# TABLE OF CONTENTS

## LIST OF FIGURES

TABLE OF CONTENTS

## LIST OF TABLES

### CHAPTER 1. INTRODUCTION

1

### CHAPTER 2. BACKGROUND

5

#### 2.1. Clustering Algorithms

5

- 2.1.1. Spanning Tree Based Algorithms

6

- 2.1.2. Dominating Set Based Algorithms

9
  - 2.1.2.1. Clustering Using IDS

9
  - 2.1.2.2. Clustering Using WCDS

10
  - 2.1.2.3. Clustering Using CDS

10

- 2.2. Backbone Formation Algorithms

12

- 2.3. Distributed Mutual Exclusion Algorithms

16
  - 2.3.1. Performance Metrics

16
  - 2.3.2. Ricart-Agrawala Algorithm

16
  - 2.3.3. Token-Based Algorithms

17
  - 2.3.4. Mutual Exclusion Algorithms on MANET

18

### CHAPTER 3. MERGING CLUSTERING ALGORITHM

21

#### 3.1. General Idea and Description of the Algorithm

21

#### 3.2. An Example Operation

25

#### 3.3. Analysis

29

#### 3.4. Results

30

### CHAPTER 4. BACKBONE FORMATION ALGORITHM

38

#### 4.1. General Idea and Description of the Algorithm

38

#### 4.2. An Example Operation

44

#### 4.3. Analysis

45

#### 4.4. Results

47
CHAPTER 5. MOBILE RICART-AGRAWALA ALGORITHM

5.1. General Idea and Description of the Algorithm
5.2. An Example Operation
5.3. Analysis
5.4. Results

CHAPTER 6. CONCLUSION

APPENDIX A. SIMULATION SETUP
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>(a)IDS (b)WCDS (c)CDS</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Finite State Machine of the Merging Clustering Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Message Flow Diagram of Merging Clustering Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Clusters obtained using Merging Clustering Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>MANET with 40 nodes located on a surface area of 600m × 600m</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Clusters of MANET with 40 nodes</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Run-time of Merging Clustering Algorithm</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Total Message Numbers of Merging Clustering Algorithm</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Cluster Quality of MCA Against K for 40 nodes</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Cluster Quality of MCA Against K for 60 nodes</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Cluster Quality of MCA Against Mobility for 40 nodes</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Cluster Quality of MCA Against Surface Area for 40 nodes</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Cluster Quality of MCA Against Total Number of Nodes</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Total Edge-cut for Merging Clustering Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Average Edge-cut for Merging Clustering Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Total Edge-cut Against K for Merging Clustering Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>Average Edge-cut Against K for Merging Clustering Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>Total Edge-cut for MCA Against Mobility</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Average Edge-cut for MCA Against Mobility</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>Total Edge-cut for MCA Against Surface Area</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3.20</td>
<td>Average Edge-cut for MCA against Surface Area</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Finite State Machine of the Backbone Formation Algorithm</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>A MANET with its minimum spanning tree</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>An Example Operation for Backbone Formation Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Run-time Performance for Backbone Formation Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Round-trip Delay Against Mobility for BFA</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Round-trip Delay Against Surface Area for BFA</td>
<td>48</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Lower Bounds for Performance Metrics</td>
<td>17</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Performance Metrics of Ricart-Agrawala Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Performance Metrics of General Token-Based Algorithms</td>
<td>18</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Cluster Formation for 20 nodes</td>
<td>27</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Cluster Formation for 40 nodes</td>
<td>29</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Cluster Formation for 40 nodes, continued</td>
<td>29</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Comparison of DAMWST and MCA</td>
<td>30</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Comparison of Backbone Constructing Algorithms</td>
<td>46</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Comparison of Backbone Constructing Algorithms cont...</td>
<td>46</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Comparison of Mobile_RA and RA</td>
<td>55</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Wireless communication is growing fast in the last few years. Future information technology will be mainly based on wireless technology. Traditional cellular and mobile networks are still, in some sense, limited by their need for infrastructure. For mobile ad hoc networks (MANETs), this final limitation is eliminated. Ad hoc networks are key to the evolution of wireless networks. MANETs are non-fixed infrastructure networks which consist of dynamic collection of nodes with rapidly changing topologies of wireless links. Although military tactical communication is still considered the primary application for ad hoc networks, commercial interest in this type of networks continues to grow. Applications such as rescue missions in times of natural disasters, law enforcement operations, commercial and educational use of sensor networks, personal area networking are just a few possible commercial examples (Stojmenovic 2002). MANETs have the problems of bandwidth optimization, transmission quality, discovery, ad hoc addressing, self routing and power control. Power control is a very important issue in MANETs because nodes are powered by batteries only. Therefore, amount of communication should be minimized to avoid a premature drop out of a node from the network.

Clustering has become an important approach to manage MANETs. The clustering problem can be described as classifying nodes in a MANET hierarchically into equivalence classes with respect to certain attributes such as geographical regions or small neighborhood of 1 or 2 hops from special nodes called the clusterheads (Krishna et al. 1997). Under a cluster structure, mobile nodes may be assigned a different status or function, such as clusterhead, clustergateway or a cluster member. A clusterhead serves as a local coordinator for its cluster, performing intra-cluster transmission arrangement, data forwarding, and so on. A clustergateway is a non-clusterhead node with inter-cluster links, so it can access neighboring clusters and forward interaction between clusters. A cluster member is usually called an ordinary node, which is a non-clusterhead node without any inter-cluster links (Yu and Chong 2005). There are three main benefits of clustering. Firstly, clustering MANETs provides spatial reuse of resources to increase the system capacity (Hou and Tsai 2001, Lin and Gerla 1995). Two clusters may deploy the
same frequency or code set if they are not overlapping. Secondly, the set of clusterheads and clustergateways can normally form a virtual backbone for inter-cluster routing, and thus the generation and spreading of routing information can be restricted in this set of nodes. Thirdly, a cluster structure makes an ad hoc network appear smaller and more stable in the view of each mobile terminal ”(McDonald and Znati 1999). When a mobile node changes its attaching cluster, only mobile nodes residing in the corresponding clusters need to update the information. Thus, local changes need not be seen and updated by the entire network, and information processed and stored by each mobile node and the messaging complexity of upper layer protocols can be greatly reduced ”(Iwata et al. 1999, Chen et al. 1999, Erciyes 2004, 2005).

The mutual exclusion problem involves a group of processes, each of which intermittently requires access to a resource or a piece of code called the critical section (CS). At most one process may be in the CS at any given time. Providing shared access to resources through mutual exclusion is a fundamental problem in computer science, and is worth considering for the ad hoc environment, where stripped down mobile nodes may need to share resources ”(Walter et al. 2001a). In general, distributed mutual exclusion algorithms may be classified as permission based or token based. Suzuki-Kasami’s algorithm ”(Suzuki and Kasami 1985)” (N messages) and Raymond’s tree based algorithm ”(Raymond 1989)” (log(N) messages) are examples of token based mutual exclusion algorithms. Examples of nontoken-based distributed mutual exclusion algorithms are Lamport’s algorithm ”(Lamport 1978)” (3(N-1) messages), Ricart-Agrawala (RA) algorithm (2(N-1) messages) ”(Ricart and Agrawala 1981)” and Maekawa’s algorithm ”(Maekawa 1985)”. In permission based algorithms, a node would enter a critical section after receiving permission from all of the nodes in its set for the critical section. For token-based algorithms however, processes are on a logical ring and possession of a system-wide unique token would provide the right to enter a critical section. Safety, liveness and fairness are the main requirements for any mutual exclusion algorithm. Lamport’s algorithm and RA algorithm are considered as the only fair distributed mutual exclusion algorithms in literature. Distributed mutual exclusion in mobile networks is a relatively new research area. A fault tolerant distributed mutual exclusion algorithm using tokens is discussed in ”(Walter et al. 2001a)” and a k-way mutual exclusion algorithm for ad hoc wireless networks where there may be k processes executing a critical section at any time is presented.
Aim of this thesis is to design and implement a cluster based mutual exclusion algorithm for MANETs where the node counts in the clusters are balanced and a ring is formed by cluster heads to process Mobile Ricart-Agrawala Algorithm (Mobile_RA) algorithm” (Erciyes 2004, 2005). Firstly, MANET must be clustered to the balanced number of nodes in defined range to distribute the workload of the network more evenly. The clusters must be non-overlapped to provide unique clusterhead for each cluster. To solve these problems, we propose a graph theoretic clustering algorithm, Merging Clustering Algorithm (MCA)” (Dagdeviren et al. 2005, 2006), for clustering in MANETs using merging as in constructing Spanning Trees where part of a tree or a tree of a forest designates a cluster. Reference point of study of the MCA is the Gallagher et. al’s distributed minimum spanning tree algorithm which merges fragments by defined rules” (Gallagher et al. 1983). Gallagher et. al’s algorithm and other related work is reviewed in Chapter 2 in Sections 2.1.1 and 2.1.2. General idea of the MCA, illustration by an example, analysis and extensive simulation results are explained respectively in Sections 3.1 through 3.4 in Chapter 3.

After partitioning MANET into balanced clusters, we aim to construct a directed ring architecture from clusterheads to maintain the backbone. The background of backbone formation algorithms is given in section 2.2. The first step of this backbone formation algorithm is the construction of minimum spanning tree between clusterheads with respect to minimum number of hops or minimum distance to classify clusterheads as BACKBONE or LEAF clusterheads. The second step and the main idea is the formation of the ring architecture by two directed paths from BACKBONE clusterheads and LEAF clusterheads as described in Section 4.1 and illustrated in Section 4.2 by an example operation. Runtime performance and round-trip delay against mobility, surface area, clusterhead number and total number of nodes are shown in Section 4.4.

Lastly, after clustering MANET by MCA and constructing backbone architecture periodically, we aim to show the implementation considerations and the results of the Mobile_RA algorithm that was designed previously” (Erciyes 2005). MCA provides the clusterhead for each cluster which is same as the coordinator of Mobile_RA. Backbone formation algorithm finds the next clusterhead of each clusterhead to maintain the ring architecture across coordinators to perform the required critical section entry and exit.
procedures for the nodes. Using this architecture, we improve and extend the RA algorithm described in "(Erciyes 2004)" to MANETs and show that these algorithms may achieve an order of magnitude reduction in the number of messages required to execute a critical section at the expense of increased response times and synchronization delays which may also be useful in MANETs where energy efficiency, therefore message complexity is of paramount importance. Section 2.3 also provides a background of mutual exclusion algorithms for distributed systems and MANETs. The description of the Mobile_RA, illustration, analysis and extensive simulation results are provided respectively in Sections 5.1 through 5.4 in Chapter 5. Information about implementations and simulation environment is given in Appendix.
CHAPTER 2

BACKGROUND

2.1. Clustering Algorithms

Clustering algorithms can be categorized as spanning tree based algorithms and dominating set based algorithms. An undirected graph is defined as $G = (V, E)$, where $V$ is a finite nonempty set and $E \subseteq V \times V$. The $V$ is a set of nodes $v$ and the $E$ is a set of edges $e$. A graph $G_S = (V_S, E_S)$ is a spanning subgraph of $G = (V, E)$ if $V_S = V$. A spanning tree of a graph is an undirected connected acyclic spanning subgraph. Intuitively, a minimum spanning tree (MST) for a graph is a subgraph that has the minimum number of edges for maintaining connectivity "(Grimaldi, 1997)".

A dominating set is a subset $S$ of a graph $G$ such that every vertex in $G$ is either in $S$ or adjacent to a vertex in $S$" (West, 2001)". Dominating sets are widely used in clustering networks" (Chen and Liestman, 2002)". Dominating sets can be classified into three main categories, Independent Dominating Sets (IDS), Weakly Connected Dominating Sets (WCDS) and Connected Dominating Sets (CDS)" (Haynes et al., 1978)".

- **Independent Dominating Sets**: IDS is a dominating set $S$ of a graph $G$ in which there are no adjacent vertices. Fig. 2.1a shows a sample independent dominating set where black nodes show cluster heads.

- **Weakly Connected Dominating Sets (WCDS)**: A weakly induced subgraph $(S)_w$ is a subset $S$ of a graph $G$ that contains the vertices of $S$, their neighbors and all edges of the original graph $G$ with at least one endpoint in $S$. A subset $S$ is a weakly-connected dominating set, if $S$ is dominating and $(S)_w$ is connected " (Chen et al., 2004)". Black nodes in Fig. 2.1b show a WCDS example.

- **Connected Dominating Sets**: A connected dominating set (CDS) is a subset $S$ of a graph $G$ such that $S$ forms a dominating set and $S$ is connected. Fig. 2.1c shows a sample CDS. CDSs have many advantages in network applications such as ease of broadcasting and constructing virtual backbones " (Stojmenovic et al., 2002)".
2.1.1. Spanning Tree Based Algorithms

Gallagher, Humblet and Spira [Gallagher et al. 1983] proposed a distributed algorithm which determines a minimum-weight spanning tree for an undirected graph that has distinct finite weights for every edge. Aim of the algorithm is to combine small fragments into larger fragments with outgoing edges. A fragment of an MST is a subtree of the MST. An outgoing edge is an edge of a fragment if there is a node connected to the edge in the fragment and one node connected that is not in the fragment. Combination rules of fragments are related with levels. A fragment with a single node has the level $L = 0$. Suppose two fragments $F$ at level $L$ and $F'$ at level $L'$;

- If $L < L'$, then fragment $F$ is immediately absorbed as part of fragment $F$. The expanded fragment is at level $L'$.

- Else if $L = L'$ and fragments $F$ and $F'$ have the same minimum-weight outgoing edge, then the fragments combine immediately into a new fragment at level $L + 1$

- Else fragment $F$ waits until fragment $F'$ reaches a high enough level for combination.

Under the above rules, the combining edge is called the core of the new fragment. The two nodes adjacent to the core exchange messages on the core branch itself, allowing each of these nodes to determine both the weight of the minimum outgoing edge and the side of the core on which this edge lies. The upper bound for the number of messages exchanged during the execution of the algorithm is $5N\log_2 N + 2E$, where $N$ is the number of nodes and $E$ is the number of edges in the graph. A message contains at most one edge weight and $\log_2 8N$ bits. Worst case time for this algorithm is $O(E + N \log_2 N)$.
Awerbuch "(Awerbuch 1987)" proposed an algorithm in which each tree will hook itself on edge leading to the neighboring tree of maximum level instead of hooking itself on its minimum weight edge. The algorithm has two stages: Counting Stage and MST Stage. In this algorithm, if the tree waits for a long time, it will be rewarded properly. This is the main idea behind the Counting stage of the algorithm. The MST stage assumes knowledge of $V$, the total number of nodes, which is provided by the previous Counting stage. This has a lot of similarity with Gallagher, Humblet and Spira’s algorithm "(Gallagher et al. 1983)". The only difference is that level increases are originated by many nodes, not only by the root node. The MST stage is performed in two phases. The first phase runs an algorithm identical to Gallagher, Humblet and Spira’s algorithm"(Gallagher et al. 1983)", and terminates when all trees reach the size of $\Omega(V/\log V)$. The new algorithmic ideas are introduced in the second phase. Algorithm updates the levels in a very accurate fashion, which prevents small trees waiting for big trees and speeds up the algorithm. The algorithm requires $O(E + V \log V)$ messages and $O(V)$ time.

The algorithms proposed by Gallagher, Humblet and Spira (Gallagher et al. 1983) and Awerbuch "(Awerbuch 1987)" uses Tree—join—tree approach. Yao-Nan Lien "(Lien 1988)" proposed a distributed minimum spanning tree algorithm that uses Node—join—tree approach. The algorithm is initialized from a single node such that there is no need to wake up all nodes at the beginning as stated in Gallagher, Humblet and Spira’s algorithm "(Gallagher et al. 1983)". Starting from any node, an MST fragment($M$) grows from a single node to complete MST iteratively by drafting nodes into $M$. In each iteration, each terminal node of $M$ tries to draft more nodes into $M$ by sending a Follow—me message to each of its neighboring nodes except its preceding node. Each neighboring node decides whether or not to hook itself to $M$ as a new terminal node based on its own local information. The new terminal nodes continue the drafting process iteratively until the end of the iteration when there is no node that wants to hook to $M$. A complete MST is formed if all nodes are included in $M$. The algorithm needs at most $(2e+n(n-1)/4)$ messages in $O(n^2)$ time. In the best case, it needs only $2e$ messages in $O(n \log n)$.

Ahuja and Zhu "(Ahuja and Zhu 1989)" proposed a distributed minimum spanning tree algorithm which uses the Tree—join—tree approach as used in the Gallagher, Humblet and Spira’s Algorithm "(Gallagher et al. 1983)". The algorithm works in phases. In phase 1 of the algorithm, each node needs to do the following:
• Sets the minimum adjacent edge as *Branch* and notifies its decision to the node on the other side of the edge.

• Learns the *nodeIDs* at the other side of its adjacent edges.

• Participates in the construction of underlying spanning tree.

By cutting the number of fragments at least one half in each phase, it needs at most $O(\log n)$ phases. In the worst case, the algorithm needs at most $(2m + 2(n - 1)\log(n/2))$ messages and $(2d\log n)$ time, where $d$ is the diameter of the network. In the best case, it needs only $2m$ messages in $2d$ time. On the average, the algorithm needs only $O(m)$ messages and $O(d)$ time.

Garay et al. "(Garay et al. 1993)" provide a modified, controlled version the Gallagher, Humblet and the Spira’s algorithm "(Gallagher et al. 1983)". The algorithm is able to achieve the following:

• Upon termination, the number of the fragments is bounded above by $\frac{n}{2^{\text{number of phases}}}$.

• Throughout the execution, the diameter of every fragment $F$ satisfies $\text{Diameter}(Fragment) < 3^{\text{number of phases}}$.

The time complexity of the algorithm is $O(\text{Diam}(G) + n^{0.614})$.

Banerjee and Khuller "(Banerjee and Khuller 2000)" proposed a protocol based on a spanning tree for hierarchical routing in wireless networks "(Kleinrock and Faroukh 1997, Xu and Dai 1998)". In their scheme, a cluster is a subset of vertices whose induced graph is connected. These subsets are chosen with consideration to cluster size and the maximum number of clusters to which a node can belong. Banerjee and Khuller "(Banerjee and Khuller 2000)" defined their clustering problem in a graph theoretic framework, and present an efficient distributed solution that meets all the desirable properties. The algorithm proceeds by finding a rooted spanning tree of the graph. The algorithm creates a BFS tree and then visits each vertex in the the tree in post order. The time complexity of the algorithm is $O(|E|)$.

A topology graph for a mobile ad hoc network "(Royer and Toh 1999)" can have any arbitrary structure. Srivastava and Ghosh "(Srivastava and Ghosh 2003)" proposed a distributed algorithm for constructing a rooted spanning tree of a dynamic graph with
the root being located towards the center of the graph. They described the $\alpha$ cone as the origin concerned node and bounded by two rays with an angle $\alpha$ between them. The attribute color is given for each node to define their states. The algorithm proposed works in two stages. In the first stage, it finds a spanning forest. In the second stage the trees of the spanning forest are connected together to produce tree with a single root. The authors proposed a priority-based algorithm for the second stage.

2.1.2. Dominating Set Based Algorithms

Various algorithms exist for clustering in dominating sets (DS). This section mentions most recent, well known dominating set algorithms, categorizing them by the resulting dominating set type.

2.1.2.1. Clustering Using IDS

By using independent dominating sets, one can guarantee that there are no adjacent cluster heads in the entire graph. This minimizes the number of dummy clusters in the network.

Baker and Ephremides "(Baker and Ephremides 1981)" proposed an independent dominating set algorithm called highest vertex id. In this algorithm, each vertex scans its closed neighbor set and chooses the highest id neighbor as a cluster head. A very similar algorithm to the highest vertex id algorithm is the lowest id algorithm by Gerla and Tsai "(Gerla and Tsai 1995)" where each vertex with the lowest id within its closed neighborhood is selected as the cluster head. Gerla and Tsai developed another algorithm to find the independent dominating sets called the highest degree algorithm. In this algorithm, each vertex with the highest degree in its closed neighborhood is selected as the cluster head "(Gerla and Tsai 1995)".

Although these algorithms are considered as important algorithms, Chen et al. "(Chen et al. 2002)" proposed that these algorithms are not working correctly for some graphs. In some situations, some independent sets cannot form a dominating set. To solve this incorrect operation, Chen et al. "(Chen et al. 2002)" developed the $k$-distance independent dominating set algorithm. By this algorithm, Chen "(Chen et al. 2002)" adds one more rule to the above algorithms such that in a $k$-distance dominating set, every
cluster head must be at least \(k + 1\) distant from each other \(\text{(Ohta et al. 2003)}\).

### 2.1.2.2. Clustering Using WCDS

Although independent dominating sets are suitable for constructing optimum sized dominating sets, they have some deficiencies such as lack of direct communication between cluster heads. In order to obtain the connectivity between cluster heads, WCDSs can be used to construct clusters. The WCDS for clustering in ad hoc networks was first proposed by Chen and Liestman \(\text{(Chen and Liestman 2003)}\). In this algorithm, the graph is first partitioned into non-overlapping regions -this is done by growing a spanning forest of the graph- and at the end of this phase, the subgraph induced by each tree defines a region. Then a greedy approximation algorithm is executed to find a small WCDS of each region. The greedy algorithm is based on Guha and Khuller’s second algorithm \(\text{(Guha and Khuller 1998)}\). Once small WCDSs are constructed, the union of these WCDSs constructs the dominating set of the entire graph. Some additional vertices from region borders can be added to the dominating set to ensure that the final dominating set of \(G\) is weakly-connected. This type of clustering is called zonal clustering.

### 2.1.2.3. Clustering Using CDS

CDSs have many advantages in network applications such as ease of broadcasting and constructing virtual backbones \(\text{(Stojmenovic et al. 2002)}\), however undesirable number of clusterheads can be obtained. So, in constructing connected dominating sets, primary problem is the minimum connected dominating set decision problem.

Guha and Khuller \(\text{(Guha and Khuller 1998)}\) proposed two centralized greedy algorithms for finding suboptimal connected dominating sets. In the first algorithm, initially all vertices are white colored. In the first step, the algorithm selects the node with the maximum number of white neighbors as a dominating node. The dominating node becomes black, and its neighbors become gray. Then the algorithm iteratively scans the gray nodes and their white neighbors. In each iteration, the gray node or the pair of nodes with the maximum number of white neighbors is selected as a cluster node. This iteration process continues until no white vertex left in the graph. In the second algorithm, white vertex with the maximum number of white neighbors is selected as a
dominating node. This iteration lasts until no white colored vertex left in the graph. When the iteration ends, the algorithm re-colors some gray nodes to black so that the dominating set becomes connected.

Wu and Li ”(Wu and Li 2002, 1999)”, improved Das and Bharghavan’s ”(Das and Bharghavan 1997; Das et al. 1997)” distributed algorithm as a localized distributed algorithm for finding connected distributed sets in which each node only needs to know its distance-two neighbor ”(Chen et al. 2004)”. In Wu and Li’s algorithm ”(Wu and Li 2002)”, initially each vertex marks itself as $F$ indicating that it is not dominated yet. In the first phase, a vertex marks itself as $T$ if any two of its neighbors are not connected to each other directly. In the second phase, a $T$ marked vertex v changes its mark to $F$ if either of the following conditions is met:

1. $\exists u \in N(v)$ which is marked $T$ such that $N[v] \subseteq N[u]$ and $id(v) < id(u)$;
2. $\exists u, w \in N(v)$ which is marked $T$ such that $N(v) \subseteq N(u) \cup N(w)$ and $id(v) = \min\{id(v), id(u), id(w)\}$;

Dai and Wu ”(Dai and Wu 2004; Wu 2002)” proposed an extended localized algorithm for finding CDS. The algorithm is based on Wu and Li ”(Wu and Li 2002)” algorithm with improved pruning rules. Dominant pruning rules with more than two connector hosts were not considered in early studies due to the following two assumptions: 1) testing the coverage of multiple hosts could be costly and 2) only a few hosts neighbor sets need to be covered by three or more other hosts. However, Dai and Wu ”(Dai and Wu 2004)” showed that these assumptions are not always true. They proposed a generalized dominant pruning rule, cluster heads, where $k$ can be any number. According to this algorithm, first phase works same as Wu and Li’s algorithm ”(Wu and Li 2002)”, but in phase two, instead of rule 1 and rule 2, Rule $k$ pruning rule is applied to eliminate dummy cluster heads. According to Rule $k$, if neighbors of a cluster head is dominated by more than two directly connected cluster heads, it can be eliminated. With this work, Dai and Wu ”(Dai and Wu 2004)” showed that Rule $k$ can be implemented with local neighborhood information that has the same complexity as Rule 1 and, less complexity than Rule 2. Cokuslu et al. modify this algorithm to get smaller set of clusterheads. They provide significant modifications by considering the degrees of the nodes during marking process and also provide further heuristics to determine the color of a node in the initial
Xinfang Yan et al. "(Yan et al. 2003)" proposed a heuristic algorithm for minimum connected dominating set. The algorithm first calculates a weight for each node indicating node’s uptime and its amount of power left. It then uses weight parameter and some rules from Wu and Li’s algorithm "(Wu and Li 2002)" in selecting the cluster heads. By using this heuristic, Yan et al. "(Yan et al. 2003)" make a better estimation on the stability of the backbone topology.

Peng-Jun Wan et al. "(Wan et al. 2004)" proposed a distributed algorithm for finding a CDS. This algorithm consists of two phases. The first phase constructs a maximal independent set (MIS) using a rooted spanning tree which is constructed at the beginning of the phase. The second phase constructs a dominating tree from the MIS, whose internal nodes would become a CDS. The algorithm uses $O(n)$ messages and takes $O(n)$ time.

Hui Liu et al. "(Liu et al. 2004)", improved Wu and Li’s "(Wu and Li 2002)" algorithm by adding a third phase elimination. In the additional third phase, the algorithm searches redundant cluster heads. A cluster head is eliminated if it is dominated by two of its cluster head neighbors. The distributed algorithm has time complexity $O(n^2)$ and message complexity $O(n)$.

### 2.2. Backbone Formation Algorithms

Dominating set based algorithms which were introduced in section 2.1.2 construct a backbone architecture. The advantages and disadvantages of ICDS, WCDS and CDS schemes were also discussed in section 2.1.2. In this section we will introduce other algorithms related to backbone formation. Power-saving, minimal routing and the topology control is the goal of these algorithms which are mainly focused on one issue according to the different needs.

Rubin et. al "(Rubin et al. 2002)" classifies the nodes as high capacity and low capacity nodes and unmanned vehicles according to their power status. High capacity nodes include Backbone Nodes ($BNs$) and Backbone Capable Nodes ($BCNs$). They present a topological synthesis algorithm that selects a subset of high capacity nodes to form a backbone network. Each backbone node manages the allocation of resources for transport of messages from/to itself and among regular nodes ($RN$) that reside in its managed cluster.
of nodes. Backbone nodes also interact to coordinate the allocation of MAC layer communications assets such as time slots in their access nets to prevent interferences. They introduce the TBONE protocol to implement the key networking schemes for such a Mobile Backbone Network (MBN). TBONE protocol consists of three algorithms: the Anet association algorithm, the BN election algorithm, and the time slot allocation algorithm. The Anet association algorithm provides a mechanism that associates an unassociated low power node with exactly one BN. Every unassociated low power node instigates the Anet association algorithm by sending join_request message to a BN or associated BCN.

The purpose of the BN election algorithm is to elect eligible BCNs and convert them into BNs in order to satisfy the covering requirement. The dynamic weighted labels of BCNs determine their eligibility. The unassociated BCN that initiates the BN election algorithm broadcasts its ID and dynamic weighted label request to all power link associated BCN neighbors. All BCNs collect data of others. The one with maximum dynamic label will convert itself to a BN. The purpose of the BN-BCN conversion algorithm is to provide a mechanism to determine redundant BNs and convert them into BCNs in order to support minimality. If a BN determines that each of the nodes in its Anet has at least one low power link BN neighbor and all its high power link BN neighbors would remain in the same component, it converts to BCN. The time slot algorithm provides a mechanism for allocation of time slots by BNs among their associated low power nodes. The time and message complexity is not given in the article.

Ya-Feng et al. [2004] focused on the construction of the optimal Virtual Multicast Backbone (VMB) with the fewest forwarding nodes to decrease overhead and cost, due to the scarce resource in ad hoc networks. Instead of conventional Steiner tree model, the optimal shared VMB in ad hoc networks is modeled as Minimum Steiner Dominating Set (MSCDS) in Unit-Disk Graphs (UDG), which is NP-hard. One-hop algorithm and d-hop algorithm is proposed for approximating MSDCS. One-hop algorithm is divided into steps below:

1. Find a maximal independent set I in G(V)

2. In G, apply the Steiner tree algorithm in [1998] to find a Steiner tree T for the subset I, with all edges having unit weight. The final solution is the set of the nodes of T.
The *One-hop* Algorithm constructs a hierarchical VMB. However, when deployed in sparse UDG, where most multicast nodes are two or more hops apart from each other, it mostly results in trivial single-node multicast clusters and consequently flat VMB. This implies that *One-hop* Algorithm is not fit for VMB construction in sparse ad hoc networks. To address this issue, an extended *d-hop* Algorithm is proposed with detailed description of distributed implementation, whose approximation ratio proves also constant. The *d-hop* Algorithm finds a *d-MIS* among multicast nodes, which is also a *d-hop* dominating set of the multicast group, and then each node in the *d-MIS* becomes a cluster-head and forms a *d-hop* cluster with all its d-neighbors. Intra-cluster, some multicast nodes are further chosen to dominate multicast nodes of the cluster. These nodes are connected to the cluster-head with the shortest paths. Inter-cluster, a Steiner tree is used to connect all cluster-heads. The distributed implementation of *d-hop* Algorithm for constructing an SCDS of multicast nodes has a time complexity $O(Dn)$ where $D$ is the graph diameter and a message complexity of $O(n\log(n))$ if $d$ equals to 1, otherwise $O(n^d)$.

Haitao and Gupta *(Haitao and Gupta 2004)* proposed the Selective Backbone Construction Algorithm (SBC) for energy efficiency in MANETs. SBC constructs backbone in two steps. In the first step, one or more backbone seed nodes are elected. Next they choose their neighbor nodes into backbone to connect the whole network. When SBC starts, every node computes its priority and broadcasts it in its neighborhood. It also broadcasts the identities of its direct neighbors that it has discovered. Thus each node gets to know the topology information in its two-hop neighborhood. Backbone seeds are also elected based on two-hop neighborhood information. When electing backbone seeds, they consider two factors. An ideal backbone seed should have high priority. In addition, to speed up the process of backbone construction, it is desirable to have backbone seed nodes chosen from an area of high node density so that more nodes can be covered quickly. They use node degrees as the indicator of node density. Considering these requirements, every node first compares its degree with the degrees of its neighbors based on the two-hop topology information. If its degree is the highest, it picks the neighbor with highest priority as backbone seed. Otherwise, it depends on nodes in other neighborhoods to pick backbone seeds. Time and message complexities is not given in the article.

In Min et. al’s scheme (RVBSM) *(Min et al. 2005)*, they assume that every node records its own location at every second during the period. And whenever a node
selects a node from its neighbors, it chooses the one with the highest rank. Every message contains color, rank and locations of both the sender and the 1 hop backbone neighbors of the sender. Initially every node has white color and changes its color during the procedure. They define the ranking to be an ordering of \((\text{stability}, \text{coverage}, \text{id})\) of nodes and they claim that a node \(v\) with rank \((s_v, c_v, id_v)\) has a higher order than a node \(u\) with rank \((s_u, c_u, id_u)\) if:

1. \(s_v > s_u\)
2. \(s_v = s_u\) and \(c_v > c_u\) or
3. \(s_v = s_u\) and \(c_v > c_u\) and \(id_v > id_u\)

The stability and the coverage of each node can be estimated. Every message contains color, rank and locations of both the sender and the 1 hop backbone neighbors of the sender. The algorithm has message complexity of \(O(Dn)\) and time complexity of \(O(n)\), where \(D\) is the maximum degree.

Ju and Rubin \(\text{"Huejiun and Rubin 2005"}\) proposed the Enhanced Backbone Synthesis (EBS) in which every node has two timers: \textit{Short Timer} and \textit{Long Timer}. There is no time synchronization between nodes; every node maintains its own time. Whenever the \textit{Short Timer} expires at a node, the node broadcasts a \textit{Hello} message to its direct neighbors. The \textit{Hello} message contains the nodes \(\text{ID}\), status, weight, associated \(\text{BN ID}\), \(\text{BN-to-BCN}\) indicator, and its \(\text{BN}\) neighbor list. The weight of a node can be based on its \(\text{ID}\), degree, capability, congestion level, or on some stability measure. Through periodic \textit{Hello} message exchange, each node learns its 1-hop neighborhood and 2-hop \(\text{BN}\) neighborhood. A node does not learn its complete 2-hop neighborhood, as assumed by typical CDS construction algorithms. Whenever the \textit{Long Timer} expires at a node, the node updates its neighbor list based on the number of \textit{Hello} messages received within the previous period. In accordance with its type, this node then executes the following operations: A \(\text{BCN}\) runs the association and the \(\text{BCN-to-BN}\) conversion algorithms; a \(\text{BN}\) runs the \(\text{BN-to-BCN}\) conversion algorithm. The message complexity of the \(\text{MBN}\) topology synthesis algorithm is order of the \(O(1)\) message per node. The enhanced \(\text{MBN}\) topology synthesis algorithm converges in \(O(1)\) time per node.
2.3. Distributed Mutual Exclusion Algorithms

2.3.1. Performance Metrics

Performance of a distributed mutual exclusion algorithm depends on whether the system is lightly or heavily loaded. If no other process is in the critical section when a process makes a request to enter it, the system is lightly loaded. Otherwise, when there is a high demand for the critical section which results in queueing up of the requests, the system is said to be heavily loaded. The important metrics to evaluate the performance of a mutual exclusion algorithm are the number of messages per request, response time and the synchronization delay as described below:

- **Number of Messages per Request** ($M$): The total number of messages required to enter a critical section is an important and useful parameter to determine the required network bandwidth for that particular algorithm. $M$ can be specified for high load or light load in the system as $M_{\text{heavy}}$ and $M_{\text{light}}$.

- **Response Time** ($R$): The Response Time $R$ is measured as the interval between the request of a node to enter critical section and the time it finishes executing the critical section. When the system is lightly loaded, two message transfer times and the execution time of the critical section success resulting in $R_{\text{light}} = 2T + E$ units. Under heavy load conditions, assuming at least one message is needed to transfer the access right from one node to another, $R_{\text{heavy}} = w(T + E)$ where $w$ is the number of waiting requests.

- **Synchronization Delay** ($S$): The synchronization delay $S$ is the time required for a node to enter a critical section after another node finishes executing it. The minimum value of $S$ is one message transfer time $T$ since one message success to transfer the access rights to another node. The lower bounds for $M$, $R$ and $S$ are shown in Table 2.1.

2.3.2. Ricart-Agrawala Algorithm

The Ricart-Agrawala Algorithm (RA) represents a class of decentralized, permission based mutual exclusion algorithms. In RA Algorithm, when a node wants to enter
Table 2.1. Lower Bounds for Performance Metrics

<table>
<thead>
<tr>
<th>$M_{light}$</th>
<th>$M_{heavy}$</th>
<th>$R_{light}$</th>
<th>$R_{heavy}$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>$2T + E$</td>
<td>$w(T + E)$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Table 2.2. Performance Metrics of Ricart-Agrawala Algorithm

<table>
<thead>
<tr>
<th>$M_{light}$</th>
<th>$M_{heavy}$</th>
<th>$R_{light}$</th>
<th>$R_{heavy}$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2(N - 1)$</td>
<td>$2(N - 1)$</td>
<td>$2T + E$</td>
<td>$w(T + E)$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

In a critical section, it sends a timestamped broadcast Request message to all of its peers in that critical section request set. When a node receives a Request message, it returns a Reply message if it is not in the critical section or requesting it. If the receiving node is in the critical section, it does not reply and queues the request. However, if the receiver has already made a request, it compares the timestamp of its request with the incoming one and replies the sender if the incoming request has a lower timestamp. Otherwise, it queues the request and enters the critical section. When a node leaves its critical section, it sends a reply to all the deferred requests on its queue which means the process with the next earliest request will now receive its last reply message and enter the critical section. The total number of messages per critical section is $2(N - 1)$ as $(N - 1)$ requests and $(N - 1)$ replies are needed. One of the problems with this algorithm is that if a process crashes, it fails to reply which is interpreted as a denial of permission to enter the critical section, so all other processes that want to enter are blocked. Also, the system should provide some method of clock synchronization between processes. The performance metrics for the RA Algorithm are shown in Table 2.2. When a node finishes execution of a critical section, one message is adequate for a waiting node to enter, resulting in $S = T$.

2.3.3. Token-Based Algorithms

The general Token Passing (TP) Algorithm for mutual exclusion is characterized by the existence of a single token where the possession of it denotes permission to enter
a critical section. The token circulation can be performed in a logical ring structure or by broadcasting "(Suzuki and Kasami 1985)". In a ring based TP Algorithm, any process that requires its critical section will block the token and issue it when it finishes executing. Fairness is ensured in this algorithm as each process waits at most $N - 1$ entries to enter the critical section. There is no starvation since passing is in strict order. The main difficulties with TP Algorithm are as follows. There would be the idle case of no processes entering CS which would incur overhead of constantly passing the token. There could be lost tokens which would require diagnosis and creating a new token by a central node or distributed control is needed and to prevent duplicate tokens, central coordinator should ensure generation of only one token. Crashes should also be dealt with as these would require detection of the dead destinations in the form of acknowledgements. One important design issue with TP Algorithm is the determination of the holding time for unneeded token. If this time is too short, there will be high overhead. However, keeping this time too long would result in high CS latency. The performance metrics for a general Token-Based Algorithm is shown in Table 2.3 "(Erciyes 2004, 2005)".

2.3.4. Mutual Exclusion Algorithms on MANET

Distributed mutual exclusion in mobile networks is a relatively new research area. Singhal et al. "(Singhal and Manivannan 1997)" proposed a concept of look-ahead technique for distributed mutual exclusion which instead of enforcing mutual exclusion among all the sites of a mobile system, enforces mutual exclusion only among the sites which are concurrently competing for critical section (CS), resulting in less message overhead. Mutual exclusion algorithm involves two issues: First is identifying sites which are concurrently competing for CS, and second enforcing mutual exclusion among these sites. Once a site knows all the sites which are concurrently requesting CS, it can use Ricart-Agrawala method on those sites to enforce mutual exclusion. Walter et al. "(Walter 18)"
et al. (2001b)" proposed a mobility aware token based distributed mutual exclusion algorithm which combines ideas from several papers. The partial reversal technique from "(Gafni and Bertsekas 1981)" used to maintain a destination oriented directed acyclic graph (DAG) in a packet radio network when the destination is static, is used in the algorithm to maintain a token oriented DAG with a dynamic destination. Like the algorithms of "(Chang et al. 1990, Dhamdhere and Kulkarni 1994, Raymond 1989)" each node in the algorithm maintains a request queue containing the identifiers of neighboring nodes from which it has received requests for the token. Like Dhamdhere and Kulkarni’s algorithm "(Dhamdhere and Kulkarni 1994)", the algorithm totally orders nodes. The lowest node is always the current token holder, making it a sink toward which all requests are sent. Each node dynamically chooses its lowest neighbor as its preferred link to the token holder. Nodes sense link changes to immediate neighbors and reroute requests based on the status of the previous preferred link to the token holder and the current contents of the local request queue. All requests reaching the token holder are treated symmetrically, so that requests are continually serviced while the DAG is being re-oriented and blocked requests are being rerouted.

Baldoni "(Baldoni et al. 2002)" et al. proposed a token based distributed mutual exclusion algorithm suited for mobile ad-hoc networks. The algorithm is based on a dynamic logical ring and combines the two families of token based algorithms (i.e., token asking and circulating token) in order to get a optimal number of messages exchanged per CS access under heavy request load. The algorithm aims at maintaining device power consumption as low as possible by reducing the number of hops traversed per CS execution and by not sending any control message when no processes request the CS. Mobility is addressed by exploiting the information of the routing table in order to send each message to the closest node in terms of number of hops.

The $h$-out of $k$ mutual exclusion problem is also known as the $h$-out of $k$ resource allocation problem. It concerns with how to control nodes in a distributed system so that each node can access $h$ resources out of totally $k$ shared resources, $l \leq h \leq k$, with the constraint that no more than $k$ resources can be accessed concurrently. Jiang "(Jiang 2003)" proposed a prioritized distributed $h$-out of $k$ mutual exclusion algorithm for MANETs with real-time or prioritized applications. The proposed algorithm is sensitive to link forming and link breaking and thus is suitable for MANETs. The proposed
algorithm claims the highest priority first serve property for real-time applications. For non-real-time applications, one may associate the priority with the number of requested resources to achieve the maximum degree of concurrency. Yang \cite{Yang2005} proposed a distributed algorithm to solve the mutual exclusion problem in MANETs. The proposed algorithm improves the CS execution time by allowing at most R tokens to be concurrently dispatched, it employs logical ring construction to adapt the token navigation to the system requirements and it is designed with the consideration of the dynamical link formation characteristics in MANETs and is thus suitable for mobile environments. Wu et al. \cite{Wu2005} proposed a permission-based MUTEX algorithm for MANETs. In order to reduce the message cost, the algorithm uses the "look-ahead" technique as in \cite{Singhal1997}, which enforces MUTEX only among the hosts currently competing for the critical section (CS). The constraint of FIFO channel is also relaxed. The proposed mechanism handles the "doze" mode and "disconnection" of mobile hosts. Using timeout, a fault tolerance mechanism is introduced to tolerate transient link and host failures.
CHAPTER 3
MERGING CLUSTERING ALGORITHM

3.1. General Idea and Description of the Algorithm

Merging Clustering Algorithm (MCA) \(^{(\text{Dagdeviren et al., } 2005, 2006)}\) finds clusters in a MANET by merging the clusters to form higher level clusters as mentioned in Gallagher, Humblet, Spira’s algorithm \(^{(\text{Gallagher et al., } 1983)}\). However, we focus on the clustering operation by discarding minimum spanning tree. This reduces the message complexity as explained in Section 3.3. The second contribution is to use upper and lower bound parameters for clustering operation which results in balanced number of nodes in the clusters formed. The lower bound is limited by a parameter which is defined by \(K\) and the upper bound is limited with \(2K\). The last contribution is the clusterhead(leader) selection method as an alternative to the core of the fragment in \(^{(\text{Gallagher et al., } 1983)}\).

We assume that each node has distinct \textit{node.id}. Moreover, each node knows its \textit{cluster.leader.id}, \textit{cluster.id} and \textit{cluster.level}. \textit{Cluster.level} is identified by the number of the nodes in a cluster. Leader node is the node with maximum \textit{node.id}. \textit{Cluster.leader.id} is equal to the \textit{cluster.id}. The local algorithm consists of sending messages over adjoining links, waiting for incoming messages and processing messages. Successful packet transfers are detected by receiving the \textit{ACK} message from MAC layer of IEEE 802.11. The finite state machine of the algorithm is shown in Fig. 3.1.

The algorithm requires the sequence of messages as in Fig. 3.2. Firstly a node sends a \textit{Poll.Node} message to a destination node. Destination node sends a \textit{Node.Info} message back to originator node. Originator node then sends a \textit{Connect.Ldr} or \textit{Connect.Mbr} message to destination node to state it is the current leader or not. Destination node sends a \textit{Ldr.ACK} or \textit{Mbr.ACK} message to originator node. We assume that the underlying network provides broadcast communication. After the above message exchange, the new leader node multicasts a \textit{Change.Cluster} message to new cluster nodes. New cluster members are replied with \textit{Change.Cluster.ACK} messages. Messages can be transmitted independently in both directions on an edge and arrive after an unpredictable but finite delay, without error and in sequence. Message types are
Figure 3.1. Finite State Machine of the Merging Clustering Algorithm

Poll_Node, Ldr_Poll_Node, Node_Info, Ldr_ACK, Mbr_ACK, Connect_Mbr, Connect_Ldr, Change_Cluster and Change_Cluster_ACK as described below.

- **Poll_Node**: A cluster leader node will send Poll_Node (node_id, cluster_level) message to a destination node to begin the clustering operation. A node in IDLE state changes its state to IDLE_WT_CONN, a node in LEADER state changes its state to LDR_WT_CONN after receiving Poll_Node message.

- **Ldr_Poll_Node**: A cluster member node will send Ldr_Poll_Node (node_id, cluster_level) message to cluster leader node if cluster member node receives a Poll_Node (node_id, cluster_level) message from a node which is not in the same cluster. A node in LEADER state changes its state to LDR_WT_CONN state after receiving Ldr_Poll_Node message.

- **Node_Info**: A cluster leader node will send Node_Info (node_id, cluster_level) message if it receives a Poll_Node (node_id, cluster_level) or Ldr_Poll_Node (node_id, cluster_level) message. A node in WT_INFO state changes its state to WT_ACK
Figure 3.2. Message Flow Diagram of Merging Clustering Algorithm

state after receiving Node_Info message.

- \textit{Connect}_Mbr: A cluster node will send \textit{Connect}_Mbr (node id) message after it receives a Node_Info (node id, cluster level) which has a smaller node id than sender. A node either in IDLE\_WT\_CONN or LDR\_WT\_CONN state changes its state to MEMBER state after receiving Connect_Mbr message.

- \textit{Connect}_Ldr: A cluster node will send \textit{Connect}_Ldr (node id) message after it receives a Node_Info (node id, cluster_level) message which has a greater node id than sender’s node id. A node either in IDLE\_WT\_CONN or LDR\_WT\_CONN state changes its state to LDR\_WT\_ACK state after receiving Connect_Mbr message.

- Ldr_ACK: A node will send Ldr_ACK (node id, cluster_level) message when it receives a Connect_Mbr message. A node in WT\_ACK changes its state to LDR\_WT\_ACK state after receiving Ldr_ACK message.

- Mbr_ACK: A node will send Mbr_ACK message when it receives a Connect_Ldr message. The receiver node of the Mbr_ACK message is a member of the cluster and changes its state to MEMBER state after receiving Mbr_ACK message.

- Change_Cluster: A node will multicast a Change_Cluster (node id, cluster_level) message after it receives a Ldr_ACK message. The leader of a cluster calculates new
level and multicasts Change_Cluster (node_id, cluster_level) to all cluster member
nodes to update their cluster_id and cluster_level information.

- Change_Cluster_ACK: A node will send a Change_Cluster_ACK message after it
receives Change_Cluster message. A node in LDR_WT_ACK state changes its state
to LEADER after receiving Change_Cluster_ACK messages from all new cluster
member nodes.

- Period_TOUT: This message can be regarded as an internal message. Per-
period_TOUT occurs for every node in the network to start clustering operation peri-
odically.

Every node in the network performs the same local algorithm. Each node can be either in
IDLE, WT_ACK, MEMBER, LDR_WT_ACK, LEADER, LDR_WT_CONN or WT_INFO
states described below.

- IDLE: Initially all nodes are in IDLE state. If Period_TOUT occurs, node sends a
Poll_Node message to destination node and will make a state transition to WT_INFO
state.

- WT_INFO: A node in WT_INFO state waits for Node_Info message.

- WT_ACK: A node in WT_ACK state waits for a Mbr_ACK or Ldr_ACK. If
Mbr_ACK is received, node will make a state transition to MEMBER state. If
Ldr_ACK is received, the node will multicast CHANGE_LEADER message and
make a state transition to LEADER state.

- MEMBER: A node which is a member of a cluster, is in the MEMBER state. If
a Poll_Node message is received, the node will send Ldr_Poll_Node message to the
leader node of the cluster. If a Change_Cluster message is received, the node will
update its cluster information.

- LDR_WT_ACK: A node in LDR_WT_ACK state waits for Change_Cluster_ACK
messages of all new member nodes in the new cluster.

- LEADER: When A cluster leader node is in the LEADER state, if a Poll_Node or
a Ldr_Poll_Node is received, the node will firstly check the 2K parameter to decide
on the clustering operation. If cluster level is smaller, the node will send a Node_Info message and make a state transition to LDR_WT_CONN state.

- **LDR_WT_CONN**: A node in LDR_WT_CONN state waits for Connect_Mbr or Connect_Ldr message. If Connect_Mbr is received, the node will make a state transition to MEMBER state. If Connect_Ldr is received, node will make a state transition to LDR_WT_ACK state.

- **IDLE_WT_CONN**: A node in IDLE_WT_CONN state waits for Connect_Mbr or Connect_Ldr message. If Connect_Mbr is received, the node will make a state transition to MEMBER state.

Timeouts can occur when two nodes are communicating. If a timeout occurs at a node which is not a cluster leader either in IDLE, IDLE_WT_CONN, WT_INFO or WT_ACK states, it returns back to the IDLE state, a node which is a cluster leader either in LDR_WT_CONN, WT_ACK or WT_INFO states returns back to the LEADER state, a node either in LEADER, MEMBER, LDR_WT_ACK states doesn’t change its state.

### 3.2. An Example Operation

Assume the mobile network in Fig. 3.3. $K$ parameter is given as 4. Initially all the nodes are in IDLE state. Period_TOUT occurs in Node 1, Node 2, Node 6, Node 9, Node 19, Node 8, Node 12, Node 13, Node 16. Node 1 sends a Poll_Node message to Node 5 and sets its state to WT_INFO. Node 5 receives the Poll_Node message and sends Node_Info message to Node 1. Node 5 sets its state to IDLE_WT_CONN. Node 1 receives the Node_Info message and sends a Connect_Ldr message to Node 10 since the node_id of Node 5 is greater than Node 1. Node 1 sets its state to WT_ACK. Node 5 receives the Connect_Ldr message and sends a Mbr_ACK message to Node 1. Node 1 receives the message and sets its state to MEMBER. Node 5 sends Change_Cluster message to Node 1 indicating that new cluster is formed between Node 1 and Node 5. Node 1 replies with a Change_Cluster_ACK message to Node 5. Node 2 and Node 14, Node 3 and Node 0, Node 4 and Node 16, Node 6 and Node 8, Node 7 and Node 19, Node 9 and Node 11, Node 10 and Node 12, Node 13 and Node 15 are connected same as Node 1 and Node 5.
to form clusters with level 2.

Figure 3.3. Clusters obtained using Merging Clustering Algorithm

After clusters with level 2 are formed, Node 18 in IDLE state sends a Poll_Node message to Node 15. Node 18 sets its state to WT_INFO. Node 15 in LEADER state receives Poll_Node message and checks the $2K$ parameter. Since cluster level of Node 15 is smaller than $2K$, Node 15 sends a Node_Info message to Node 18. Node 15 sets its state to LDR_WT_CONN. Node 18 in WT_INFO_STATE receives NODE_INFO message from Node 15 and sends a Connect_Mbr message to Node 15. Node 18 sets its state to WT_ACK. Node 15 receives Connect_Mbr and sends Ldr_ACK message to Node 18. Node 15 sets its state to MEMBER. Node 18 in WT_ACK state receives Ldr_ACK message and multicasts Change_Cluster message to Node 13 and Node 15 to update new cluster information. Node 18 sets its state to LDR_WT_ACK. After that, Node 15 receives Change_Cluster_ACK messages and sets its state to LEADER. The clustering operation between Node 18 and Node 15 ends by these messages. Node 17 which is in IDLE state sends a Poll_Node message to Node 0 which is in MEMBER state. Node 0 receives the Poll_Node message and sends a Ldr_Poll_Node message to its cluster leader, Node 3. The clustering operation between Node 17 and Node 3 occurs same as clustering operation between Node 18 and Node 15. At the same time Node 16 which is in LEADER state compares its cluster level with $K$ parameter. Since its cluster level is lower than $K$, it
Table 3.1. Cluster Formation for 20 nodes

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Cluster 1</th>
<th>Cluster 2</th>
<th>Cluster 3</th>
<th>Cluster 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7-19 17 0-3</td>
<td>10-12 18 15-13</td>
<td>2-14 16-4 6-8</td>
<td>1-5 9-11</td>
</tr>
<tr>
<td>2</td>
<td>7-19 17-0-3</td>
<td>10-12 18-15-13</td>
<td>2-14-16-4 6-8</td>
<td>1-5-9-11</td>
</tr>
<tr>
<td>3</td>
<td>7-19 17-0-3</td>
<td>10-12-18-15-13</td>
<td>2-14-16-4 6-8</td>
<td>1-5-9-11</td>
</tr>
</tbody>
</table>

sends a \textit{Poll\_Node} message to Node 14. Node 14 and Node 16 connect to form a new cluster level with 4, same as explained previously. When the cluster levels of each cluster reach $K$, the cluster formation is ended. Lastly the clusters in Fig. 3.3 are summarized in Tab. 3.1.

Figure 3.4. MANET with 40 nodes located on a surface area of 600m $\times$ 600m
Figure 3.5. Clusters of MANET with 40 nodes

A real world example is illustrated in Fig. 3.4. A MANET with 40 nodes is located on a surface area of 600m × 600m. X, Y coordinates are written near to each node and a coverage area of a mobile node is shown in dotted circle. Node 37’s coverage area is demonstrated with dashed circle in Fig. 3.5. Merging Clustering Algorithm with $K=7$ is implemented on this architecture. Obtained Clusters are shown in Fig. 3.5. Iterations shown in Fig. 3.5 are summarized in tables 3.2 and 3.3.
Table 3.2. Cluster Formation for 40 nodes

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Cluster1</th>
<th>Cluster2</th>
<th>Cluster3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17 14 26 19 21 31 30</td>
<td>6 29 13 18 12 23 16 27 28</td>
<td>1 24 7 8 37 22 25 15 20</td>
</tr>
</tbody>
</table>

Table 3.3. Cluster Formation for 40 nodes, continued..

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Cluster4</th>
<th>Cluster5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32 3 10 2 36 9 11</td>
<td>39 14 35 38 33 0 34 5</td>
</tr>
<tr>
<td>2</td>
<td>32-3 10 2-36 9-11</td>
<td>39-14 35 38-33 0-34 5</td>
</tr>
<tr>
<td>3</td>
<td>32-3-10 2-36-9-11</td>
<td>39-14-35 38-33 0-34-5</td>
</tr>
<tr>
<td>4</td>
<td>32-3-10-2-36-9-11</td>
<td>39-14-35-38-33 0-34-5</td>
</tr>
<tr>
<td>5</td>
<td>32-3-10-2-36-9-11</td>
<td>39-14-35-38-33-0-34-5</td>
</tr>
</tbody>
</table>

3.3. Analysis

Time and message complexity is given in this section. Also a comparison of Gallagher et. al’s Algorithm (DAMWST) and MCA can be seen in table 3.4.

Theorem 3.3..1. Time complexity of the clustering algorithm has a lower bound of $\Omega(\log n)$ and an upperbound of $O(n)$.

Proof. Assume that we have $n$ nodes in the mobile network. Best case occurs when each node can merge with each other exactly, to double member count at each iteration such that Level 1 clusters are connected to form Level 2 clusters. Level 2 Clusters are connected to form Level 4 Clusters and so on. The clustering operation continues until the to Cluster Level becomes $n$. The lower bound is $\Omega(\log N)$. Worst case occurs when a cluster is connected to a Level 1 cluster at each iteration. Level 1 cluster is connected to
Table 3.4. Comparison of DAMWST and MCA

<table>
<thead>
<tr>
<th></th>
<th>DAMWST</th>
<th>MCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Complexity</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Messaging Complexity</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Advantages</td>
<td>MST Construction</td>
<td>Balanced Clustering</td>
</tr>
</tbody>
</table>

a Level 1 cluster to form a Level 2 cluster, Level 2 cluster is connected to a Level 1 cluster to form a Level 3 cluster and so on. The clustering operation continues until the Cluster Level becomes $n$. The upper bound is therefore $O(n)$. 

**Theorem 3.3.2.** Message complexity of the clustering algorithm is $O(n)$.

*Proof.* Assume that we have $n$ nodes in our network. For every merge operations of two clusters, 4 messages ($Poll_{Node}$, $Node_{Info}$, $Connect_{Ldr}/Connect_{Mbr}$, $Leader_{ACK}/Member_{ACK}$) and maximum $2K-1$ $Change_{Cluster}$ and $2K-1$ $Change_{Cluster}_ACK$ are required. For each clustering operation, $4K + 2$ messages are required. The maximum cluster size is $3K - 1$, hence number of the clusters is $n/(3K - 1)$. Total number of messages in this case is $(4K+2)*(n/(3K-1))$ which means that the message complexity has an upper bound of $O(n)$.

**3.4. Results**

The merging clustering algorithm is implemented with $ns2$ simulator. Total number of nodes vary from 10 to 100 nodes. Different size of flat surfaces are chosen for each simulation to create very small, small and medium distances between nodes, as well as, high dense, dense and medium connected topologies. Surface areas vary from $310m \times 310m$ to $400m \times 400m$, $410m \times 410m$ to $500m \times 500m$, $515m \times 515m$ to $650m \times 650m$ respectively. Random movements are generated for each simulation. Low, medium and high mobility scenarios are generated and respective node speeds are limited from 1.0m/s to 5.0m/s, 5.0m/s to 10.0m/s, 10.0m/s to 20.0m/s. $K$ is changed to obtain different size of clusters.
Fig. 3.6 displays the run-time results of the merging clustering algorithm ranging from 10 to 100 nodes. Run-time values are increased linearly when the total number of MANETs from is increased 10 to 100 nodes. Linear increase of the total message number can be seen in Fig. 3.7.

![Figure 3.6. Run-time of Merging Clustering Algorithm](image)

Cluster Quality can be measured by the coefficient of variation. In probability theory and statistics, the coefficient of variation is a measure of dispersion of a probability distribution. It is defined as the ratio of the standard deviation to the mean. For a MANET with 40 nodes, $K$ is chosen from 3 to 7 to measure the cluster quality as shown in Fig. 3.8. Coefficient of variation values from $K=3$ to $K=7$ are respectively, 0.53, 0.24, 0.13, 0.64 and 0.63. When $K$ is equal to 5, maximum balanced clusters are obtained with a coefficient variation of 0.13. Also we can state that total number of clusters are decreased when $K$ is increased as shown in from Fig. 3.8.

![Figure 3.7. Total Message Numbers of Merging Clustering Algorithm](image)
Cluster Quality against $K$ which is chosen from 5 to 9 for a MANET with 60 nodes is shown in Fig. 3.9. Coefficient of variation values from $K=5$ to $K=9$ are respectively, 0.36, 0.34, 0.32, 0.62, 0.46. Maximum balanced clustering is gained at $K=7$.

Figure 3.8. Cluster Quality of MCA Against $K$ for 40 nodes

Figure 3.9. Cluster Quality of MCA Against $K$ for 60 nodes

In Fig. 3.10, $K$ is fixed to 5 for a MANET with 40 nodes and cluster quality with respect to mobility parameter is measured by selecting from low to high mobility. Coefficient of variations from low to high mobility parameter is respectively, 0.25, 0.37 and 0.18. The MANET with low mobility results in more balanced clusters than MANET with high mobility due to rapid change of network topology.

Total Surface area of the MANET is an important parameter which effects the distance between nodes and connectivity. In Fig. 3.11, cluster quality with respect to total surface area is shown. Size of surface areas are $340\text{m} \times 340\text{m}$, $440\text{m} \times 440\text{m}$ and $560\text{m} \times 560\text{m}$. Coefficient of variations are respectively, 0.15, 0.25 and 0.28. As we increase the
surface area from $340m \times 340m$ to $560m \times 560m$, clusters become more balanced. Also, cluster node counts are decreased and total number of clusters in MANET are increased.

Different size of networks are constructed to measure general cluster quality. Merging clustering algorithm is simulated on MANETs, beginning from 10 nodes and increasing the network size by 10, up to 100 nodes. From 10 nodes to 100 nodes, $K$ is selected respectively, 2, 3, 4, 5, 7, 7, 10, 11, 12. In Fig. 3.12 general cluster quality is shown.

Total and Average edge-cut values are recorded as shown in Fig. 3.13 and Fig. 3.14. The connectivity between the nodes are squared when we double the total number of nodes in MANET. Total edge-cut values have a polynomial increase when we consider this fact. Average edge-cut values are stable against the size of the network which can be seen in Fig. 3.14.

$K$ parameter of the MCA changes the cluster node count and total number of clusters in MANET. Total and Average edge-cut values for 40 and 60 nodes are measured against $K$ as shown in Fig. 3.15 and Fig. 3.16. Total edge-cut values are decreasing linearly with $K$ because the total number of clusters in MANET is increasing. Average edge-cut values are fluctuating between 150 and 155m with $K$, resulting in approximate values since the distance between nodes does not change.

A MANET consisting of highly mobile nodes is open for link failures. Total and average edge-cut values are measured against mobility as shown in Fig. 3.17 and Fig. 3.18. Total edge-cut values are increasing linearly with mobility parameter but average edge-cut values are again fluctuating in narrow band of values.

Lastly, total and average edge-cut values against surface area is measured for 40 and 60 nodes in small, medium and large surface areas as shown in Fig. 3.19 and Fig. 3.20. As expected both values are increasing when the total surface area of the MANET is increased.

Consequently, our results conform with the analysis that run-time values and message counts grow linearly. Balanced clustering is best achieved in low mobile scenarios with medium surface area. Also algorithm is stable under different mobility and surface area conditions. $K$ heuristic changes the cluster sizes and its selection is very important. Total edge cut values show a polynomial increase when number of nodes are increased linearly, on the other hand average edge cut values are stable.
Figure 3.10. Cluster Quality of MCA Against Mobility for 40 nodes

Figure 3.11. Cluster Quality of MCA Against Surface Area for 40 nodes

Figure 3.12. Cluster Quality of MCA Against Total Number of Nodes
Figure 3.13. Total Edge-cut for Merging Clustering Algorithm

Figure 3.14. Average Edge-cut for Merging Clustering Algorithm

Figure 3.15. Total Edge-cut Against K for Merging Clustering Algorithm
Figure 3.16. Average Edge-cut Against K for Merging Clustering Algorithm

Figure 3.17. Total Edge-cut for MCA Against Mobility

Figure 3.18. Average Edge-cut for MCA Against Mobility
Figure 3.19. Total Edge-cut for MCA Against Surface Area

Figure 3.20. Average Edge-cut for MCA against Surface Area
CHAPTER 4

BACKBONE FORMATION ALGORITHM

4.1. General Idea and Description of the Algorithm

MCA partitions the network into balanced clusters but it lacks a backbone since the clusterheads are not connected in one hop distance or a connection path between clusterheads is not defined. On the other hand, DS-based algorithms which are described in Section 2.1.2 constructs a backbone of clusterheads, however, balanced clustering is not mentioned. The backbone formation algorithm constructs a backbone architecture on a clustered MANET. The backbone is constructed as a directed ring architecture to gain the advantage of this topology and to give better services for other middleware protocols such as distributed mutual exclusion (Erciyes 2004, 2005) and total order multicast. The second contribution is to connect the clusterheads of a balanced clustering scheme which completes two essential needs of clustering by having balanced clusters and minimized routing delay. Beside these, the backbone formation algorithm is fault tolerant as the third contribution.

The main idea is maintaining a directed ring architecture by constructing a minimum spanning tree between clusterheads and classifying clusterheads into BACKBONE or LEAF nodes, periodically. To maintain these structures, each clusterhead broadcasts a Leader_Info message by flooding. In this phase, cluster member nodes are acting as router to transmit Leader_Info messages. Algorithm has two modes of operation; hop-based backbone formation scheme and position-based backbone formation scheme. In hop-based backbone formation scheme, minimum number of hops between clusterheads are taken into consideration in the minimum spanning tree construction. Minimum hop counts can be obtained during flooding scheme. For highly mobile scenarios, an agreement between clusterheads must be maintained to guarantee the consistent hop information. In position-based backbone formation scheme, positions of clusterheads are used to construct the minimum spanning tree. If each node knows its velocity and the direction of velocity, these information can be appended with a timestamp to the Leader_Info message to construct better minimum spanning tree. But in this mode, nodes must be equipped
with a position tracker like a GPS receiver.

Every node in the network performs the same local algorithm. The finite state machine of the algorithm is shown in Fig. 4.1. Each node can be either in IDLE, BACKBONE or LEAF states described below.

![Finite State Machine of the Backbone Formation Algorithm](image)

**Figure 4.1. Finite State Machine of the Backbone Formation Algorithm**

- **IDLE**: Initially all the clusterheads are in IDLE state. If Period_TOUT occurs, each clusterhead broadcasts a Leader_Info message to destination node and will make a state transition to WT_INFO state. If Leader_Info message is received, the clusterhead makes a state transition to LEAF state and reconstructs the ring by reorganizing the minimum spanning tree.

- **WT_INFO**: A clusterhead in WT_INFO state waits for Leader_Info message. If a Leader_Info message is received, the clusterhead makes a state transition to LEAF state and reconstructs the ring. If TOUT occurs, clusterhead makes a transition to LEAF state which indicates that network has only two active partitions.

- **LEAF**: A clusterhead in LEAF state has a degree of 1 in its local minimum spanning tree. If a Leader_Info message is received, the clusterhead reconstructs the
ring and makes a state transition to BACKBONE state if the degree exceeds 1. If Period.TOUT occurs, clusterhead makes a transition to IDLE state to restart backbone formation.

- **BACKBONE**: A clusterhead in BACKBONE state has degree greater than 1. For each Leader.Info message received, the ring is reconstructed. If Period.TOUT occurs, backbone formation is restarted.

![Figure 4.2. A MANET with its minimum spanning tree](image)

A balanced clustered MANET with its clusterheads and minimum spanning tree is shown in Fig. 4.2. BACKBONE clusterheads are filled with black and LEAF clusterheads are filled with white. The main part of the algorithm is the construction of a ring architecture by orienting clusterheads in the minimum spanning tree. General idea is to divide the ring into two parts, a directed path of BACKBONE clusterheads and a directed path of LEAF nodes. Finally, these two directed paths are connected each other to maintain the ring architecture. Each clusterhead aims to find the next clusterhead(leader) to construct the ring architecture by the procedure in Fig. 4.1.
ring_constructProc

begin

construct minimum spanning tree by total received leader information.

if my degree is equal to 1 then
execute ordinary_leaf_proc.

else
set my state to BACKBONE.

end

if I am a BACKBONE leader or a LEAF leader which can’t find next
leader then
execute backbone_proc.

end

end

Algorithm 4.1. Procedure executed by all leaders to construct a Ring Architecture

First aim is to make the vital part of backbone formation. The BACKBONE clusterheads are directing each other from starting BACKBONE clusterhead to the end. Starting BACKBONE clusterhead is the one with smallest connectivity to other BACKBONE nodes. This selection policy of BACKBONE clusterhead results in smaller hops and reduced routing delay. Ending BACKBONE clusterhead is directing to its LEAF’s with smallest node_id.
Algorithm 4.2. Procedure executed by BACKBONE leaders and LEAF leaders which can’t find next leader

LEAF leaders firstly execute the procedure in Fig. 4.4 to find the next leader on the ring. The aim of directing LEAF leaders with the same BACKBONE leaders to each other is to make routing process over the same BACKBONE leader to reduce delay. LEAF leaders which can’t find the next leader, executes the procedure in Fig. 4.2 and searches a LEAF leader from the previous BACKBONE leaders of their parent to find a LEAF leader. Last aim is to connect LEAF leaders of different BACKBONE parents to maintain routing operation by using BACKBONE leaders.
Algorithm 4.3. Procedure executed to connect BACKBONE leaders

Third contribution of the algorithm is the fault tolerance of clusterheads. Each clusterhead can maintain the list of cluster member nodes. In our backbone formation algorithm, this list can be appended to Leader_Info message by each clusterhead. After the formation of the ring is completed, if a clusterhead detects the crash of the next clusterhead, it can multicast a Leader_dead message to all cluster members which initiates
clustering operation. To support this functionality, clustering layer must be updated. If this crash occurs during a real time operation, clusterhead updates its next leader to next-next leader and continues its operation since it knows the global information of all clusterheads.

\[
\text{ordinary_leaf_proc} \\
\begin{align*}
\text{begin} \\
\text{set my state to LEAF.} \\
\text{Find a LEAF leader with same parent and nearest greater node_id.} \\
\text{if found then} \\
\text{set my next leader to this LEAF leader’s node_id and mark this LEAF.} \\
\text{end}
\end{align*}
\text{end}
\]

Algorithm 4.4. Procedure executed by LEAF leaders

4.2. An Example Operation

![Figure 4.3. An Example Operation for Backbone Formation Algorithm](image)

Figure 4.3. An Example Operation for Backbone Formation Algorithm
Assume the MANET with clusterheads (leaders) in Fig. 4.3a. Clusters are obtained using MCA. Nodes 65, 15, 98, 30, 40, 13, 28, 80, 74, 19, 51 and 99 are leaders of clusters 1 to 12, respectively. Each clusterhead floods the Leader_Info message to the network. After each clusterhead receives the Leader_Info message of the others, minimum spanning tree in Fig. 4.3a is constructed by all clusterheads. Nodes 65, 98, 40, 13, 80, 19 and 99 identify themselves as LEAF leaders since their degree are all 1. Node 15, 30, 28, 74 and 51 identify themselves as BACKBONE leaders since their degrees are greater than 1. BACKBONE leaders are filled with black and LEAF leaders left unfilled as shown in Fig. 4.3a.

To connect BACKBONE nodes, a starting BACKBONE leader must be chosen. The criteria is to select the BACKBONE node which has the smallest connection to other BACKBONE leaders. Node 15 is connected to 30, 30 is connected to 15 and 28, 28 is connected to node 30 and node 74, node 74 is connected to node 28 and 51, 51 is connected to 74. Either of the node 15 and 51 can be the choice for starting BACKBONE leader. Node 15 is selected because its node_id is smaller. 15 selects the next leader as 30, 30 selects the next leader 28, operation continues in this way. Ending BACKBONE leader points to its LEAF with the smallest node_id. These directions can be seen in Fig. 4.3b with bold directed lines.

LEAF leaders of a BACKBONE leader are directed to each other from smallest to greatest. Node 19 is directed to 99, 13 is directed to 80, 65 is directed 98 as seen in Fig. 4.3c with dotted directed lines.

Lastly, LEAF leaders of different BACKBONE leaders are connected as in Fig. 4.3d. Each LEAF leader which can’t find next leader, searches a LEAF leader from children of previous BACKBONE leader of its parent BACKBONE leader. 99 is connected to 13, 80 is connected to 40, 40 is connected 65, 98 is connected to 15 as shown with dashed lines in Fig. 4.3d.

4.3. Analysis

Time and message complexity is given in this section. Also a comparison TBONE "(Rubin et al. 2002)", d-hop algorithm "(Ya-feng et al. 2004)”, SBC "(Haitao and Gupta 2004)”, RVBSM "(Min et al. 2005)”, EBS "(Huejium and Rubin 2005)" and Backbone
Table 4.1. Comparison of Backbone Constructing Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Messaging Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBONE</td>
<td>not given</td>
<td>not given</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d-hop algorithm</td>
<td>$O(Dn)$, $D$: graph diameter</td>
<td>$O(n^d)$</td>
</tr>
<tr>
<td>SBC</td>
<td>not given</td>
<td>not given</td>
</tr>
<tr>
<td>RVBSM</td>
<td>$O(n)$</td>
<td>$O(Dn)$, $D$: maximum degree</td>
</tr>
<tr>
<td>EBS</td>
<td>$O(1)$ per node</td>
<td>$O(1)$ per node</td>
</tr>
<tr>
<td>BFA</td>
<td>$O(Kn)$, $K$: # of clusters</td>
<td>$O(Kn)$</td>
</tr>
</tbody>
</table>

Table 4.2. Comparison of Backbone Constructing Algorithms cont...

<table>
<thead>
<tr>
<th>Main Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBONE</td>
</tr>
<tr>
<td>d-hop algorithm</td>
</tr>
<tr>
<td>SBC</td>
</tr>
<tr>
<td>RVBSM</td>
</tr>
<tr>
<td>EBS</td>
</tr>
<tr>
<td>BFA</td>
</tr>
</tbody>
</table>

Formation Algorithm(BFA) can be seen in table 4.1 and 4.2.

**Theorem 4.3.1.** Message complexity of the backbone formation algorithm is $O(Kn)$.

**Proof.** Assume that we have $n$ nodes in our network. $K$ leaders flood the message to the network. Total number of messages in this case is $Kn$ which means that message complexity has an upper bound of $O(Kn)$. \(\Box\)

**Theorem 4.3.2.** Time complexity of the backbone formation algorithm is $O(Kn)$.

**Proof.** Assume that we have $n$ nodes in our network. Flooding of $K$ messages to the network takes $Kn$ time. \(\Box\)
4.4. Results

The position-based backbone formation algorithm is implemented with the ns2 simulator. Clustering is obtained using the MCA algorithm. Number of nodes in the clusters can be adjusted by the $K$ parameter of MCA. Different size of flat surfaces are chosen for each simulation to create medium, small and very small distances between nodes. Medium, small and very small surfaces vary between respectively $310\times 310$ m to $400\times 400$ m, $410\times 410$ m to $500\times 500$ m, $515\times 515$ m to $650\times 650$ m. Random movements are generated for each simulation. Low, medium and high mobility scenarios are generated and node speeds are limited between 1.0 m/s to 5.0 m/s, 5.0 m/s to 10.0 m/s, 10.0 m/s to 20.0 m/s respectively. $K$ parameter of merging clustering algorithm is changed to obtain different number of clusterheads.

Round-trip delay as measured against number of clusterheads, total number of nodes, mobility and surface area are recorded. As depicted in Fig. 4.4, the time complexity increases linearly and at worst, backbone formation scheme is completed in 1.5 s for a MANET with 100 nodes. For a MANET with 50 nodes, number of clusterheads are selected from 3 to 8 to measure the round-trip delay in Fig. 4.7. A linear increase can be seen in Fig. 4.7 which starts from 35 ms and ends in 65 ms approximately. Round-trip delay against total number of nodes is measured with constant 4 clusters. Total number of nodes are varied between 10 to 100 in Fig. 4.8. Round-trip delay times are increasing linearly from 20 ms to 60 ms approximately as shown in Fig. 4.8. In small surface scenarios connectivity between nodes are higher because of small distance between nodes. Connectivity between nodes decreases the routing delay. Fig. 4.6 shows the effects of distance between nodes to round-trip delay of the ring. Lastly, mobility parameter is changed to obtain the behavior of the algorithm with respect to mobility. Our algorithm results in approximate round-trip delay values for high mobile scenarios as shown in Fig. 4.5.

Consequently, our results conform with the analysis that run-time values are growing linearly. Round-trip delay increases linearly against the total number of nodes and cluster number in MANET. Round-trip delay values are stable under different mobility conditions and different size of surface areas.
Figure 4.4. Run-time Performance for Backbone Formation Algorithm

Figure 4.5. Round-trip Delay Against Mobility for BFA

Figure 4.6. Round-trip Delay Against Surface Area for BFA
Figure 4.7. Round-trip Delay Against number of Clusterheads for BFA

Figure 4.8. Round-trip Delay Against number of Nodes for BFA
CHAPTER 5

MOBILE RICART-AGRAWALA ALGORITHM

5.1. General Idea and Description of the Algorithm

For distributed mutual exclusion in MANETs, a hierarchical architecture is constructed where nodes form clusters and each cluster is represented by a coordinator in the ring ")(Erciyes 2004, 2005). After constructing this architecture, we can execute the Mobile_RA algorithm. The main idea is to form coordinators as interface of other nodes to the ring. The relation between the cluster coordinator and an ordinary node is similar to a central coordinator based mutual exclusion algorithm. The types of messages exchanged are Node_Req, Coord_Req, Coord_Rep and Node_Rel where a node first requests a critical section and upon the reply from the coordinator, it enters its critical section and then releases the critical section. The message types are described below:

- **Node_Req**: Any node which wants to execute CS, sends a Node_Req message to its coordinator. After sending this message, node waits for Coord_Rep message from its coordinator to execute CS.

- **Coord_Req**: After receiving Node_Req message, Coordinator sends this message to the next coordinator on the ring if all pending requests in its cluster have greater timestamps than this incoming request. Otherwise, it enqueues this message to the wait_queue. If a coordinator receives this message, it forwards it according to the timestamp of this message. If the coordinator which is the originator of this message receives its own message, it sends a Coord_Rep message if there is no other waiting requests with timestamp lower than itself or a CS executing node at the same time.

- **Coord_Rep**: A node which wants to execute CS, will execute CS after receiving this message from its coordinator.

- **Node_Rel**: After executing CS, node sends this message to its coordinator indicating that their execution is completed.
The coordinators can be either in IDLE, WAITRP or WAITND state as described below:

- **IDLE**: Coordinators in IDLE state only forwards the Coord\_Req messages to the next coordinator on the ring. If a Node\_Req message is received, coordinator sends a Coord\_Req message to the next coordinator and makes a transition to WAITRP state.

- **WAITRP**: Coordinator in WAITRP state waits for its original Coord\_Req message. Coord\_Req messages of other coordinators are enqueued if the timestamp is greater. After receiving its original Coord\_Req message, coordinator makes a transition to WAITND state. Received Node\_Req messages are forwarded as Coord\_Req messages to next coordinators.

- **WAITND**: Coordinator in WAITND state waits for Node\_Rel message from its cluster member which executes CS. After receiving Node\_Rel, it makes a transition to IDLE state, if there is no pending request from its cluster members in its wait\_queue. Otherwise it makes a transition to WAITRP state. Received Node\_Req messages are forwarded as Coord\_Req messages to the next coordinators.

The finite state machine representation of the Mobile\_RA coordinator is shown in Fig. 5.1” (Erçiyès 2004, 2005).
5.2. An Example Operation

Fig. 5.2 shows an example scenario for the Mobile_RA Algorithm. Node 6, node 4, node 16 makes request for critical section respectively at 3.75s, 3.85s, 3.90s. Execution Time of critical section is taken as 350ms. The following describes the events that occur:

1. Node 6, in cluster 19 makes a critical section request at 3.75s by sending Node Req \((6,19,3.75)\) message to node 19 which is the cluster coordinator. Node 19 receives the message at 3.76s and changes its state to WAIT_RP. Node 19 sends a Coord Req \((6,19,3.75)\) message to the next coordinator(node 14) on the ring. Node 14, which is in IDLE state and has no pending requests in its cluster, receives the Coord Req \((6,19,3.75)\) message at 3.78s and forwards the message to the next coordinator(node 17) on the ring. The message traverses the ring and is received by node 19 which is in WAIT_RP state at 3.82s meaning all of the coordinators have confirmed that either they have no pending requests or their pending requests all have higher timestamps. Node 19 sends a Coord Rep message to node 6 and changes its state to WAIT_ND. Node 6 receives the Coord Rep message at 3.83s and enters the critical section. Step 1 is depicted in Fig. 5.2(a).

2. Node 4, in cluster 18 makes a critical section request by sending a Node Req \((4,18,3.85)\) at 3.85s. Node 18 receives the Node Req \((4,18,3.85)\) message at 3.86s and sends a Coord Req \((4,18,3.85)\) message to its next coordinator(node 19) on the ring. Node 19, which is in WAIT_ND state, receives the message and enqueues the Coord Req \((4,18,3.85)\) at 3.87s. Node 16 makes a critical section request at 3.90s. Node 18 which is in WAIT_RP state receives the Coord Req \((16,17,3.90)\) message and enqueues the message at 3.93s. Step 2 is depicted in Fig. 5.2(b).

3. Node 6 exits from critical section at 4.18s and sends a Node Rel message to node 19. Node 19 which is in WAIT_ND state receives the message at 4.19s and makes a transition to IDLE state. Node 19 dequeues and forwards Coord Req \((4,18,3.85)\) message to the next coordinator(node 14). The Coord Req \((4,18,3.85)\) message is forwarded by node 17 since its request has higher timestamp. Node 18 receives its original request at 4.25s and sends a Coord Rep message to node 4. Node 4 enters the critical section at 4.26s. Step 3 is depicted in Fig. 5.2(c).
Figure 5.2. Operation of the Mobile_RA Algorithm

4. Node 4 finishes to execute critical section at 4.61s. Node 18 receives the Node_Rel message at 4.62s. Node 18 dequeues and forwards the Coord_Req (16,17,3.90) message to its next coordinator (node 19) on the ring. Operation is continued as explained before. Node 17 receives Node_Rel message from node 16 at 5.03s. The Step 4 is depicted in Fig. 5.2.(d).

If there are multiple requests within the same cluster, time stamps are checked similarly for local request. The order of execution in this example is nodes 4 → 6 → 16 in the order of the timestamps of the requests.

5.3. Analysis

Since the sending and receiving ends of the Mobile_RA algorithm are the same as of RA algorithm, the safety, liveness and fairness attributes are the same. The upper
and lower bounds for total message number, response time and synchronization delay are shown and proven in theorems and corollaries below. "(Erciyes 2004, 2005)". The comparison of Mobile_RA and RA is shown in table 5.1.

**Theorem 5.3..1.** The total number of messages per critical section using the Mobile RA Algorithm is $k+3d$ where $k$ is an upper bound on the number of neighbor nodes in the ring including the cluster coordinators and $d$ is an upper bound on the diameter of a cluster.

*Proof.* An ordinary node in a cluster requires three messages (Node Req, Coord Rep and Node Rel) per critical section to communicate with the coordinator. Each of these messages would require maximum $d$ transfers between a node and the coordinator. The full circulation of the coordinator request (Coord Req) requires $k$ messages resulting in $k+3d$ messages in total.

**Corollary 5.3..2.** The Synchronization Delay ($S$) in the Mobile RA Algorithm varies from $2dT$ to $(k + 2d - 1)T$.

*Proof.* When the waiting and the executing nodes are in the same cluster, the required messages between the node leaving its critical section and the node entering are the Node Rel from the leaving node and Coord Rep from the coordinator resulting in $2dT$ message times for $S_{\text{min}}$. However, if the nodes are in different clusters, the Node Rel message has to reach the local coordinator in $d$ steps, circulate the ring through $k-1$ node to reach the originating cluster coordinator in the worst case and a Coord Rep message from the coordinator is sent to the waiting nodes in $d$ steps resulting in $S_{\text{max}} = (k-1)T + 2dt = (k+2d-1)T$.

**Corollary 5.3..3.** In the Mobile RA Algorithm, the response times are $R_{\text{light}} = (k + 3d)T + E$ and $R_{\text{heavy}}$ varies from $w(2dT + E)$ to $w((k + 2d-1)T + E)$ where $k$ is the number of clusters and $w$ is the number of pending requests.

*Proof.* According to Theorem 5.3..1, the total number of messages required to enter a critical section is $k+3d$. If there are no other requests, the response time for a node will be $R_{\text{light}} = (k+3d)T + E$ including the execution time (E) of the critical section. If there are $w$ pending requests at the time of the request, the minimum value $R_{\text{heavy, min}}$ is $w(2dT + E)$. In the case of $S_{\text{max}}$ described in Corollary 5.3..2 $R_{\text{heavy, max}}$ becomes $w((k + 2d-1)T + E)$ since in general $R_{\text{heavy}} = w(S + E)$. 

54
Table 5.1. Comparison of Mobile_RA and RA

<table>
<thead>
<tr>
<th></th>
<th>Mobile_RA</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Message Count</td>
<td>k+3d</td>
<td>2(N-1)</td>
</tr>
<tr>
<td>Response time_light</td>
<td>(k+3d)T+E</td>
<td>2T+E</td>
</tr>
<tr>
<td>Response time_heavy</td>
<td>w((k-1+2d)T+E)</td>
<td>w(T+E)</td>
</tr>
<tr>
<td>Synchronization delay</td>
<td>(k+2d-1)T</td>
<td>NT</td>
</tr>
</tbody>
</table>

5.4. Results

We implemented the distributed mutual exclusion algorithm with the ns2 simulator. A random load generator is implemented to generate high, medium and low loads for different number of nodes. Different size of flat surfaces are chosen for each simulation to create small, medium and large distances between nodes.

Very Small, Small and Medium surfaces vary between 310m × 310m to 400m × 400m, 410m × 410m to 500m × 500m, 515m × 515m to 650m × 650m respectively. Random movements are generated for each simulation. Low, medium and high mobility scenarios are generated and respective node speeds are limited between 1.0m/s to 5.0m/s, 5.0m/s to 10.0m/s, 10.0m/s to 20.0m/s. $K$ parameter of merging clustering algorithm is changed to obtain different size of clusters. Response times and synchronization delays as measured with respect to load, mobility, distance and $K$ are recorded. Execution of critical section is selected as 100ms.

Response time behaves as expected in low load scenarios as shown in Fig. 5.3. Synchronization delay values are smaller in medium load as shown in Fig. 5.4. The synchronization delay is 0 in low load scenarios since there will be no waiting requests in the queues. When the load is increased response time increases due to waiting time of requests in the queue. Also response time and synchronization delay increases due to collisions and routing delays caused by high network traffic as shown in Fig. 5.3 and Fig. 5.4. Response time and synchronization delay results in approximate values by mobility parameter due to rapid change of network topology as shown in Fig. 5.5 and Fig. 5.6.
Figure 5.3. Response Time against Load for Mobile_RA

Figure 5.4. Synchronization Delay against Load for Mobile_RA

Fig. 5.7 and Fig. 5.8 shows the effects of distance between nodes to response time and synchronization delay. $K$ parameter is selected between 3 to 8 in a MANET with 60 node. In fixed number of nodes, as the cluster size increases, total number of clusters in network decreases. This also reduces the number of cluster leaders forming the ring and routing delay which causes decreasing the response time and synchronization delay as shown in Fig. 5.9 and Fig. 5.10.
Figure 5.5. Response Time against Mobility for Mobile_RA

Figure 5.6. Synchronization Delay against Mobility for Mobile_RA
Consequently, our results conform with the analysis that response time values against low and medium loads increase linearly with a small gradient. Synchronization delay values against medium and high load also increase linearly. Response time against high load makes a sharp increase due to high network traffic. Response time and synchronization delay values are stable under different mobility and surface area conditions. Response time and synchronization delay values decrease linearly against the number of clusters in MANET.
Figure 5.9. Response Time against K for Mobile_RA

Figure 5.10. Synchronization Delay against K for Mobile_RA
CHAPTER 6

CONCLUSION

We proposed and implemented three protocols on a hierarchical architecture to solve the balanced clustering, backbone formation and distributed mutual exclusion problems for mobile ad hoc networks (MANET)s. First layer is the clustering layer which is solved by the design and implementation of Merging Clustering Algorithm. The original idea of the Merging Clustering Algorithm is to focus on the clustering operation by discarding the details of minimum spanning tree algorithms to reduce time and message complexity. The second contribution is the usage of lower and upper bound heuristics which results balanced number of nodes in the clusters formed. We describe the algorithm, present it by a finite state machine and explain the states and message types in this finite state machine. We analyzed its time and message complexity. We first measure the runtime performance, total message number. Cluster quality against $K$ for 40 and 60 nodes, against mobility for 40 nodes, against surface area for 40 nodes and against total number of nodes are measured. We also measure the total edge cut, average edge cut against $K$, mobility and surface area. The implementation results obtained conform with the theoretical analysis and show that the algorithm is scalable in terms of its running time and produces evenly distributed clusters.

Backbone Formation Algorithm solves the problem of second layer, backbone construction layer. The original idea of the Backbone Formation Algorithm is the construction of backbone architecture as a directed ring. The second contribution is to connect the clusterheads of a balanced clustering scheme which completes two essential needs of clustering by having balanced clusters and minimized routing delay. Beside these, the backbone formation algorithm is fault tolerant as the third contribution. We describe the algorithm by its finite state machine representation and explain necessary details. Also, the procedures used to construct the ring is included. We measure the runtime performance, round-trip delay against clusterhead number, total number of nodes, mobility and surface area is measured. The implementation results shows that the algorithm is scalable in terms of its running time and round-trip delay against mobility, surface area, number of nodes and number of clusterheads.
Lastly, we implement the Mobile Ricart-Agrawala Algorithm on top of these protocols. The original idea of the Mobile Ricart-Agrawala Algorithm (Mobile_RA) is the construction of the hierarchical architecture where nodes form clusters and each cluster is represented by a coordinator in the ring. The MANET is partitioned into clusters at regular intervals by the Merging Clustering Algorithm which also provides clusterheads same as coordinators. The Backbone Formation Algorithm provides directed ring architecture. The Mobile_RA Algorithm, together with the architecture that it is executed on, provides improvement over message complexities of Ricart and Agrawala and other distributed mutual exclusion algorithms. The algorithm is explained by finite state machine representation. The analysis of the algorithm supports to sketch upper and lower bounds for response time and synchronization delay. Response time and synchronization delay against load type, surface area, mobility, $K$ heuristic and total number of clusters is measured. From the test results, we observe that response time $R$ and synchronization delay $S$ is scalable with respect to the number of mobile nodes for all load states in the MANET as high, medium or low loads. $R$ and $S$ is also scalable with respect to node mobility and the distance between the mobile nodes.

![Figure 6.1. Our architecture](image)

In summary, we proposed a three layer architecture for MANETs as shown in Fig. 6.1. Implementations of other higher level functions on top of the last two layers are possible and future studies can benefit from this architecture. We are also planning to make enhancements for Merging Clustering Algorithm, Backbone Formation Algorithm and Mobile Ricart Agrawala Algorithm. We first plan to reduce the delay of clustering
process which is effected by exposed terminal problem. Secondly, we are planning to implement a proactive cluster based routing protocol for MANETs by using the clusters of MCA. After implementing routing protocol, $K$ parameter can be dynamically calculated by all nodes since the number of active nodes in the network can be estimated. Cluster gateway nodes will be defined by MCA to reduce the message overhead of BFA. A clock synchronization algorithm will be maintained for Mobile_RA. After these enhancements total order multicast protocol will be implemented. Lastly, we plan to adopt these protocols to the Sensor Network environment.
REFERENCES


APPENDIX A

SIMULATION SETUP

GloMoSim “(Zeng et al. 1998)”, QualNet “(Scalable Network Technologies Corporation 2005)”, ns2 “(Fall and Varadhan 2006)”, and OPNET “(OPNET Technologies Corporation 2006)” are programs used to simulate MANETs. We choose ns2 simulator for several reasons. Firstly, ns2 is probably the most commonly used software of the four. The source code can be downloaded, free of charge, and compiled on different platforms, e.g. Unix and Windows. Many wireless extensions have been contributed from the UCB Daedelus, the CMU Monarch projects and Sun Microsystems. In ns2, it’s possible to alter and write your own code to make it more suitable for your own scenarios “(Halvardsson and Lindberg 2004)”.

The wireless model essentially consists of the MobileNode class at the core, with additional supporting features that allows simulations of multi-hop ad-hoc networks, wireless LANs etc. The MobileNode object is a split object. The C++ class MobileNode is derived from parent class Node. MobileNode is the basic ns2 Node object with added functionalities like movement, ability to transmit and receive on a channel that allows it to be used to create mobile, wireless simulation environments. The mobility features including node movement, periodic position updates, maintaining topology boundary etc. are implemented in C++ while plumbing of network components within MobileNode itself (like classifiers, dmux, LL, Mac, Channel etc) have been implemented in Otcl. The network stack for a mobile node consists of a link layer(LL), an ARP module connected to LL, an interface priority queue(IFq), a mac layer(MAC), a network interface(netIF), all connected to the channel. These network components are created and plumbed together in Otcl “(Fall and Varadhan 2006)”. To create and configure nn number of mobile nodes with network stack, Otcl codes are in Fig. A.1.

The protocol stack is implemented in C++. We must firstly modify the existing UDP and MAC layer implementations to communicate with our new layers. UDP-Mergyclus class header with its comments can be seen in Fig. A.2. Modifications to recvACK method of MAC802_11 class must be made to communicate with upper layers which can be seen in Fig. A.3. Also a new Queue must be implemented to ensure the
packet delivery. *MCFQue* class which wraps a queue implementation can be seen in Fig. A.5. A singleton *UDPMCContainer* class is implemented to provide communication of layers which can be seen in Fig. A.4. MCA, BFA and Mobile_RA is implemented respectively by *MergClusApp* class header in Figs. A.6, A.7 and A.8. *RingApp* class header in Figs. A.9 and A.10. *MCDMAp* class header in Figs. A.11 and A.12 with their comments.

The mobile node is designed to move in a three dimensional topology. However the third dimension (Z) is not used. That is, the mobile node is assumed to move always on a flat terrain with Z always equal to 0. Thus the mobile node has X, Y, Z(=0) coordinates that is continually adjusted as the node moves. The node movement is defined in a separate file for convenience. Movement file can be generated using CMU’s movement generator. By this generator, one can set number of nodes, pause time, maximum speed, minimum speed, simulation time, maximum X and Y coordinates “(Fall and Varadhan 2006)”. The executable command of movement generator is given below:

- `setdest -v <version> -n <number of nodes> -m <minimum speed(m/s)> -M <maximum speed(m/s)> -t <simulation time(s)> -p <pause time> -x <width of surface area> -y <height of surface area> > scenario_file`

In our simulations, we use the last version(2) of *setdest* command. We generate very small, small and medium surfaces which vary between 310m × 310m to 400m × 400m, 410m × 410m to 500m × 500m, 515m × 515m to 650m × 650m respectively. Low, medium and high mobility scenarios are generated and respective node speeds are limited between 1.0m/s to 5.0m/s, 5.0m/s to 10.0m/s, 10.0m/s to 20.0m/s. Pause time is set to the simulation time divided by 5.

After protocols are implemented in *C++*, the scenario files are written in *Otcl*. A complete scenario file is given with its comments in Figs. A.13, A.14 and A.15.
$ns_\ node-config
-\ adhocRouting \ $opt\(\text{adhocRouting}\)
-\ llType \ $opt\(\text{ll}\)
-\ macType \ $opt\(\text{mac}\)
-\ ifqType \ $opt\(\text{ifq}\)
-\ ifqLen \ $opt\(\text{ifqlen}\)
-\ antType \ $opt\(\text{ant}\)
-\ propInstance \ [\text{new} \ $opt\(\text{prop}\)]
-\ phyType \ $opt\(\text{netif}\)
-\ channel \ [\text{new} \ $opt\(\text{chan}\)]
-\ topoInstance \ $\text{topo}$
-\ wiredRouting \ OFF
-\ agentTrace \ ON
-\ routerTrace \ OFF
-\ macTrace \ OFF
for \ { \ set \ j \ 0 } \ { \ j < \ $opt\(\text{mn}\) } \ { \ incr \ j } \ { 
  \ set \ node\_($j) \ [ \ $ns\_node ] 
} 

Figure A.1. *Otel* code to create and configure mobile node
#ifndef ns_udp_mergclus_h
#define ns_udp_mergclus_h
#include "udp.h"
#include "ip.h"
#include "mc_protocol.h"

// Merging Clustering Header Structure
struct hdr_mergclus {
    int nbytes; // bytes for mergclus pkt
    double time; // current time
    int source; // source address
    int destination; // destination address
    int real_destination; // real destination
    int real_source; //use for ldr_poll_node message
    int cluster_level; // Cluster Level
    int cluster_leader; // Cluster Leader
    int old_cluster_leader; // Old Cluster Leader
    int message_type; //Message Type
    int leader_states[N_NODES]; //
    int message_propagation_type;//Message Propagation Type
    ra_req_t request;
};

// Packet header access functions
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_mergclus* access(const Packet* p) {
    return (hdr_mergclus*) p->access(offset_);
}

// UdpMergClusAgent Class definition
class UdpMergClusAgent : public UdpAgent {
    public:
        UdpMergClusAgent(); // Constructor
        UdpMergClusAgent(packet_t); // Constructor
        Application* getApplication(); // Sends application reference
        virtual int supportMergClus() { return 1; }
        virtual void enableMergClus() { support_mergclus_ = 1; }
        // Send and receive functions
        virtual void sendmsgdst(int nbytes, const char *flags = 0, int32_t dist=0);
        virtual void sendmsg(int nbytes, const char *flags = 0);
        void recv(Packet*, Handler*);
        // Notification for ACKs
        void notification(int src);
};
#endif

Figure A.2. UDPMergClusAgent class header
void Mac802_11::recvACK(Packet *p) {
  
  int source=-1;
  if(p!=NULL){
    struct ack_frame *af = (struct ack_frame*)p->access(hdr_mac::offset_);
    source=af->src;
  }
  
  int my_addr=addr();
  UdpMergClusAgent* udp =
      (UdpMergClusAgent*)UDPMCContainer::instance().getElementById(my_addr);
  udp->notification(source);
}

Figure A.3. Modifications in Mac802_11 recvACK method

// Author: Orhan Dagdeviren
// File: udpmccontainer.h // Singleton class for
//        // communicating layers
// Date: 1/6/2006 (for ns2.28)
#ifndef ns_udpmc_h
#define ns_udpmc_h
#include "udp-mc.h"
class UDPMCContainer{
public:
  UDPMCContainer(); // Constructor
  static UDPMCContainer& instance() {
    return (*instance_); // general access to UDPMContainter
  }
  UdpMergClusAgent* udpcont[N_NODES]; //Reference array
  void addObject(UdpMergClusAgent *element); // Adds an object to
    // reference array
  void initInstance(); // Initializes the instance
  UdpMergClusAgent* getElementById(int paramid); // returns
    // reference with id
protected:
  static UDPMCContainer* instance_; // container instance
  int index;
};
#endif

Figure A.4. UDPMCContainer class header
class MCFQue : public Application
{
    public:
        MCFQue(); // Constructor
        void init(); // Initializes the queue
        void init_fqueue();
        void check_fqueue(); // Checks the queue
        void enqueue_fqueue(hdr_mergclus *element); // Enqueues
        // into queue
        void dequeue_fqueue(); // Dequeues from queue
        void print_fqueue(); // Prints elements in queue
        hdr_mergclus send_queue[MAX_SEND]; // Queue
        int number_to_send; // Number of elements for sending
        int head; // Index of head element
        int debug; // Debug status
        int ready_to_send; // Ready-to-send status
    protected:
        int command(int argc, const char* const argv); // Otcl command
        // linkage
};

Figure A.5. MCFQue class header
MergClusApp and MCTimer classes.

```cpp
class MCTimer : public TimerHandler {
public:
    MCTimer(MergClusApp* t) : TimerHandler(), t_(t) {}
    inline virtual void expire(Event*);
    inline virtual void init();
    inline virtual void tout_on(double added_time);
    inline virtual void tout_off();
    inline virtual void set_tout_seconds(double seconds);
    inline virtual void tout_schedule(double time);
    void state_tout();
protected:
    MergClusApp* t_;  
    double tout_seconds;
    int tout_state;
};
```

```cpp
class MergClusApp : public Application {
public:
    MergClusApp(); //Constructor
    // Send an receive packets
    void send_mergclus_pkt(int32_t dist, int32_t real_source_p, int message_type,
                            int cluster_level_p, int cluster_leader_p, int old_cluster_leader_p);
    void send_msg(int nbytes, const char* msg);
    virtual void recv_msg(int nbytes, const char *msg = 0); // (Sender/Receiver)
    //Action Not Applicable
    void actNA(void *message);
    //IDLE State transition functions
    void idle_poll_node(void *message);
    void idle_period_tout(void *message);
    void idle_tout(void *message);
    //WT_INFO State transition functions
    void wt_info_node_info(void *message);
    void wt_info_period_tout(void *message);
    void wt_info_tout(void *message);
    //WT_ACK State transition functions
    void wt_ack_mbr_ack(void *message);
    void wt_ack_ldr_ack(void *message);
    void wt_ack_period_tout(void *message);
    void wt_ack_tout(void *message);
};
```

Figure A.6. MergClusApp class header

76
//MEMBER State transition functions
void member_poll_node(void *message);
void member_change_cluster(void *message);
void member_period_tout(void *message);
void member_tout(void *message);

//LEADER State transition functions
void leader_poll_node(void *message);
void leader_ldr_poll_node(void *message);
void leader_period_tout(void *message);
void leader_tout(void *message);
void leader_k_check(void *message);

//LDR_WT_CONN State transition functions
void ldr_wt_conn_connect_mbr(void *message);
void ldr_wt_conn_connect_ldr(void *message);
void ldr_wt_conn_period_tout(void *message);
void ldr_wt_conn_tout(void *message);

//IDLE_WT_CONN State transition functions
void idle_wt_conn_connect_mbr(void *message);
void idle_wt_conn_connect_ldr(void *message);
void idle_wt_conn_period_tout(void *message);
void idle_wt_conn_tout(void *message);

//LDR_WT_ACK State transition functions
void ldr_wt_ack_change_cluster_ack(void *message);
void ldr_wt_ack_period_tout(void *message);
void ldr_wt_ack_tout(void *message);
void ldr_wt_ack_poll_node(void *message);
void ldr_wt_ack_ldr_poll_node(void *message);
void ldr_wt_ack_connect_ldr(void *message);

// To manage leader status
void reset_chg_clu_acks();
void reset_leader_states();
int check_acks();

// Poll neighbor functions
void poll_neighbor_next();
void poll_neighbor();

// Broadcasts CHANGE_CLUSTER
void broadcast_change_cluster(int old_cluster_leader_p);

// Finite state function for jumping to next state
void fsm_jump_next_state(void *message);
void init();
void init_fsm();

// Utility functions
char * message_to_string(int message_type_p);
char * state_to_string();

// Debug message function
void debug_msg(void *message);

// Notification functions for lower and upper layers
void notification(int source);
virtual void upper_notification();
void print_end_info(); // Prints end info

Figure A.7. MergClusApp class header, cont...
// Protocol variables
int current_state;
int previous_state;
int cluster_level;
int cluster_leader;
int peer_old_cluster_leader;
int my_id;
int leader_states[N_NODES];
int leader_timeout_count;
int debug;
int message_count;
int last_rec_con_type; // last received connection type
    // LDR_CONN or MBR_CONN
int next_coordinator;
int last_send_message_type;
hdr_mergclus last_recv_message;
hdr_mergclus last_send_message;
Application* upper_app_;    
Application* ring_app_;    
MCFQueue* fque; // Reference to Queue
int pktsize_;  // Packet size
int running_;   // If 1 clustering app. is running
MCTimer mc_timer_;  // Timer reference
protected:
int command(int argc, const char*const* argv); // Otcl command
    // linkage

};

Figure A.8. MergClusApp class header, cont...


```c
#include "packet.h"
#include "app.h"
#include "mc-ring.h"
#include"mobilenode.h"

class MCRingApp : public Application {
    public:
        MCRingApp(); // Constructor
        void init(); // initializes the variables
        // Calculates distance between X1,Y1 and X2,Y2
double calculateDistance(double X1,double X2,double Y1,double Y2);
        // Forms the ring by calling other procedures
        void formRing();
        void constructRingFromGraph();
        void createGraphFromLeaders();
        // Streams the Leader_Info message
        void broadcastLeader(int real_source,double X,double Y,int cluster_leader);
        // Receives message
        void recv_msg(int nbytes, const char *msg);
        // Finds the MST by Prim's Algorithm
        void primmst(int start);
        void printMst(); // Prints MST
        // Finds the start leader for mst construction
        int findStartLeaderForMst();
        // Finds the initial backbone leader for backbone connection
        int findInitialBackboneLeaderForMst();
        // Finds starting backbone node
        int findStartingBackboneNode();
        // Finds the Next Backbone Node
        int findNextBackboneNode(int start);
        // Finds the next LEAF in same Parent Backbone Node
        int getNextLeafInSameParent(int leaf);
        // Finds the Previous Backbone Node
        int getBackBackbone(int coordinator);
        // Finds the Previous Backbone Node's Leaf node
        int getBackBackboneLeaf(int leaf);
```

Figure A.9. MCRingApp class header
int getAvailableLeafInBackbone(int backbone);

int findNextLeaf(int leaf);

int findSmallestLeafofBackbone(int start);

void findDegreeofNodes();

void identifyNodes();

void connectBackboneNodes(int startingBackboneNode);

void connectLeafNodes(int startingLeaf, int parent, int startingBackbone);

int getParentofLeaf(int leaf);

hdr_mergclus header;
Application *mcapp_; Application reference
graph_t graph_st; // Graph instance
hdr_mergclus leaderInfo[N_NODES];

protected:
    int command(int argc, const char*const* argv); // Otcl command
    void start(); // To start and stop
    void stop();

private:
    int running_; // If 1 application is running
};
class MCDMApp;

class MCDMApp : public Application {
public:
MCDMApp(); // Constructor
// Receive and send messages
virtual void recv_msg(int nbytes, const char *msg = 0);
void send_message(int source, int destination, int message_type,
int source_of_req, int resource_id, double timestamp,
double ring_timestamp, int coordinator_id);
void prepare_msg(hdr_mergclus *mc,int source, int destination,
int message_type, int source_of_req, int resource_id,
double timestamp,int coordinator_id, double ring_timestamp);

// Functions to communicate with upper and lower layers
void notification();
virtual void upper_notification();
void actNA(void *message);

// Finite State Machine Functions
void fsm_jump_next_state(void *message);
void dm_any_state_tout();
void dm_any_state_mut_req(void *buffer);
void dm_any_state_coord_rep(void *buffer);
void dm_idle_coord_req(void *buffer);
void dm_idle_node_req(void *buffer);
void dm_waitrp_coord_req(void *buffer);
void dm_waitrp_node_req(void *buffer);
void dm_waitrp_node_rel(void *buffer);
void dm_waitnd_coord_req(void *buffer);
void dm_waitnd_node_req(void *buffer);
void dm_waitnd_node_rel(void *buffer);
void mutex_request();

// Mutex execution and release functions
void execute_mutex(int my_id);
void release_mutex(int my_id);

// Utility functions
char * message_to_string(int message_type);
char * state_to_string(int state);
void debug_msg(void *message);

Figure A.11. MCDMAp class header
// Queue Functions
int compare_timestamps(ra_req_ptr req1, ra_req_ptr req2);
int find_empty_index(coord_info_ptr cd);
int find_empty_ack_index(coord_info_ptr cd);
void copy_request(ra_req_ptr req1, ra_req_ptr req2);
int check_queue(tsp_que_ptr qp, ra_req_ptr req);
void tsp_insert_request(coord_info_ptr cd, ra_req_ptr req);
void tsp_insert(tsp_que_ptr qp, ra_req_ptr req);
int compare_tsp_insert(coord_info_ptr cd, ra_req_ptr req1);
void safe_tsp_insert(coord_info_ptr cd, ra_req_ptr req1);
void safe_ack_insert(coord_info_ptr cd, ra_req_ptr req1);
void print_request(ra_req_ptr req);
void print_queue(tsp_que_ptr qp);
void safe_print_queue(tsp_que_ptr qp, int id);
int is_empty_queue(tsp_que_ptr qp);
ra_req_ptr tsp_dequeue_request(coord_info_ptr cd);
ra_req_ptr dequeue_req(tsp_que_ptr qp);
int remove_req_queue(tsp_que_ptr qp, int source, int timestamp);
void lock_queue(tsp_que_ptr qp);
void unlock_queue(tsp_que_ptr qp);
void insert_or_forward_request(void *buffer);
void tsp_insert_ack(coord_info_ptr cd, ra_req_ptr req);
ra_req_ptr tsp_dequeue_ack(coord_info_ptr cd);
int get_request_mutex_number(tsp_que_ptr qp);

// Protocol Variables
MergClusApp *mcapp_; // MergClusApp reference
int global_error; // MergClusApp reference
coord_info_t my_info; // coordinator info

// Mutex variables
int mutex_flag;
double mutex_tout_seconds;
double last_mutex_req_seconds;

global_error;
coord_info_t my_info; // coordinator info

// Mutex variables
int mutex_flag;
double mutex_tout_seconds;
double last_mutex_req_seconds;

global_error;
coord_info_t my_info; // coordinator info

protected:
int command(int argc, const char*const* argv); // 0tcl command

void start(); // Start sending packets
void stop(); // Stop sending packets

private:

void init();
void init_fsm();
int running_; // If 1 application is running
MCDMTimer mcdm_timer_; // SendTimer

};

Figure A.12. MCDMAp class header, cont...
# sim.tcl # Simulation of MCA, BFA, Mobile_RA #

======================================================================

# Define options #

======================================================================

set val(chan) Channel/WirelessChannel ;# channel type
set val(prop) Propagation/TwoRayGround ;# radio-propagation model
set val(netif) Phy/WirelessPhy ;# network interface type
set val(mac) Mac/802_11 ;# MAC type
set val(ifq) Queue/DropTail/PriQueue ;# interface queue type
set val(ll) LL ;# link layer type
set val(ant) Antenna/OmniAntenna ;# antenna model
set val(ifqlen) 500 ;# max packet in ifq
set val(nn) 10 ;# number of mobilenodes
set val(rp) DumbAgent ;# routing protocol

# movement file name format: {"mcmp"+number of nodes+area type
#(Very Small:dt,Small:mt,Medium:lt)+scenario number}

set val(mp) ../../ns-2.28/indep-utils/cmu-scen-gen/setdest/mcmp10mt1

# load file name format: {"mrequest"+number of nodes+load type
#(High:H,Medium:M,Low:L)+scenario number}

set val(mrequest) ../mrequests/mrequest10M1

set val(start_time) #start time of simulation
set val(mmapp_end_time) 3.00 #Clustering algorithm end time
set val(mcring_start_time) 3.01 #Backbone formation start time
set val(mcring_end_time) 3.99 #Backbone formation end time
set val(mcdm_start_time) 4.00 #Mobile_RA start time
set val(mcdm_end_time) 6.99 #Mobile_RA end time

# Period Timeout is equal to the Mobile_RA end time
set val(end_time) 7.00 #Simulation end time(Status is informed)
set val(halt_time) 7.01 #Simulation halt time
set val(tr_file) out10mt1.tr #Trace file

# Main Program #

======================================================================

Figure A.13. Complete Scenario File
# Create the specified number of mobilenodes [$val(nn)$] and
# "attach" them # to the channel.
# Here all the nodes
# configure channel
set chan_1_ [new $val(chan)]
$ns_ node-config -adhocRouting $val(rp) -llType $val(ll) -macType $val(mac) -ifqType $val(ifq) -ifqLen $val(ifqlen) -antType $val(ant) -propType $val(prop) -phyType $val(netif) -channel $chan_1_ -topoInstance $topo -agentTrace OFF -routerTrace OFF -macTrace OFF -movementTrace ON

# Create nodes and attach them to new implementation of UDP
#which supports Merging Clustering
for {set i 0} {$i < $val(nn) } {incr i} {
    set node_($i) [$ns_ node]
    $node_($i) set X_ [expr ($i+1)]
    $node_($i) set Y_ [expr ($i+1)]
    $node_($i) set Z_ 0
    $node_($i) random-motion 1 ;# enable random motion
    set udp_($i) [new Agent/UDP/UDPmergclus]
    $ns_ attach-agent $node_($i) $udp_($i)
    $udp_($i) set packetSize_ 1000
    $udp_($i) set fid_ 1
}

# set mcring [new Application/MCRingApp]
for {set i 0} {$i < $val(nn) } {incr i} {
    set mmapp_($i) [new Application/MergClusApp]
    set mcdmapp_($i) [new Application/MCDMApp]
    set mcring_($i) [new Application/MCRingApp]
    $mmapp_($i) attach-agent $udp_($i)
}

for {set i 0} {$i < $val(nn) } {incr i} {
    $mcdmapp_($i) attach-app $mmapp_($i)
    $mcring_($i) attach-app $mmapp_($i)
    $mcring_($i) record $node_($i)
}

for {set i 0} {$i < $val(nn) } {incr i} {
    for {set j 0} {$j < $val(nn) } {incr j} {
        $mmapp_($i) record $node_($j)
    }
}

Figure A.14. Complete Scenario File, cont...
puts "Loading movement pattern..."
source $val(mp)
puts "Loading Mutex Request Pattern..."
source $val(mrequest)
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(start_time) "$mmapp_($i) start"
    $ns_ at $val(mmapp_end_time) "$mmapp_($i) stop"
    $ns_ at $val(mcring_start_time) "$mcring_($i) start"
    $ns_ at $val(mcring_end_time) "$mcring_($i) stop"
    $ns_ at $val(mcdm_start_time) "$mcdmapp_($i) start"
    $ns_ at $val(mcdm_end_time) "$mcdmapp_($i) stop"
}
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(mmapp_end_time) "$node_($i) log-movement"
}
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(end_time) "$node_($i) reset"
}
$ns_ at $val(end_time) "stop"
global ns_tracefd
$ns_ flush-trace
close $tracefd
proc stop {} {
    $ns_ halt
}
## Run the simulation #
$ns_ run

Figure A.15. Complete Scenario File, cont...