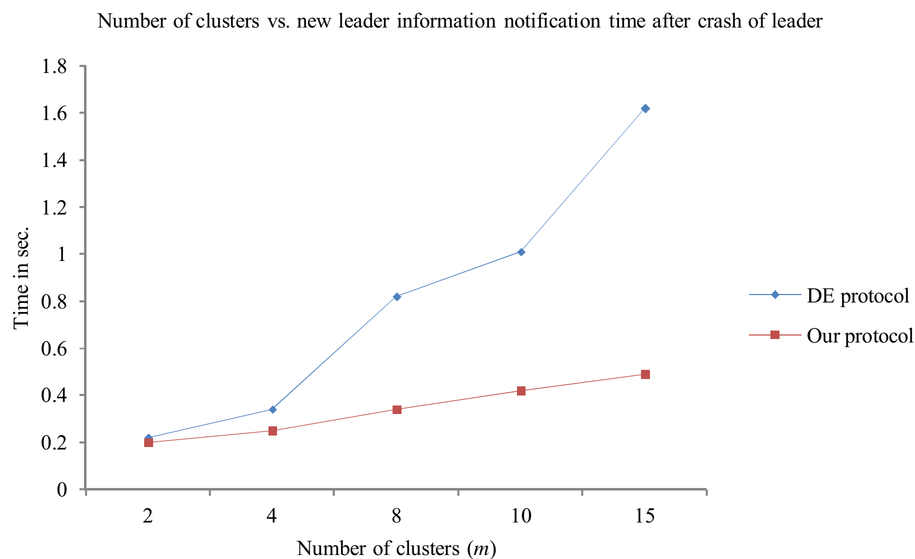

Figure 15 Number of clusters vs. new leader information notification time.

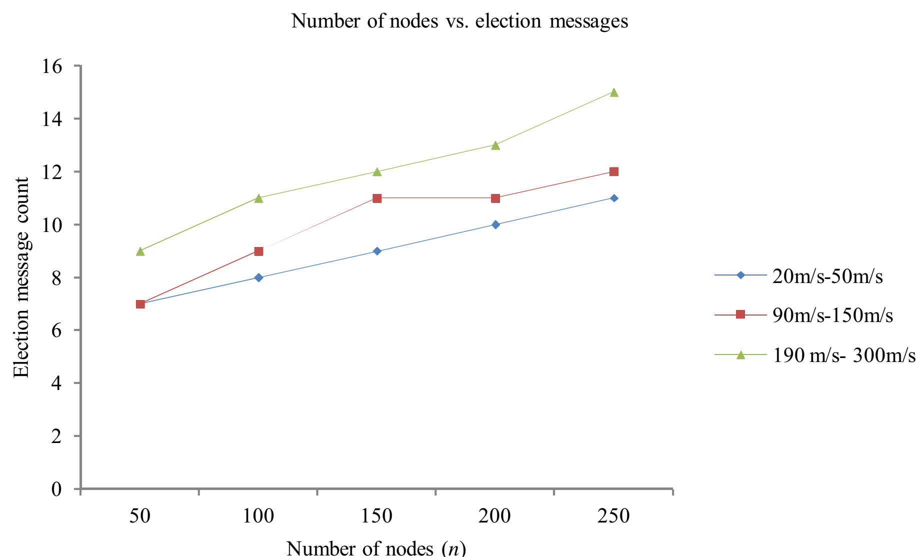

Figure 16 The impact of mobility.

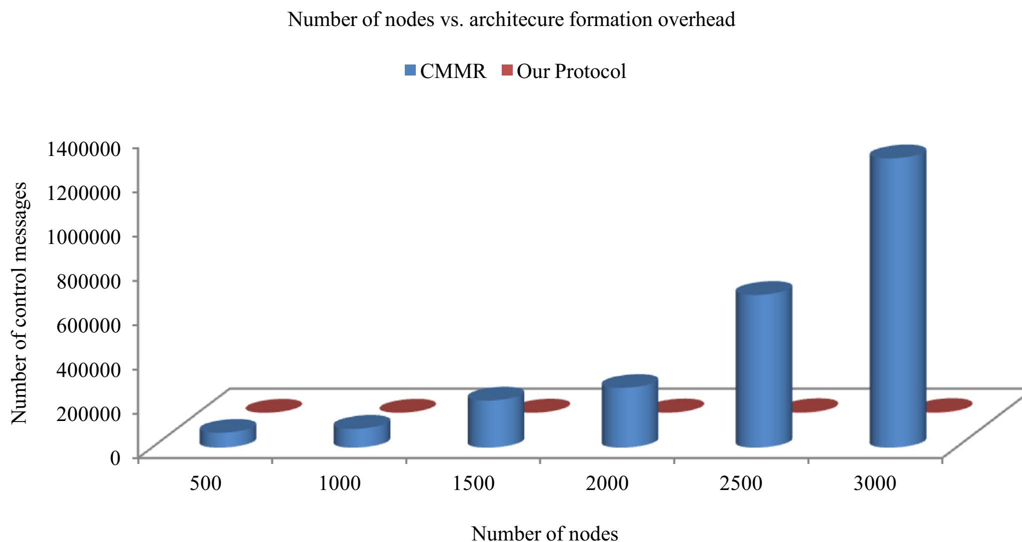

Figure 17 The architecture formation overhead.

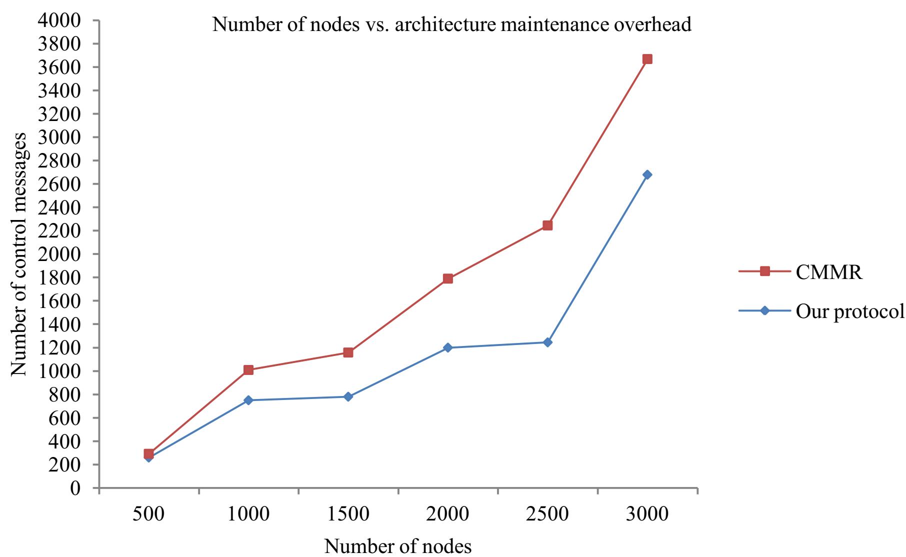

Figure 18 The architecture maintenance overhead.

scalability if the node degree is constant. For instance, with a network size of N nodes, the minimal incremental size is approximately $\sqrt{N}$ for the perfectly balanced network where the number of links per node is four. These properties are extremely difficult to ensure in the dynamic networks. Moreover, the large diameter ($\sqrt{N}$ for an N-node network) is a major limitation of the mesh network. Furthermore, a relatively small portion of algorithms for scientific and engineering problems efficiently fits the mesh topology (Louri and Sung, 1994).

The node clustering is a popular practice to alleviate the topological dynamics in MANET. It stabilizes the end-to-end communication links and enhances the network scalability in order to control the routing overhead in large scale MANET. In addition, it restricts the node count involved in the route establishment process (Gábos and Varga, 2012).

We propose a cluster-centric methodology to form the backbone (a.k.a. core) using cluster-heads. It reduces the scope of the core maintenance and makes it scalable and more stable against frequent node mobility. Our approach delivers better performance over the node-centric core-mesh strategies. It is confirmed by our simulation results that the composite transmission overhead of the proposed hierarchy is lower than that of recent core-mesh based hierarchical protocols, which claim better scalability. The EHMRP (Biswas et al., 2004) is a hybrid of differential destination multicast (DDM) Ji and Corson, 2001 and low overhead local clustering from MCEDAR (Sinha et al., 1999). In DDM, The header-encoded destination mechanism does not scale well with group size. Secondly, DDM requires the existence of a unicast routing protocol, though it does not limit itself to any particular unicast

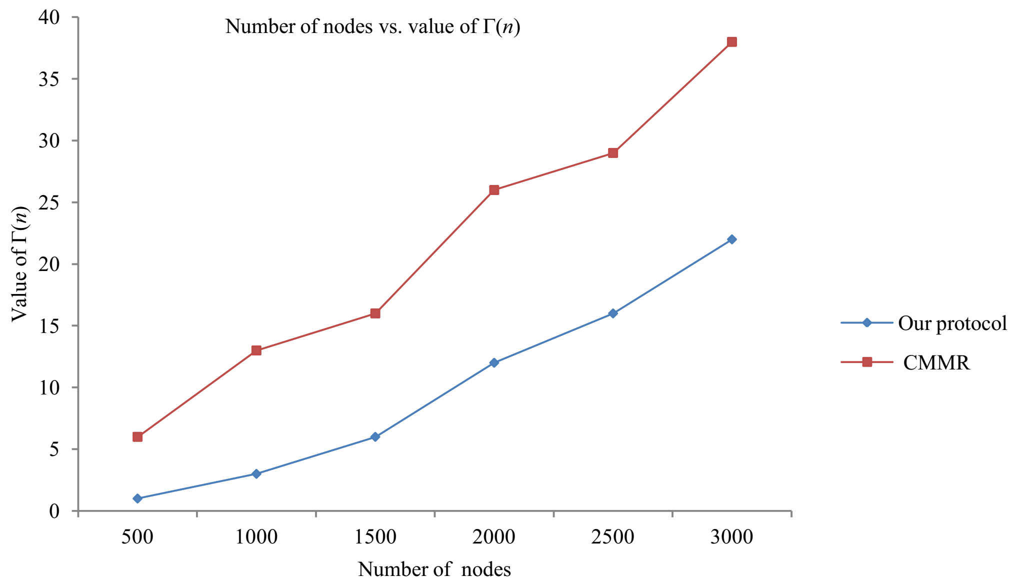

Figure 19 The comparison of scalability.

protocol. Furthermore, the MCEDAR performs core extraction in constant time; however, the core mesh is in fact a connected dominating set and thus the size of the mesh grows with the node degree in the network (Blum et al., 2004). The node degree is proportional to the communication range and node density, and thus the number of nodes in the core mesh may be large in practice causing the performance to suffer. Note that CMMR (Younis et al., 2011) is an improved version of MCEDAR. Hence, we compare the performance of our protocol with CMMR. It is clear from Fig. 17 that the number of control messages exchanged in the architecture formation is huge in CMMR than our protocol and further it increases exponentially with the increase in the number of nodes. Therefore, the initial formation as well as reconstruction would be economical for our architecture. Also, it can be computed from Fig. 18 that our protocol has 56.4% lower maintenance overhead than CMMR.

11. Discussion on scalability

The authors of Chiang and Yang (2004) argue that while characterizing the scalability, maximum size should not be the only point of concern. Some network architectures may handle a large number of nodes; however the quality of service (QoS) may suffer seriously with the increase in the number of nodes. A scalable architecture should handle a large number of nodes without sacrificing QoS too much. In general, they quantify this tradeoff as a *size-QoS product*, which is the product of maximum network size and some measure of QoS. Note that, many QoS metrics, e.g. reliability and latency, are monotonic functions of the average number of hops $H(n)$ required to reach one node from another, if there are n nodes in the network. Thus, they use $H(n)$ as the QoS metric in their illustration. They define a new metric $\Gamma(n)$, essentially as the reciprocal of the *size-QoS product*:

Definition. $\Gamma(n)$ is the product of $H(n)$ and $M(n)$, where $M(n)$ is the size of the bottleneck network element needed to support n nodes.

It is obvious from the above definition that a smaller $\Gamma(n)$ implies a more scalable network. The graph shown in Fig. 19 confirms the intuition that the ring network is more scalable than mesh network. Further, it has been verified that our architecture is overall 56% more scalable.

In addition, compared with other architectures, a ring allows conflict-free two-way communications, supports node ordering, and provides cost-free status feedbacks of operations (Banerjee and King, 2009). However, if we model the network as a random graph, then the problem of building ring architecture for a set of nodes in such a graph is NP-complete. It is also known as the Hamiltonian cycle. Thus, we resort to divide a problem instance into small instances that can be solved in reasonable times. Therefore, we organize the nodes into bounded-sized clusters and the clusters are linked with a ring structure. This topology makes it easy for us to monitor the system, update software, synchronize clocks, detect abnormal conditions, and reorganize the network (Banerjee and King, 2009).

12. Conclusion

The proposed approach guarantees fast and fault tolerant leader election in mobile ad hoc networks. The simulation results substantiate the fact that our approach delivers reduced message count, low election latency as well as short leader notification time against high node count, wide network area, and a large number of clusters. Thus, our approach manifests better scalability. Moreover, the performance does not degrade heavily under various mobility conditions. Consequently, the pause time during the execution of leader based mobile computing applications is reduced significantly. Therefore, as a desirable offshoot, the applications achieve enhanced throughput. Further, we refrain from developing a new leader election protocol, but present a framework for hierarchical leader election in MANETs. Based on the framework, a variety of techniques can be adopted for effective distributed synchronization and efficient execution of other distributed applications, e.g. mutual exclusion and consensus, in MANETs.

Appendix I

We assume a single node has ring count 1 and it can send a message *Setf* ⟨*curr*, *curr*, _, 1⟩ to initiate the merge operation, where *curr* is the identity of the node itself. Similarly, the nodes with ring count 2 can send a message *Setf* ⟨*curr*, *succ*, _, 2⟩ to initiate the merge operation. The local ring count is represented as

```
lrc: local ring count;
frc: foreign ring count;
1. On detecting a foreign node in the neighborhood//initiate ring merge
begin//0 begin (procedure to send Setf message to initiate ring merge)
while (new cluster head is chosen or timeout occurs)
   begin//1 begin
      send Blk message to its pred and succ, if any;
      if Blk_ack are received then
      //2 begin
         a. if(lrc = = 1) send Setf ⟨curr, curr, _, 1, CH_heap⟩ to foreign neighbor
         b. if(lrc = 2) send Setf ⟨curr, succ, _, 2, CH_heap⟩ to foreign neighbor
         c. if (lrc > 2) send Setf ⟨succ, s_succ, lrc, CH_heap⟩ to foreign neighbor
      //2 end
      set nmop = 0;
   end while   //1 end
end main   //0 end
2. On receiving Setf ⟨A, B, C, frc, CH_heap) message   //frc is foreign ring count and nmop = 1
begin   //1 begin
if (nmop = = 1 && sender.id = = Idn)
begin   //2 begin
      if (lrc = = 1)   //5 begin
         send Setf ⟨curr, curr, _, 1⟩ to the sender of Setf message
         if (A ≠ '_') set pred = A else skip
         if (B ≠ '_') set succ = B else skip
         if (C ≠ '_') set s_succ = C else skip
         set lrc = lrc + frc
         merge local heap with foreign heap
      endif   //5 end
      if (lrc = = 2)   //6 begin
         if (frc = = 1)   //7 begin
         begin
         send Setf ⟨curr, succ, _, 2) to the sender of Setf message
         set s_succ = succ
         set succ = B
         set lrc = lrc + frc
         send Setl ⟨B, _, B) to pred
         multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
         end   //7 end
         if (frc > = 2)   //8 begin
         begin
         send Setf ⟨curr, succ, _, 2) to the sender of Setf message
         set s_succ = B
         set pred = A
         set lrc = frc + lrc
         send Setl ⟨_, B, A⟩ to succ
         multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
         endif   //8 end
      endif   //6 end
else if (lrc > 2)
begin//9 begin
      if (frc = = 1)   //10 begin
      begin
send Setl ⟨B, _, _⟩ to succ
      set s_succ = succ
      set succ = B
      send Setl ⟨_, _, B⟩ to pred
      multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
      end   //10 end
      if (frc > = 2)   //11 begin
```

(*continued on next page*)

```
        begin
        send Setl ⟨A, _, _⟩ to succ
        set s_succ = C, if C = = '_' then set succ = A
        set succ = B
        send Setl ⟨_, _, B) to pred
        multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
        endif   //11 end
endif   //9 end
endif   //2 end
else   //4 begin
   send Rej message to the sender
endelse   //4 end
end   //1 end
3. On receiving Setf ⟨A, B, C, frc, CH_heap⟩ when nmop = 0
begin   //1 begin
   send Blk ⟨pm, Node_id, 1⟩ to pred
   If Blk_tm expired then send Rej message to the sender of Setf message
   else if (lrc = = 1)   //2 begin
      begin
         if (A ≠ '_') set pred = A else skip
         if (B ≠ '_') set succ = B else skip
         if (C ≠ '_' && frc ≠ 2) set s_succ = C else if (frc = 2) set s_succ = pred
         set lrc = lrc + frc
         merge local heap with foreign heap
      endif   //2 end
   if (lrc = = 2)   //3 begin
      begin
         if (frc = = 1)   //4 begin
         set succ = A
         set s_succ = pred
         send Setl ⟨A, _, A) to pred
         multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
         endif   //4 end
         if (frc = = 2)   //5 begin
         set succ = A
         set s_succ = B
         send Setl ⟨B, _, A⟩ to pred
         set lrc = lrc + frc
         multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
         endif   //5 end
         else if (frc > 2)   //6 begin
         begin
            set pred = A
            send Setl ⟨_, B, C) to succ
            set s_succ = B
            multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
         end   //6 end
   endif   //3 end
   if (lrc > 2)   //7 begin
      begin
         if (frc = = 1)   //8 begin
            begin
            send Setl ⟨_, A, curr) to pred
            set pred = A
            multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
            end   //8 end
         if(frc = = 2)   //9 begin
            set pred = A
            send Setl ⟨_, B, A⟩ to pred
            multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
            endif   //9 end
         if(frc > 2)   //10 begin
            set pred = A
            send Setl ⟨_, B, C) to pred
            multicast the foreign heap and new ring count to members of local heap and merge foreign heap with local heap
      endif   //10 end
```

```
  endif   //7 end
end   //1 end
4. On receiving Blk ⟨pm, Node_id, f⟩
a.   if (nmop = 0 && f = 0) send Blk_ack to the sender.
else if (nmop = 0 && f = 1) then send Blk message to pred with f = 0 and wait for Blk_ack from pred
b.   if Blk_tm times out then do nothing
else if Blk_ack is received then send Blk_ack to the sender of Blk message
5. On receiving Blk_ack message
if all the needed Blk_ack messages have been received within Blk_tm, continue with the merge operation
else if node itself is not the initiator then send Rej message to the initiator
6. On receiving Setl ⟨A, B, C⟩ message
      If (B ≠ '_') send Setl ⟨_, _, B⟩ to s_succ.//begin 1
      begin
         if A ≠ '_' set pred = A
         if B ≠ '_' set succ = B
         if C ≠ '_' set s_succ = C
      endif//end 1
7. On receiving Rej message
  Reset Time_out timer and wait
```

References

Abid, S.A., Othman, M., Shah, N., 2014. A survey on DHT-based routing for large-scale mobile ad hoc networks. ACM Comput. Surv. 47 (2), 20:1–20:46.

Attiya, H., Welch, J.L., 2004. Distributed Computing: Fundamentals, Simulations, and Advanced Topics. John Wiley & Sons (Chapter 3).

Banerjee, A., King, C.-T., 2009. Building ring-like overlays on wireless ad hoc and sensor networks. IEEE Trans. Parallel Distrib. Syst. 20 (11), 1553–1566.

Basagni, S., Mastrogiovanni, M., Petrioli, C., 2004. A performance comparison of protocols for clustering and backbone formation in large scale ad hoc network. In: IEEE MASS, pp. 70–79.

Biswas, J., Barai, M., Nandy, S.K., 2004. Efficient hybrid multicast routing protocol for ad-hoc wireless networks. In: 29th Annual IEEE Int. Conf. on Local Computer Networks (LCN'04), pp. 180–187.

Blum, A.T.J., Ding, M., Cheng, X., 2004. Connected dominating set in sensor networks and MANETs. In: Du, D.-Z., Pardalos, P. (Eds.), Book Chapter, Handbook of Combinatorial Optimization. Kluwer Academic Publishers, Dordrecht, pp. 329–369.

Boukerche, A., Abrougui, K., 2007. An efficient leader election protocol for wireless quasi-static mesh networks: proof of correctness. In: IEEE ICC, pp. 3491–3496.

Chang, E.J.H., Roberts, R., 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processors. Commun. ACM 22 (5), 281–283.

Chiang, M., Yang, M., 2004. Towards network x-ities from a topological point of view: evolvability and scalability. In: 42nd Annual Allerton Conf. on Communication, Control, and Computing, Allerton House, Monticello, Illinois, pp. 532–541.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. Introduction to Algorithm. McGraw Hill Book Co., NY.

Dagdeviren, O., Erciyes, K., 2006. A distributed backbone formation algorithm for mobile ad hoc networks. In: Guo, M. et al. (Eds.), ISPA 2006, LNCS 4330. Springer, Heidelberg, pp. 219–230.

Dagdeviren, O., Erciyes, K., 2008. A hierarchical leader election protocol for mobile ad hoc networks. In: Bubak, M. et al. (Eds.), ICCS 2008, Part I, LNCS 5101. Springer, Heidelberg, pp. 509–518.

Dagdeviren, O., Erciyes, K., Cokuslu, D., 2005. Merging clustering algorithms in mobile ad hoc networks. In: Chakraborty, G. (Ed.), ICDCIT 2005, LNCS 3816. Springer, Heidelberg, pp. 56–61.

Derhab, A., Badache, N., 2008. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. IEEE Trans. Parallel Distrib. Syst. 19 (7), 926–939.

Gábos, D., Varga, M., 2012. Cluster formation in wireless mesh networks. Acta Univ. Sapientiae Electr. Mech. Eng. 4, 5–32.

Gallager, R.G., Humblet, P.A., Spira, P.M., 1983. A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. 5 (1), 66–77.

Garcia-Molina, H., 1982. Elections in distributed computer systems. IEEE Trans. Comput. C-31 (1), 48–59.

Garg, V.K., 2004. Concurrent and Distributed Computing in Java. Wiley (Chapter 13).

Ghosh, S., 2010. Distributed Systems: An Algorithmic Approach. Chapman & Hall/CRC Computer & Information Science Series, Taylor & Francis (Chapter 11).

Haddar, M.A., Kacem, A.H., Metivier, Y., Mosbah, M., 2008. Electing a leader in the local computation model using mobile agents. In: IEEE/ACS AICCSA, pp. 473–480.

Han, B., Jia, W., 2007. Clustering wireless ad hoc networks with weakly connected dominating set. J. Parallel Distrib. Comput. 67, 727–737.

Hirschberg, D.S., Sinclair, J.B., 1980. Decentralized extrema-finding in circular configurations of processors. Commun. ACM 23 (11), 627–628.

Ingram, R., Shields, P., Walter, J.E., Welch, J.L., 2009. An asynchronous leader election algorithm for dynamic networks. In: IEEE IPDPS, pp. 1–12.

Jain, A.K., Sharma, R., 2012. Leader election algorithm in wireless environments using fibonacci heap structure. Int. J. Comput. Technol. Appl. 3 (3), 871–873.

Ji, L., Corson, M.S., 2001. Differential destination multicast – a MANET multicast routing for multihop ad hoc network. IEEE INFOCOM 2, 1192–1201.

Louri, A., Sung, H., 1994. An optical multi-mesh hypercube: a scalable optical interconnection network for massively parallel computing. J. Lightwave Technol. 12 (4), 704–716.

Malpani, N., Welch, J.L., Vaidya, N.H., 2000. Leader election algorithms for mobile ad hoc networks. In: ACM DIAL M workshop, pp. 96–103.

Raychoudhury, V., Cao, J., Niyogi, R., Wu, W., 2014. Top k-leader election in mobile ad hoc networks. Pervasive Mob. Comput. 13, 181–202.

Raynal, M., 2013. Distributed Algorithms for Message-Passing Systems. Springer-Verlag, Berlin, Heidelberg (Chapter 4).

Raz, D., Shavitt, Y., Zhang, L., 2004. Distributed council election. IEEE/ACM Trans. Netw. 12 (3), 483–492.

Shah, N., Qian, D., Wang, R., 2012. MANET adaptive structured P2P overlay. Peer-to-Peer Netw. Appl. 5 (2), 143–160.

Sharma, S., Singh, A.K., 2011. Democratic leader finding algorithm for large mobile ad hoc networks. In: ICDCS Workshops, pp. 304–312.

Shirmohammadi, M., Chhardoli, M., Faez, K., 2009. CHEFC: cluster head election with full coverage in wireless sensor networks. In: IEEE MICC, pp. 780–784.

Singh, A.K., Sharma, S., 2011. Elite leader finding algorithm for MANETs. In: IEEE ISPDC, pp. 125–132.

Singh, A.K., Sharma, S., 2011. Message efficient leader finding algorithm for mobile ad hoc networks. In: IEEE COMSNETS, pp. 1–6.

Sinha, P., Sivakumar, R., Bharghavan, V., 1999. MCEDAR: multicast core extraction distributed ad hoc routing. In: IEEE WCNC'99, pp. 1313–1317.

Subathra, P., Radha Krishnan, P., Sivagurunathan, S., 2012. A cluster-based reliable token circulation scheme for group communication in MANET. Arabian J. Sci. Eng. 37 (3), 647–664.

Vasudevan, S., DeCleene, B., Immerman, N., Kurose, J., Towsley, D., 2003. Leader election algorithms for wireless ad hoc networks. In: IEEE DISCEX, pp. 1–12.

Vasudevan, S., Kurose, J., Towsley, D., 2004. Design and analysis of a leader election algorithm for mobile ad hoc networks. In: IEEE ICNP, pp. 350–360.

Yen, Li-H., Chi, K.-H., 2004. Maintaining a ring structure for mobile ad hoc computing. J. Parallel Distrib. Comput. 64, 1371–1379.

Younis, M., Farrag, O., Lee, S., 2011. Cluster mesh based multicast routing in MANET: an analytical study. In: IEEE Int. Conf. Comm., pp. 1–6.

Zeng, Y., Mittal, N., Venkatesan, S., Chandrasekaran, R., 2010. Fast neighbor discovery with lightweight termination detection in heterogeneous cognitive radio networks. In: 9th IEEE ISPDC, pp. 149–156.