TABLE OF CONTENTS

LIST OF FIGURES 5

CHAPTER 1 . INTRODUCTION 5

CHAPTER 2 . BACKGROUND 9

2.1. Clustering Algorithms 10

2.1.1. Dominating Set Based Clustering Algorithms 10

2.1.1.1. Clustering Using IDS 10

2.1.1.2. Clustering Using WCDS 11

2.1.1.3. Clustering Using CDS 12

2.1.2. Spanning Tree Based Algorithms 14

2.2. Routing Algorithms in MANETs 17

2.2.1. Proactive Routing 17

2.2.2. Reactive Routing 17

2.2.3. Hybrid Routing 18

2.2.4. Cluster Based Routing 18

CHAPTER 3 . THE CONNECTED DOMINATING SET BASED CLUSTERING ALGORITHM 20

3.1. General Idea and Description of the Algorithm 20

3.2. An Example Operation 26

3.3. Analysis 27

3.4. Results 28

3.5. Discussions 35

CHAPTER 4 . TWO-LEVEL CONNECTED DOMINATING SET BASED CLUSTERING ALGORITHM 37

4.1. General Idea and Description of the Algorithm 37

4.2. An Example Operation 43

4.3. Analysis 45

4.4. Results 46
4.5. Discussions 51

CHAPTER 5 . CDS FLOODING ALGORITHM 53
5.1. General Idea and Description of the Algorithm 53
5.1.1. Sending a Message 59
5.1.2. Receiving and Processing Messages 60
5.2. An Example Operation 61
5.3. Analysis 63
5.4. Results 65
5.5. Discussions 66

CHAPTER 6 . CONCLUSION 68

APPENDIX A. SIMULATION SETUP 76

LIST OF FIGURES

Figure		Page
Figure 1.1	Hierarchical Representation of the Proposed Algorithms	8
Figure 2.1	(a)IDS (b)WCDS (c)CDS	10
Figure 3.1	Finite State Machine of the Dominating Set Based Clustering Algorithm	22
Figure 3.2	Example after the first phase	26
Figure 3.3	Example resulting graph	27
Figure 3.4	Runtime Test in a Static Network	29
Figure 3.5	Runtime Test in a Low Speed Dynamic Network	29
Figure 3.6	Runtime Test in a High Speed Dynamic Network	30
Figure 3.7	Number of Clusterheads in a Static Network	30
Figure 3.8	Number of Clusterheads in a Low Speed Dynamic Network	31
Figure 3.9	Number of Clusterheads in a High Speed Dynamic Network	31
Figure 3.10	Cluster Size Test in a Static Network	32
Figure 3.11	Cluster Size Test in a Low Speed Dynamic Network	32
Figure 3.12	Cluster Size Test in a High Speed Dynamic Network	32
Figure 3.13	Cluster Quality Test in a Static Network	33
Figure 3.14	Cluster Quality Test in a Low Speed Dynamic Network	33
Figure 3.15	Cluster Quality Test in a High Speed Dynamic Network	34
Figure 3.16	Average Number of Clusterheads Test in a Static, Moving in Low Speed and Moving in High Speed Dynamic Network	34
Figure 3.17	Comparison Between CDSC and Wu's CDS Algorithms in terms of Number of Clusterheads	35
Figure 4.1	Finite State Machine of the TLCDSC Algorithm	38
Figure 4.2	TLCDSC Algorithm Sample Graph	44
Figure 4.3	TLCDSC Algorithm Samle Graph at the End of First Level	44
Figure 4.4	TLCDSC Algorithm Resulting Graph	45
Figure 4.5	TLCDSC Algorithm Runtime Test in a Static Network	47

Figure 4.6 TLCDSC Algorithm Runtime Test in a Low Speed Dynamic Network . . 47
Figure 4.7 TLCDSC Algorithm Runtime Test in a High Speed Dynamic Network . . 48
Figure 4.8 Number of Super Cluster Heads in a Static Network 48
Figure 4.9 Number of Super Cluster Heads in a Low Speed Dynamic Network 49
Figure 4.10 Number of Super Cluster Heads in a High Speed Dynamic Network . . . 49
Figure 4.11 Size of the Super Clusters in a Static Network 50
Figure 4.12 Size of the Super Clusters in a Low Speed Dynamic Network 50
Figure 4.13 Size of the Super Clusters in a High Speed Dynamic Network 51

Figure 5.1 Finite State Machine of the 2-Level Hierarchical Clustering Algorithm . . 55
Figure 5.2 Data Structure for the MessageInfoBox 58
Figure 5.3 An Example CDS Based Clusters 62
Figure 5.4 CDS Flooding Examples 63
Figure 5.5 CDS Flooding Examples 64
Figure 5.6 CDS Flooding Test Results for Static Network 66
Figure 5.7 CDS Flooding Test Results for Low Speed Dynamic Network 67

Figure A.1 Header File of the udp-cds Class 78
Figure A.2 Header File of the udp-cds Class 79
Figure A.3 Modification Made on mac-802_11.cc 79
Figure A.4 CDSClusApp Class Header File 80
Figure A.5 CDSClusApp Class Header File (con.) 81
Figure A.6 CDSClusApp Class Header File (con.) 82
Figure A.7 TLCDSCApp Class Header File 83
Figure A.8 TLCDSCApp Class Header File (con.) 84
Figure A.9 TLCDSCApp Class Header File (con.) 85
Figure A.10 TLCDSCApp Class Header File 86
Figure A.11 CDSFloodingApp Class Header File 87
Figure A.12 CDSFloodingApp Class Header File (con.) 88
Figure A.13 CDSFloodingApp Class Header File (con.) 89
Figure A.14 An example Scenario File 90
Figure A.15 An example Scenario File (con.) 91
Figure A.16 An example Scenario File (con.) 92

CHAPTER 1

INTRODUCTION

Wireless communication technology has becoming an essential framework for the today's communication environments. Being freed of the mess of wires, and mobility issues attracts people to build wireless communication platforms everywhere. This interest leads the researchers to improve the current wireless technology. Most of the today's wireless devices are, in some sense, limited by their need for infrastructure. Today's most popular approach for the non-fixed infrastructure is the Ad hoc networking which can be considered as a solution to the infrastructure limitations of the wireless technology. Mobile Ad hoc Networks (MANETs) do not have any fixed infrastructure and consist of wireless mobile nodes that perform various data communication tasks in an Ad hoc manner. Generally in a MANET, the nodes are dynamically moving without any boundary limitations. Therefore, the nodes must have some key specifications in order to provide continuous communication. MANETs have potential applications in rescue operations, battlefield communications, mobile conferences etc. Although the primary application area of the MANETs is considered to be the military and rescue operations, the tendency of the communication community shows that commercial and educational use of MANETs is growing rapidly. The law enforcement operations, sensor networks, personal area networking, media distribution applications are such examples of the commercial and educational use of the MANETs. Although MANETs have numerous advantages over the wired and structured mobile networks, they have some issues to be addressed such as topology changes, bandwidth optimization, locality of information, ad hoc addressing, energy conservation, self routing, self organization and self routing. Most of these problems are caused by the common wireless transport layer and the awareness of the nodes. A generic structured ad hoc topology eases most of those problems.

A very popular approach for structuring MANETs is clustering the network. In a typical clustering scheme, the MANET is firstly partitioned into a number of clusters by a suitable distributed algorithm according to some common attributes of the nodes such as degrees, power levels, geographical locations etc. In a clustered structure, some nodes may be assigned with

some special functionalities in order to perform different operations. The nodes in a clustered MANET may be clusterheads, cluster gateways or cluster members. A clusterhead can be used in many roles such as management of the cluster members, routing of messages or inter cluster communications. A cluster gateway is used to establish connection between different clusters. A cluster gateway can be both a clusterhead or a cluster member. If the clusterheads in the entire network build a backbone, the cluster gateways are all clusterheads. A cluster member is called an ordinary node generally and is not loaded by any special functionalities. An ordinary node does not involve in inter-cluster communication or does not have to know about routing of messages. There are many advantages of clustering in MANETs. By clustering the network, one can build small sized clusters with clusterheads and cluster gateways forming a backbone which may be very useful in many applications. Clusters can be used in order to develop efficient routing protocols which is very necessary in MANETs. The clusters may also be used in order to perform different operations in different clusters which resides in the same environment. As the clusterheads play cluster manager roles as well, they can easily determine the operation which is to be executed within their own cluster.

There are many research studies in the field of clustering MANETs, especially in clustering using graph theoretical algorithms such as dominating set based algorithms. MANETs can easily be modeled by using graph theoretical concepts. The mobile nodes in the MANETs are modeled by the vertices, and the communication links are modeled by the edges in the graphs. Once the MANETs are mapped to the graphs, the graph theoretical algorithms can be implemented on them in order to select some special nodes with some common properties. This leads us to build clusters around the selected special nodes. The special nodes are called the clusterheads and their adjacent neighbors are called the cluster members. In this thesis, we focused on the dominating sets as the graph theoretical concept which is used to model the MANETs. A dominating set is a subset of the vertices of a graph if every vertex not in the subset is adjacent to at least one vertex in the subset. The advantage of using the dominating set concepts is that they do not have the need of a backbone formation algorithm in some special cases such as the models in which the connected dominating sets (CDS) are used. After clustering the network, a backbone which is consisting of clusterheads is already created in CDSs, because all clusterheads are directly connected to each other in a CDS. The main disadvantage of building CDSs is their cost. Therefore the primary aim of the researchers is to minimize the cost of the algorithms. Another important problem of building connected dominating set based

clusters is the large number of clusterheads. In clustered environments the clusters have to be in an optimized size in terms of number of cluster members. The ideal size depends on the application which is to be executed within the clusters. For instance, if the application requires a large number of message exchange between the cluster members, less crowded clusters are needed in order to minimize the message conflicts caused by the use of wireless communication. But the small cluster sizes increase the messaging between the clusterheads which may slow down the inter cluster communication. Therefore, optimizing the cluster sizes is an important issue in clustering MANETs.

The most commonly used applications which is built on top of clustered MANETs are the routing applications. Routing in MANETs is a very problematic issue because of the dynamicity of the network. In dynamic networks, routing tables should be updated very frequently. Keeping the routing tables up to date may consume a large part of the wireless traffic in the network. This traffic might sometimes be extremely dense which may possibly block the circulation of the messages between nodes. A virtually structured network such as a clustered MANET can be considered as a good solution to make message transfers more efficient. In a clustered scheme, the clusterheads are allocated to manage inter and intra-cluster routing. Moreover, in a connected dominating set based clustered MANET, the clusterheads form a backbone which is also used as the gateways of the clusters. There are several categories of routing protocols in ad hoc wireless networks. The most well known types are proactive, reactive, hybrid and cluster based routing. In proactive routing, routes to all destinations are computed a priori and are maintained in the background via a periodic update process. In reactive routing, route to a specific destination is computed only when needed. Hybrid routing protocols combine the advantages of both reactive and proactive routing protocols. To efficiently use resources in controlling large dynamic networks, cluster based routing is generally used. Cluster based routing protocols uses structured network topologies such as connected dominating set based clustered networks. The advantage of using such a structured topology is that it simplifies the routing process to the one in a smaller subnetwork generated from the connected dominating set. This means that only gateway hosts need to keep routing information in a proactive approach and the search space is reduced to the dominating set in a reactive approach "(Wu 2002)".

The aim of this thesis is to analyze, design and implement a communication architecture which consists of a hierarchical dominating set based clustering algorithm and a flooding based

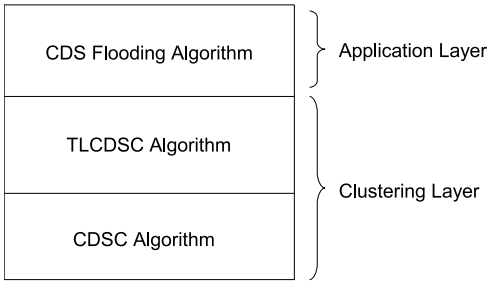

Figure 1.1. Hierarchical Representation of the Proposed Algorithms

routing algorithm which is designed to run on top of the resulting CDS based clusters. A brief hierarchical representation of the proposed algorithms can be seen in Fig. 1.1. We search various graph theoretic algorithms for clustering in MANETs and propose two new distributed connected dominating set based algorithms. Firstly, we propose the Connected Dominating Set Based Clustering (CDSC) Algorithm "(Cokuslu et al. 2006)" which finds a minimal connected dominating set in a MANET. We developed our algorithm based on Wu's CDS Algorithm "(Wu and Li 2002)" but we add some extra heuristics. We determine some situations that a node cannot change its color after the first phase. We also consider the degree of a node when marking it. We then improve the CDSC Algorithm and propose the Two-Level Connected Dominating Set Based Clustering (TLCDSC) Algorithm by applying the CDSC Algorithm recursively. We construct a CDS using the basic principles of the CDSC Algorithm and then run the same basics on the resulting CDS in order to find second level clusterheads. This approach provides more crowded clusters which are more preferable than the CDSC Algorithm in which the size of the clusters are very small compared to the number of nodes in the MANET. Third, we analyze, design and implement the CDS Flooding Algorithm which is a hierarchical routing algorithm running on top of the CDS clusters. Hierarchical routing decreases the inter-cluster message traffic and provides an efficient flooding based routing protocol.

We define dominating set and minimum spanning tree concepts and related clustering algorithms in Chapter 2. We describe, analyze and show the test results for the CDSC Algorithm in Chapter 3. The TLCDSC Algorithm is explained in Chapter 4 and CDS Flooding Algorithm is described in Chapter 5. Finally the conclusion which concludes the three algorithms is explained in Chapter 6.

CHAPTER 2

BACKGROUND

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 for clustering in networks "(Chen and Liestman 2002)". Dominating sets can be classified into three main classes, Connected Dominating Sets (CDS), Weakly Connected Dominating Sets (WCDS) and Independent Dominating Sets (IDS) "(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. 2.1.a shows a sample independent dominating set where black nodes show clusterheads.

- *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. 2.1.b 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. 2.1.c shows a sample CDS.

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) of a graph is a subgraph that has the minimum number of edges for maintaining connectivity "(Grimaldi 1997)".

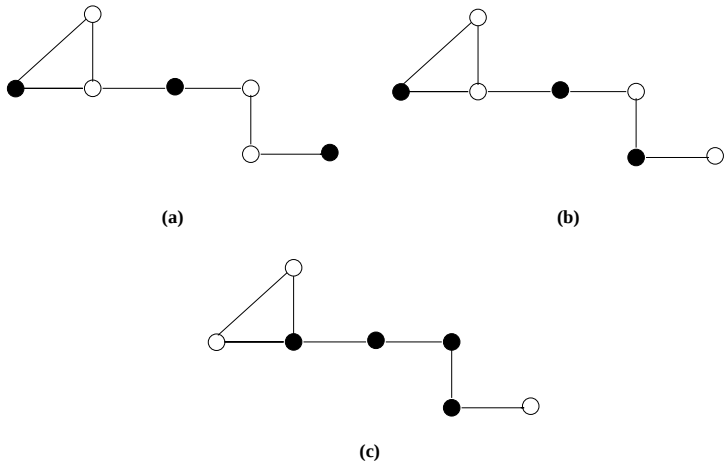

Figure 2.1. (a)IDS (b)WCDS (c)CDS

2.1. Clustering Algorithms

Dominating sets and spanning trees are widely used to cluster MANETs, since MANETs can easily be modeled by graphs where wireless nodes are mapped to vertices and communication links between the nodes are mapped to edges. Several graph theoretical clustering algorithms exist in the literature but in this thesis, we focused on the dominating set and spanning tree based clustering algorithms.

2.1.1. Dominating Set Based Clustering Algorithms

Various algorithms exist for clustering in IDS, WCDS and CDS. The nodes in the dominating sets are called clusterheads which can be loaded by several special functionalities such as gateways, backbone members, routers etc. The dominating set based clustering algorithms are examined in three classes which are explained in more detail below.

2.1.1.1. Clustering Using IDS

By using independent dominating sets, one can guarantee that there are no adjacent clusterheads in the entire graph. This minimizes the number of dummy clusters in the network and results in more crowded clusters which may be preferable in some situations.

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 clusterhead. If a node does not have

the highest id, then it does not have to mark itself as a clusterhead. However, such a node must also check whether it is the highest id neighbor of some other node. This can be done by checking the connectivity information of the node's lower numbered neighbors. If the node i is the highest id neighbor of a node j, node i must become a cluster head for at least node j . This guarantees that the selected set of nodes constructs a dominating set but does not guarantee a CDS or a WCDS. A very similar algorithm to the highest 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 a clusterhead. 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 clusterhead.

Although these algorithms are considered as important algorithms, Chen et al. "(Chen et al. 2002)" showed that these algorithms do not work 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 adds one more rule to the above algorithms such that in a k-distance dominating set, every clusterhead must be at least k+1 distant from each other "(Ohta et al. 2003)".

2.1.1.2. Clustering Using WCDS

Although independent dominating sets are suitable for constructing optimum sized dominating sets, they have some deficiencies such as the lack of direct communication between clusterheads. In order to obtain connectivity between clusterheads, WCDSs can be used to construct clusters. The WCDS was first proposed for clustering in ad hoc networks by Chen and Liestman "(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 "(Guha and Khuller 1998)" which is described next. 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*"(Chen and Liestman 2003)".

2.1.1.3. Clustering Using CDS

CDSs have many advantages in network applications such as ease of broadcasting and constructing virtual backbones "(Stojmenovic et al. 2002)" because, the clusterheads in CDS construct a backbone which is also used as the gateways of the clusters. However, when we try to obtain a connected dominating set, we may have undesirable number of clusterheads. So, in constructing connected dominating sets, the primary problem is to construct a minimal connected dominating set.

Guha and Khuller "(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 a node with the maximum number of white neighbors as a dominating node. The dominating node becomes black, and its neighbors become grey. Then the algorithm iteratively scans the grey nodes and their white neighbors. In each iteration, the grey 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. Das and Bharghavan "(Das and Bharghavan 1997)" "(Das et al. 1997)" provided distributed implementations of Ghua and Khuller's algorithms.

Wu and Li "(Wu and Li 2002)", improved Das and Bhraghavan's distributed algorithm "(Das and Bharghavan 1997)" as a localized distributed algorithm for finding connected distributed sets in which each node only needs to know its distance-two neighbor. In Wu and Li's algorithm, 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 pruning rules 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 $N(v) \subseteq N(u) \bigcup N(w)$ and $id(v) = min\{id(v), id(u), id(w)\}$;

Dai and Wu "(Dai and Wu 2004)" 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 showed that these assumptions are not always true. They proposed a generalized dominant pruning rule, called Rule k, which can unmark cluster heads covered by k other cluster heads, where k can be any number. According to this algorithm, the first phase works same as Wu and Li's algorithm, but in phase two, Rule k pruning rule is applied to eliminate dummy clusterheads instead of Wu and Li's two pruning rules. According to Rule k, if neighbors of a clusterhead is dominated by more than two directly connected clusterheads, 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 Wu and Li's pruning rule 1 and, less complexity than Wu and Li's pruning rule 2.

Xinfang Yan, Yugeng Sun, and Yanlin Wang "(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 in selecting the clusterheads. By using this heuristic, Yan et al. make a better estimation on the stability of the backbone topology.

Peng-Jun Wan, Khaled M. Alzoubi and Ophir Frieder "(Wan et al. 2004)" propsed 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.

Hui Liu, Yi Pan and Jiannong Cao "(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 clusterheads. A clusterhead is eliminated if it is dominated by two of its clusterhead neighbors. The distributed algorithm has a time complexity $O(n^2)$ and a message complexity $O(n)$.

2.1.2. 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. The 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 then 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 $O(E+N\log_2 N)$ "(Gallagher et al. 1983)".

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 (GHS) "(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 GHS

algorithm, and terminates when all trees reach the size of $\Omega(V/logV)$. 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 + Vlog V)* messages and *O(V)* time.

The algorithms proposed by GHS "(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 GHS algorithm. 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 where e is the total number of edges and n is the number of nodes in the graph. In the best case, it needs only *2e* messages in *O(nlogn)* where e is the number of edges and n is the number of nodes.

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 GHS Algorithm. 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 $nodeID$s 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 $O(logn)$ phases where n is the number of nodes. In the worst case, the algorithm needs at most $(2m + 2(n-1)log(n/2))$ messages and $(2dlogn)$ time, where d is the diameter of the graph, m is the total number of edges and n is the number of nodes in the graph. 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, Kutten and Peleg "(Garay et al. 1993)" provide a modified, controlled version the GHA algorithm. The algorithm is able to achieve the following:

- Upon termination, the number of the fragments is bounded above by $n \ / \ 2^{numberofphases}$.
- Throughout the execution, the diameter of every fragment F satisfies *Diameter(Fragment)* $< 3^{numberofphases}$.

The time complexity of the algorithm is $O(Diam(G)+n^{0,614})$.

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 α *cone* as the origin concerned node and bounded by two rays with an angle α 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.

"(Dagdeviren et al. 2006)" proposed the Merging Clustering Algorithm(MCA) which finds clusters in a MANET by merging the clusters to form higher level clusters as mentioned in GHS algorithm. However, Dagdeviren et al. focus on the clustering operation by discarding minimum spanning tree. This reduces the message complexity. They also use upper and lower bound heuristics for clustering operation which results balanced number of nodes in the resulting clusters.

2.2. Routing Algorithms in MANETs

Because of the importance of routing protocols in MANETs, many routing protocols have been proposed in the last few years for MANETs. Although the proposed routing schemes are successful in many cases, there is no one-for-all routing protocol which works well in all scenarios with different network sizes, traffic overloads and node mobilities. Moreover, those protocols are based on different approaches and are proposed to meet specific requirements from different applications. Generally, a well designed MANET routing protocol should adapt to the dynamic network changes quickly with lower consumption of communication and computing resources. We only mention the most recent ones out of many routing protocols here. Although the routing protocols for MANETs can be classified basically as Proactive, Reactive, Hybrid and Cluster Based Routing protocols " (Liu and Kaiser 2005)", many other proposed schemes exist which are not suitable for none of these classifications. In this chapter we mainly focus on the Cluster Based Routing protocols.

2.2.1. Proactive Routing

In proactive routing schemes, the routing tables are prepared priori and updated as the topology changes occur. Using a proactive routing protocol, nodes in a MANET continuously evaluate routes to all reachable nodes. Therefore, the route to a destination node is always ready to be used when needed. In proactive routing protocols, all nodes need to maintain an updated view of the whole network topology. When a topology change occurs, updates must be propagated throughout the network to distribute the change. Most proactive routing protocols proposed for mobile ad hoc networks are influenced from the routing protocols for wired networks. Wireless Routing Protocol " (Murthy and Garcia-Luna-Aceves 1996)", Destination Sequence Distance Vector (DSDV) Routing protocol " (Perkins and Bhagwat 1994)", Fisheye State Routing " (Pei et al. 2000)" and Distance Routing Effect Algorithm for Mobility " (Basagni et al. 1998)" are well known proactive routing protocols.

2.2.2. Reactive Routing

Reactive routing protocols are routing schemes in which routes are searched only when needed. When a node wants to send a message, a route discovery operation is started through

the destination node. The discovery procedure is terminated either when a route has been found or no route is available after trial for all route possibilities. Compared to the proactive routing protocols, less control overhead is a distinct advantage of the reactive routing protocols. Thus, reactive routing protocols have better scalability than proactive routing protocols in MANETs. However, when using reactive routing protocols, message transfer times may be longer due to the on-demand routing determination process. The Dynamic Source Routing " (Johnson and Maltz 1996)" and Ad hoc On-demand Distance Vector (AODV) routing " (Perkins and Royer 1999)" are examples for reactive routing protocols.

2.2.3. Hybrid Routing

Hybrid routing protocols combine the advantages of proactive and reactive routing protocols and minimize their shortcomings. Generally, hybrid routing protocols uses the benefits of the hierarchical network architectures. Proper proactive and reactive routing approaches are utilized in different hierarchical levels. A good example of hybrid routing protocols for MANETs is Zone-based Hierarchical Link State routing " (Joa-Ng and Lu 1999)".

2.2.4. Cluster Based Routing

The cluster based routing protocols use specific clustering algorithms for clustering in MANET. The cluster based routing protocols generally assume that the clusters, memberships and clusterheads are already determined. In the cluster based routing schemes, mobile nodes are grouped into clusters and clusterheads take the responsibility for membership management and routing functions. Some cluster based routing protocols potentially support a multi-level cluster structure.

Clusterhead Gateway Switch Routing " (Chiang et al. 1997)" (CGSR) is an example of cluster based routing scheme. In CGSR, gateway nodes are responsible for communication between the clusterheads. Nodes maintain a cluster member table which maps each node to its respective clusterhead. A node broadcasts its cluster member table periodically. After receiving broadcasts from other nodes, a node updates its cluster member table. In addition, each node maintains a routing table that determines the next hop to reach other clusters.

The Hierarchical State Routing (HSR) " (Pei et al. 1999)" is a multi-level cluster-based hierarchical routing protocol. In HSR, the clusterheads of low level clusters organize themselves

into upper level clusters. The nodes broadcast their link state information to all others within their clusters. The clusterhead collects link state information of its cluster and sends the information to its neighboring clusterheads via gateway nodes. Nodes in upper level hierarchical clusters flood the network topology information.

The Cluster Based Routing Protocol (CBRP) " (Mingliang et al. 1999)" is a routing protocol for MANETs in which every node maintains a neighbor table filled with the information about link states of its neighbors. The clusterheads keep the connectivity information of their clusters and their neighboring clusters. If a source node wants to send a packet but has no active route which can be used, it floods route request to clusterhead of its own and all neighboring clusters. If a clusterhead receives a request it has already received before, it discards the request. Otherwise, the clusterhead checks if the destination of the request is in its cluster. If the destination is in the same cluster, the clusterhead sends the request to the destination, or it floods the request to its neighboring clusterheads. At the end of the flooding operation, the destination node sends a reply including the route information recorded in the request. When the source node receives the reply message, it sends its message to the destination by using the route information which is collected by the route request.

Denko and Lu " (Denko and Lu 2006)" proposed an adaptive cluster based routing architecture which is based on an extended AODV " (Perkins and Royer 1999)" routing protocol. By using AODV route construction and maintenance mechanisms, clustering architecture can be constructed on demand. Clusters are maintained when data are to be sent. Denko and Lu used a clustering algorithm based on Chiang et al's " (Chiang et al. 1997)" Lowest ID (LID) clustering algorithm. Firstly, they built two level clusters, then on top of the clusters they implemented the AODV Based Routing Algorithm. The clusters are maintained only when a message is needed to be sent. They implemented their routing protocol in two groups which are the Intra-cluster Routing and Inter-Cluster Routing. Intra-cluster routing involves routing within a cluster. Each node maintains routing information about its cluster. When a route request fails to find a route for a message, the routes are maintained within the cluster. Inter-cluster routing involves routing between clusters. The clusterheads keep the connectivity information about 2-hop cluster topology. Inter-cluster communication is based on AODV routing again, but only clusterheads are involved in the routing of inter-cluster messages.

CHAPTER 3

THE CONNECTED DOMINATING SET BASED CLUSTERING ALGORITHM

3.1. General Idea and Description of the Algorithm

The Connected Dominating Set Based Clustering Algorithm (CDSC) "(Cokuslu et al. 2006)" finds a minimal connected dominating set in a MANET in a distributed manner. We developed our algorithm based on Wu's CDS Algorithm "(Wu and Li 2002)". Wu's CDS algorithm finds a connected dominating set which can be used as a backbone, it is totally distributed and it does not require a predefined routing mechanism. It also gives us a good starting point which can be easily modified. On the other hand, most of the other CDS algorithms are centralized and some of them such as Wu's Extended CDS algorithm requires a routing mechanism. Therefore, we determined our base algorithm as Wu's CDS algorithm which can be enhanced in many ways in order to work better. Using the basic principles of the Wu's CDS Algorithm, we built our CDSC Algorithm. As the basic enhancements, we determined some situations that a node cannot change its color after the first phase. We also consider the degree of a node when marking it. This is due to the fact that a node with a higher degree should have a better chance of being in CDS as it has more neighbors than a node with a lower degree. We also added a third color as a transition color during the pruning phase. We also modified the algorithm so that every node will have a clusterhead at the end of the CDSC algorithm. The assumptions which are required by the CDSC Algorithm are:

- The neighborhoods remain constant in a reasonable period of time in order to complete the algorithm in a single node.
- The graph is connected, each node has a unique *node_id*.
- Every node knows its adjacent neighbors by polling them using beacon messages or collecting their positions from a GPS or by using any other position determining method.

Each node has a *color* indicating whether the node is in the dominating set or not. The *color* is set to *BLACK* if the node is in the dominating set, or *WHITE* if the node is not in the dominating set. Color *GRAY* is used to indicate that the node is marked after the first phase, but it will change its color after the second phase to either *WHITE* or *BLACK*. The message types used in this algorithm are *Period_TOUT*, *Neighbor_REQ*, *Neighbor_LST*, *Color_REQ*, *Color_RES*, *Cluster_REQ* and *Cluster_RES* which are described below:

- *Period_TOUT*: Every node sends this message internally and periodically in order to trigger the CDSC Algorithm. The period depends on how fast the nodes are moving.
- *Neighbor_REQ*: Any node which starts the execution of the CDSC Algorithm requests a list of distance-2 neighborhood information by broadcasting a *Neighbor_REQ* message in order to determine its color at the first phase of the algorithm.
- *Neighbor_LST*: Any node which receives a *Neighbor_REQ* message, sends a *Neighbor_LST* message informing its adjacent neighbors.
- *Color_REQ*: A node will send a *Color_REQ* message in order to request its neighbors' colors after the first phase.
- *Color_RES*: The *Color_RES* message is used to send the node's first level color after the first phase. It is sent when a *Color_REQ* message is received.
- *Cluster_REQ*: When a node determines its first level color as *WHITE*, it will check its neighbors' colors in order to select a suitable clusterhead. If the node does not have any *BLACK* colored neighbor, then it broadcasts a *Cluster_REQ* message in order to find out which neighbor became a *clusterhead*.
- *Cluster_RES*: A *clusterhead* sends a *Cluster_RES* message whenever it receives a *Cluster_REQ* in order to inform its status, whether it is a clusterhead or not, to the sender of the message.

Every node in the network performs the same local algorithm periodically. The finite state diagram for the algorithm can be seen in Fig. 3.1. During the runtime of the CDSC Algorithm, the conditions which affect the state transitions are described below:

CDSC Algorithm Finite State Machine Transition Conditions:

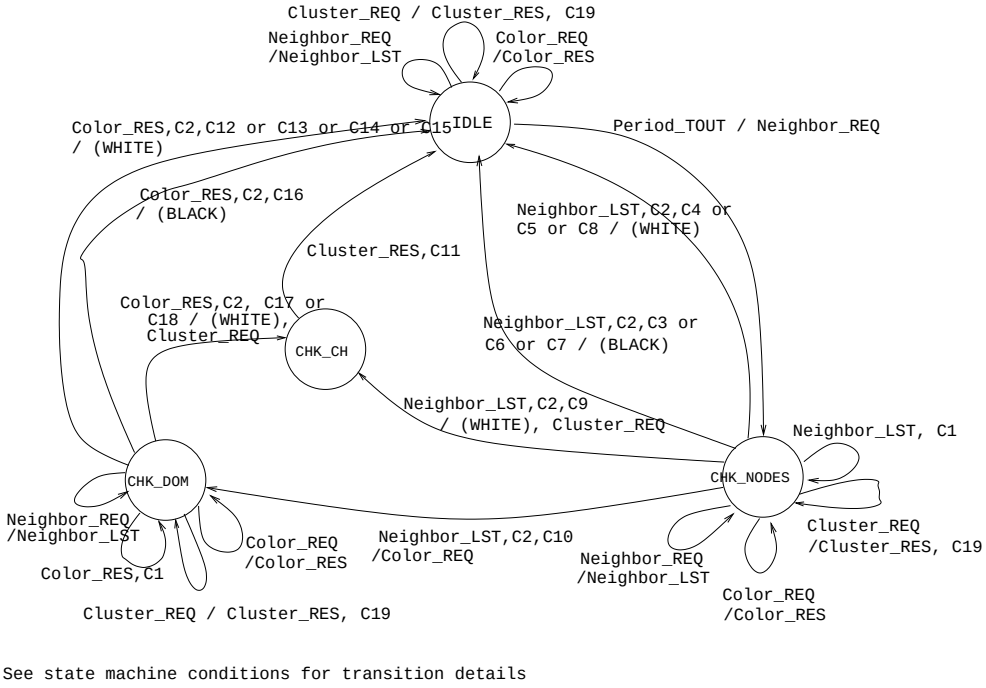

Figure 3.1. Finite State Machine of the Dominating Set Based Clustering Algorithm

- C1. The responses to the multi-casted message are not completely collected.
- C2. The responses to the multi-casted message are completely collected.
- C3. The node is isolated, its neighbor is isolated too and node's id is bigger than its neighbor's id.
- C4. The node is isolated, its neighbor is isolated too and node's id is smaller than its neighbor's id.
- C5. The node is isolated and its neighbor is not isolated.
- C6. The node has at least one isolated neighbor.
- C7. The graph is complete and the node has the biggest id in the graph.
- C8. The graph is complete and the node doesn't have the biggest id.
- C9. Node's neighbors are all connected and the graph is not complete.
- C10. The node has at least two unconnected neighbors.
- C11. Clusterhead is set to the sender's id.

- C12. CDS pruning rule 1 which is described below is true.
- C13. CDS pruning rule 2 which is described below is true.
- C14. CDS pruning rule 3 which is described below is true and the node has at least one *BLACK* neighbor.
- C15. CDS pruning rule 4 which is described below is true and the node has at least one *BLACK* neighbor.
- C16. Conditions C12 to C15 are all false.
- C17. CDS pruning rule 3 is true and the node doesn't have any *BLACK* neighbors.
- C18. CDS pruning rule 4 is true and the node doesn't have any *BLACK* neighbors.
- C19. Nodes color is currently *BLACK*.

Each node is in the *IDLE* state and colored as *UNDEFINED_COLOR* initially. When the period is timed out, the node sends a *Period_TOUT* message to itself. This message causes the node to send a *Neighbor_REQ* message to all of its adjacent neighbors and switch its state to *CHK_NODES*. Then the node waits for *Neighbor_LST* messages from all of its adjacent neighbors. If some response messages are timed out from some of the neighbors, a *Neighbor_REQ* is sent to those neighbors in case a previous one is lost in the network. When all *Neighbor_LST* messages are collected, the node checks the heuristics *C3* to *C9* defined in the state machine transition conditions list to determine if it will be among the ones whose color will not alter after the first phase.

The condition *C3* happens if the graph consists of only two nodes connected to each other or if there are isolated pairs in the graph. In such a case, the node which has a bigger node id marks itself as *BLACK*. If the node has smaller node id than its pair, condition C4 becomes true, and the node marks itself as *WHITE* and selects its pair as its clusterhead.

If the node is an isolated node and its neighbor is connected to other nodes in the graph then the node marks itself as *WHITE* and sets its clusterhead as its neighbor according to heuristic C5.

If a node has at least one isolated neighbor, then according to condition C6, it has to be *BLACK* colored as there are no other alternative nodes which dominate the isolated neighbor.

If all neighbors of the node are directly connected to each other it means that there might be other alternatives to be the clusterhead in the neighborhood. In such a case, we select the node which has the biggest degree in the neighborhood. But we first check if the graph is complete or not. If the graph is complete then the node with the biggest id is selected as the clusterhead by applying the condition C7 and marked as *BLACK*. Other nodes in the complete graph apply the condition C8 and set their color as *WHITE* and their clusterheads as the biggest id in their neighborhood.

The nodes which suit any one of the conditions C3 to C8 switch their state from *CHK_NODES* to *IDLE* as they determine their colors permanently as *BLACK* or *WHITE*.

If condition C9 is true for any node, it means that the node's neighbors are all connected to each other but the graph is not complete. In such a case, the node marks its color as *WHITE* switches its state to *CHK_CH* and multicasts a *Cluster_REQ* message in order to learn which neighbor become its clusterhead. When the node receives a *Cluster_RES* message related to its request, it then sets its clusterhead as the sender of the message and changes its state to *IDLE*. Such a node will ignore the following *Cluster_RES* messages from the other *BLACK* colored neighbors as it has already determined its clusterhead.

If a node has at least two unconnected neighbors, it is suitable for the condition C10 and is potentially a clusterhead candidate. In this case, the node needs to analyze its neighbors if there are other clusterhead candidates which dominate the two unconnected neighbors. Such an analysis requires the neighbors' colors information; the 2-distance neighborhood information collected until now is not adequate. Therefore, the node switches its state to *CHK_DOM*, changes its color to *GRAY* and multicasts a *Color_REQ* message in order to collect its neighbor's colors.

At this point, we can say that the node completed the first phase of the algorithm and starts the second phase. When the node switches its state to *CHK_DOM* it waits for all of its neighbors to send their colors. When the node v collects all color information, it starts to apply the CDS pruning rules which are described below:

CDSC Algorithm Pruning Rules:

1. $\exists u \in N(v)$ which is marked $BLACK$ such that $N[v] \subseteq N[u]$;

   If the node has a *BLACK* neighbor which covers its closed neighborhood, then it should

mark itself as *WHITE* because there is already a *BLACK* node which dominates all closed neighborhood.

2. $\exists u, w \in N(v)$which is marked $BLACK$ such that $N(v) \subseteq N(u) \bigcup N(w)$;
   If the node has two connected *BLACK* neighbors and if the union of the neighborhoods of these *BLACK* neighbors cover the node's closed neighborhood then the node should mark itself as *WHITE*.

3. $\exists u \in N(v)$ which is marked $GRAY$ such that $N[v] \subseteq N[u]$ and $degree(v) < degree(u)$ OR ($degree(v) = degree(u)$ AND $id(v) < id(u)$);
   If the node has a *GRAY* colored neighbor which covers its closed neighborhood, it means both of them are candidates to be in the dominating set. In this case, if the node has a smaller degree than its *GRAY* colored neighbor, it should mark itself as *WHITE*.

4. $\exists u, w \in N(v)$which is marked $GRAY$ or $BLACK$ such that $N(v) \subseteq N(u) \bigcup N(w)$ and $degree(v) < min\{degree(u), degree(w)\}$ OR $degree(v) = min\{degree(u), degree(w)\}$ AND $id(v) < min\{id(u), id(w)\}$;
   If the node has two connected *BLACK* or *GRAY* neighbors which covers its closed neighborhood and if its degree is smaller than these neighbors' degrees, then it will mark itself as *WHITE*.

If one of these pruning rules is true, then the node v changes its color to *WHITE*. Otherwise, the node is suitable for the condition C16 and marks itself as *BLACK*. If the node determines its color as *BLACK*, it switches its state to *IDLE*, if it determines its color as *WHITE*, the node needs to set its clusterhead. If the node is suitable for the condition C12 or C13, which indicates that the node is dominated by one or two *BLACK* neighbors, it switches its state to *IDLE* and sets its clusterhead from one of its dominators.

If the node is suitable for the condition C14 or C15, it checks its neighbors' colors and selects its smallest id *BLACK* colored neighbor as its clusterhead and switches its state to *IDLE*.

If the node is suitable for the conditions C17 and C18 it has to find out which neighbor is its clusterhead before it switches to *IDLE* state. Therefore the node multicasts a *Cluster_REQ* message and changes its state to *CHK_CH*. It waits at this state and multi-casts *Cluster_REQ* message periodically until at least one *BLACK* colored neighbor sends a *Cluster_RES* message.

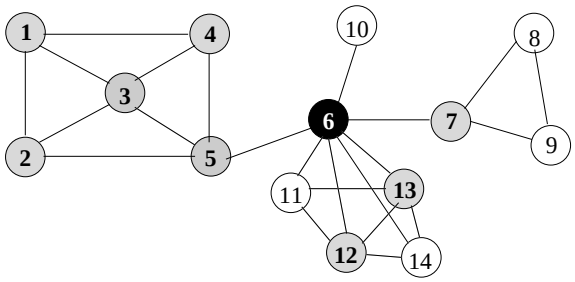

Figure 3.2. Example after the first phase

At this stage we are definitely sure that at least one neighbor will mark its color as *BLACK* and sends a *Cluster_RES* message.

At any state, a node can receive request messages to help other nodes run their algorithms. These messages are *Neighbor_REQ*, *Color_REQ* and *Cluster_REQ*. In such a case, the node prepares the required information requested in the received message and continues to its current operation. No state changes are performed in these cases.

3.2. An Example Operation

We obtained the resulting connected dominating set in Fig. 3.3 by using our algorithm. This section explains the algorithm step by step by using the sample graph in Fig. 3.3. Runtime of the algorithm is explained phase by phase, for all nodes.

- *Execution of algorithm:*

  At the end of the first phase, nodes 6, 8, 9, 10, 11 and 14 determine their colors permanently. Node 6 has an isolated neighbor, it satisfies the condition C6 thus changes its color to *BLACK*. Nodes 8, 9, 11 and 14 satisfy condition C9 and change their colors to *WHITE* because all of their neighbors are directly connected to each other and the graph is not complete. These nodes changes their states to *CHK_CH* in order to set their clusterheads. Node 10 is an isolated node, it is suitable for the condition C5, therefore it changes its color to *WHITE* and sets its clusterhead as node 6. Other nodes become *GRAY* colored because none of them are suitable for the conditions C3 to C9 and all of them satisfy the condition C10.

  At the end of the first phase, the resulting colors of the nodes can be seen in Fig. 3.2.

  In the second phase the CDSC algorithm checks the conditions C12 to C18. At the end of

this phase, nodes 1, 2 and 4 determine their colors as *WHITE* because they are suitable for the condition C17 and switch their states to *CHK_CH* in order to set their clusterheads. Nodes 12 and 13 change their colors as *WHITE* because they are suitable for the condition C12, they also set their clusterheads as node 6. Nodes 3, 5 and 7 change their colors to *BLACK* as they satisfy condition C16.

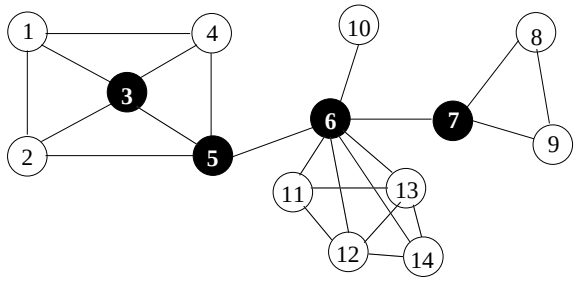

Figure 3.3. Example resulting graph

3.3. Analysis

Theorem 3.3.1. *Time complexity of the clustering algorithm is* $O(6)$.

Proof. Each node executes the distributed algorithm by the exchange of at most 6 messages. Since all these communication occurs concurrently, at the end of this phase, the members of the CDS are determined, therefore the time complexity of the algorithm is $O(6)$. The time complexity during the local iterations are ignorable compared to the messaging durations, therefore they are ignored during the time complexity analysis. □

Theorem 3.3.2. *Message complexity of the clustering algorithm is* $O(n^2)$ *where* n *is the number of nodes in the graph.*

Proof. For every mark operation of a node, at most 6 messages are required (*Neighbor_REQ, Neighbor_LST, Color_REQ, Color_RES, Cluster_REQ, Cluster_RES*). Assuming every node has n-1 adjacent neighbors, total number of messages sent is $6(n-1)$. Since there are n nodes, total number of messages exchanged during the algorithm is $n(6(n-1))$. Therefore message complexity of our algorithm is $O(n^2)$. □

3.4. Results

We implemented the Dominating Set Based Clustering Algorithm using C++ on top of the network simulator *ns2*. We generated random scenarios for static and dynamic graphs. In our experiments we collected test results for the runtime of the algorithm and the resulting number of clusterheads, namely cluster quality.

During the experiments we used three different parameters which are number of nodes, mobility of nodes and density of the network. We determined 4 number of nodes scenarios which are 20, 30, 40 and 50 nodes. We decided to use the degree of the graph as the density parameter. In the tests as the surface area decreases the density of the graph increases, it means that the nodes will have more neighbors in a smaller surface area. We set the surface area such that the degree of our graph will be between 4 and 12. For the mobility parameter we generated three mobility situations namely static, low speed and high speed tests. In static tests nodes remain still. In the low and high speed mobility scenarios respective node speeds are limited from 1 m/s to 5 m/s and 5 m/s to 10 m/s. The speed of the nodes are determined randomly by the simulation environment within the specified velocity limits. In dynamic graph experiments we take into account only the experiments in which all nodes are moving but the neighborhoods of the nodes do not change. Because the change in the neighborhoods, results in invalidation of the neighborhood information which is already distributed to the nodes.

The parameters which are described above generate 108 different test cases with the specified values. During the tests we collected 6 test results for each of the 108 different test cases. Total of 648 samples are collected during the CDSC Algorithm tests.

In Fig. 3.4, we tested the runtime of the algorithm in a static graph. In this experiment we observed that the time needed to complete the cluster formations is below 20 seconds for the densities below 8. We also observed that the runtime is nearly the same for the nodes 20 to 50 for densities smaller than 8. This is because the algorithm runs distributed in each node and is independent from the size of the graph. The only parameter that affects the runtime is the density of the graph. The density of the graph determines the number of messages exchanged between the neighbor nodes. The limitation with the maximum degree is that for the degrees over a limit value, the message conflicts increase dramatically, this results in a sudden increase in the runtime of the algorithm and makes the observations meaningless.

In Fig. 3.5 we experimented the same runtime test using a low speed dynamic graph.

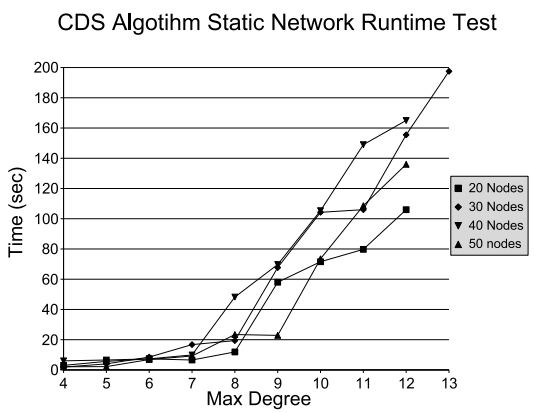

Figure 3.4. Runtime Test in a Static Network

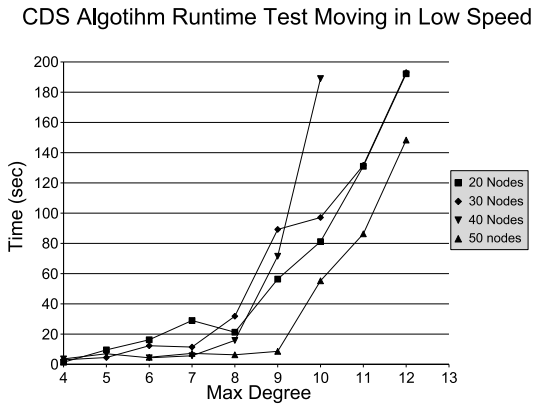

Figure 3.5. Runtime Test in a Low Speed Dynamic Network

In this experiment, we observed that the graph has nearly the same shape with the static graph test. The Fig. 3.6 shows the test results in which the nodes are moving faster. The density limit decreases to 7 and runtime values become irregular over this limit, because in some experiments the graph couldn't keep its neighborhoods stable. It means that as the message conflicts occur, the runtime of the algorithm increases, and longer experiments result in changes in the neighborhoods when the nodes are moving relatively faster.

In Fig. 3.7 we observed the number of clusterheads in a graph of size 20 to 50 for varying densities from 4 to 13. Typically in a graph, we expect to have less clusterheads as density increases. We can see the decrease in the clusterhead numbers in the graph as the degree value increases.

We repeated this experiment for dynamic graph moving in low and high speeds. The results for these experiments can be seen in Fig. 3.8 and Fig. 3.9 respectively. We can see

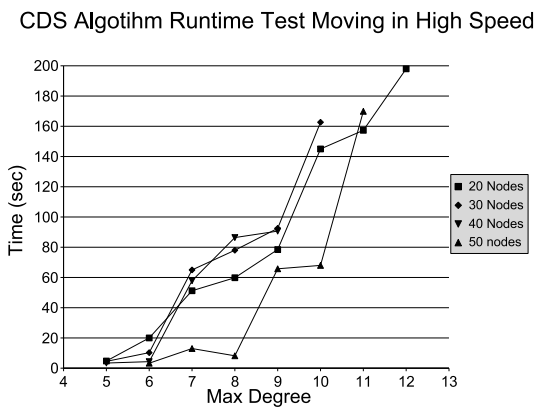

Figure 3.6. Runtime Test in a High Speed Dynamic Network

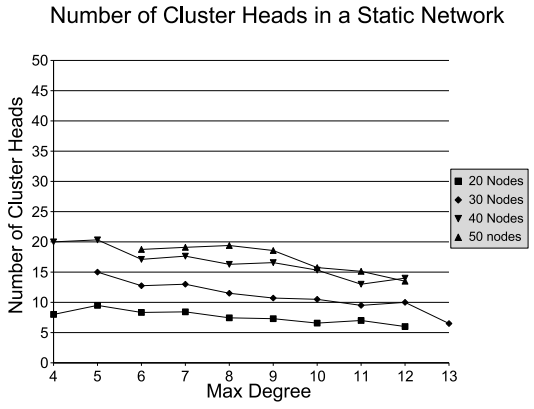

Figure 3.7. Number of Clusterheads in a Static Network

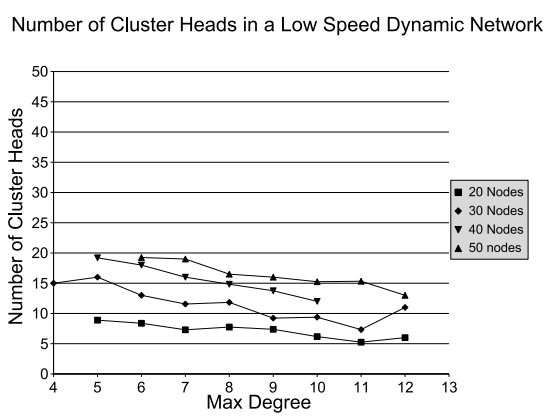

Figure 3.8. Number of Clusterheads in a Low Speed Dynamic Network

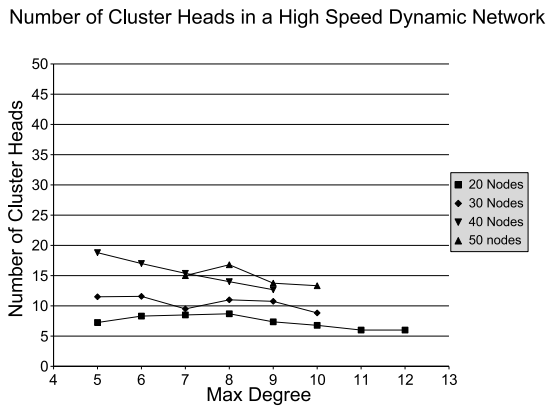

Figure 3.9. Number of Clusterheads in a High Speed Dynamic Network

almost the same decrease in the clusterheads as the maximum degree increases.

In Fig. 3.10, we varied the size of the clusters for the different parameters. Typically as the density increases, the number of clusterheads decreases. Therefore, we expect an increase in the size of the clusters as the degree increases. We can see this increase in the experiment results.

We repeated this experiment for low speed and high speed dynamic graphs. In Fig. 3.11 and Fig. 3.12, we can see that the same decrease exists in dynamic graph experiments too.

In Fig. 3.13, we calculated the deviation of the resulted cluster sizes from the average cluster size. We calculated coefficient of variances and plotted graphs for the cluster qualities. We expect to have nearly equal sizes of clusters in our network. That means, we expect to have small readings of coefficient of variances in order to say that there are little differences between clusters. In the static network of Fig. 3.13, we can see that the coefficient of variances remains

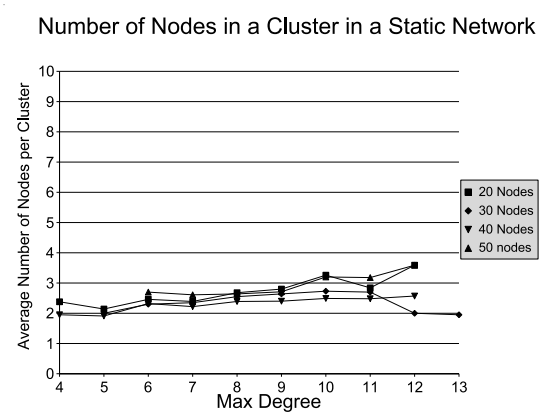

Figure 3.10. Cluster Size Test in a Static Network

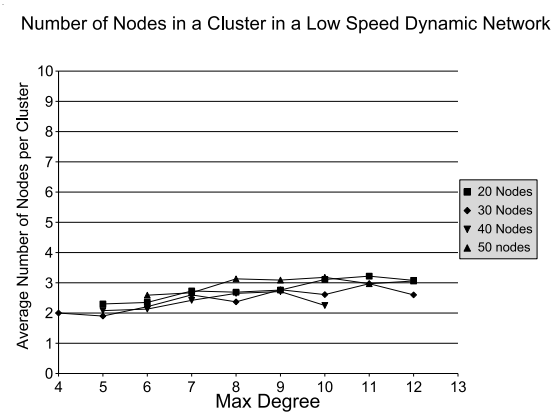

Figure 3.11. Cluster Size Test in a Low Speed Dynamic Network

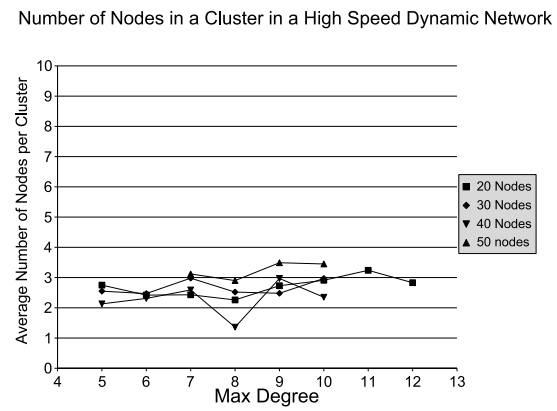

Figure 3.12. Cluster Size Test in a High Speed Dynamic Network

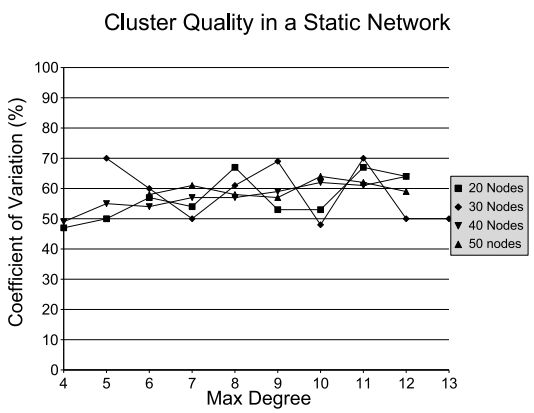

Figure 3.13. Cluster Quality Test in a Static Network

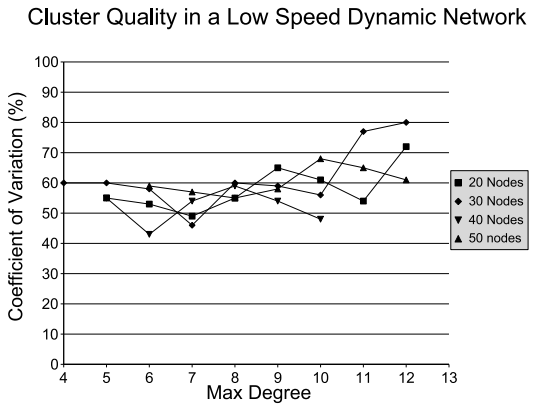

Figure 3.14. Cluster Quality Test in a Low Speed Dynamic Network

between %50 and %70, in low speed dynamic networks, in Fig. 3.14, it is between %40 and %80 and in high speed dynamic networks of Fig. 3.15 it remains between %40 and %70 in our experiments. If we look at Fig. 3.10, Fig. 3.11 and Fig. 3.12, we can see that the cluster sizes are between 2 and 4 nodes per cluster. When we look at the coefficient of variances, we can see clusters vary maximum of 3 nodes per cluster in terms of the cluster sizes. In our case, 3 nodes of differences do not affect the communication quality in a cluster of size 4. Therefore, we can say that cluster qualities are good enough to meet the communication quality expectations.

In Fig. 3.16 we calculated the average number of clusterheads for all densities. We plotted the average clusterhead numbers for static, low and high speed dynamic graphs to see the effect of dynamicity in the network. In each case, we observed that the number of clusterheads is almost proportional to the number of nodes in the network and nearly the same for the three mobility scenarios. According to these readings, we can say that our algorithm is

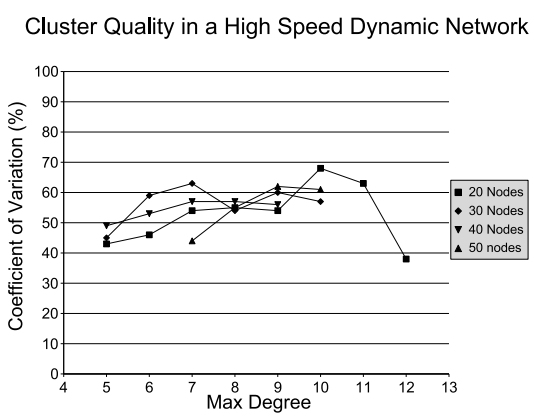

Figure 3.15. Cluster Quality Test in a High Speed Dynamic Network

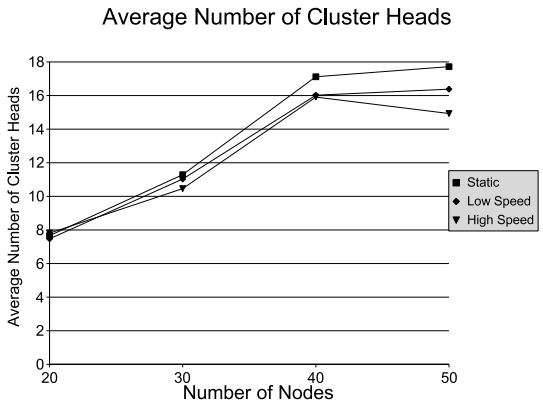

Figure 3.16. Average Number of Clusterheads Test in a Static, Moving in Low Speed and Moving in High Speed Dynamic Network

independent from the dynamicity of the network.

The comparison between Wu's CDS Algorithm and the CDSC Algorithm is given in Fig. 3.17. As it can be seen from the figure, CDSC Algorithm, results in less number of clusterheads than the Wu's CDS Algorithm.

Our overall results show that the CDSC Algorithm is independent from the mobility of the MANET. They also prove that the algorithm has very little dependence on the size of the network. The results showed that the CDSC Algorithm performs better than the Wu's CDS Algorithm for the given scenario tests. We can say that the algorithm can be preferable in environments in which the density value does not exceed the limit values shown in the graphs.

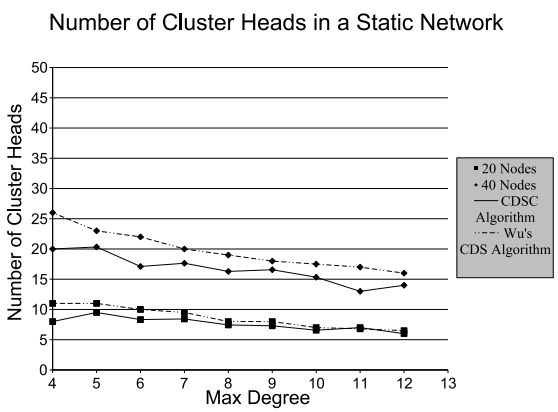

Figure 3.17. Comparison Between CDSC and Wu's CDS Algorithms in terms of Number of Clusterheads

3.5. Discussions

CDSC Algorithm results in a less crowded connected dominating sets than Wu's CDS algorithm. We also ensure that the additional heuristics and pruning rules do not change the performance of the algorithm. Wu's CDS Algorithm has $O(n^2)$ message complexity which is the same as our CDSC Algorithm. The time complexity of the modifications are ignorable compared to the unchanged messaging complexity of the algorithm.

The limitations of the CDSC Algorithm are the dynamicity of the nodes and the density of the network. We always assumed that the neighborhoods of the nodes remain constant during the runtime of the algorithm. That means nodes are moving within the intersectional coverage area of their neighbors. This is because a change in the neighborhood of any node may result in undeliverable messages as well as damaging the properties of the resulting CDS.

The limitation concerning the density of the network lead us to investigate the cause of this problem. At the end of a debugging period of the runtime of the CDSC Algorithm, we noticed that the degree limitation is raised because of the conflicting messages in the wireless network. We observed that above a limit value in the degree of the nodes, the message conflicts increase dramatically. Moreover, we observed that the nodes re-send the dropped messages at the same time because they all use same clock provided by the *ns2* simulator. Some node specific parameters such as different period values may help us to solve this problem but in order to distribute the same code evenly to all nodes, this solution is not favorable. Instead, this problem shows us that message conflicts may be a serious problem in wireless networking

protocols. A MAC Layer solution may help us to definitely solve the degree limitation of the CDSC Algorithm which is caused by the message conflicts.

The latest limitation of the CDSC Algorithm is concerning to the connectivity of the network. The nature of CDS requires the network should be connected without any isolated nodes, which is a fair assumption. Therefore in our tests we ignored such scenarios as they do not reflect reliable statistics for the tested scenario specifications.

The CDSC Algorithm results in clusters which are determined by a constructed connected dominating set. At the end of the CDSC Algorithm, we construct a connected dominating set and each *WHITE* colored node determines a suitable clusterhead for itself. But at the end of the algorithm, clusterheads are not aware of the members of their clusters. Some applications such as CDS Flooding may require this information. These kind of properties may be simply added to the protocol by inserting one or more states to the finite state machine.

CHAPTER 4

TWO-LEVEL CONNECTED DOMINATING SET BASED CLUSTERING ALGORITHM

4.1. General Idea and Description of the Algorithm

The Two-Level Connected Dominating Set Based Clustering Algorithm (TLCDSC) finds two minimal connected dominating sets in a MANET. We developed our algorithm as an extension to CDSC Algorithm. First, we find a CDS on a MANET using our CDSC Algorithm and call the resulting subset of clusterheads as *First Level CDS*, then we run the same clustering algorithm on the subset of *First Level CDS*. At the end of the algorithm, a two level connected dominating set is constructed. *First Level CDS* is composed of *Cluster Heads* and *Second Level CDS* is composed of *Super Cluster Heads*. The TLCDSC Algorithm provides more crowded clusters which are relatively better than our first approach in which the size of the clusters are very small compared to the number of nodes in the MANET.

The assumptions and coloring rules which are explained in the CDSC Algorithm are also required for the TLCDSC Algorithm as it is an extended version of the CDSC Algorithm. We assume that the nodes are almost static in a reasonable period of time in order to complete a whole cycle in every single node. We also assume that the graph is connected, each node has a unique *node_id* and knows its adjacent neighbors. Each node has a *color* indicating whether the node is in the dominating set or not. The *color* is set to *BLACK* if the node is in the dominating set, or *WHITE* if the node is not in the dominating set. Color *GRAY* is used to indicate that the node is marked after the first phase, but it will change its color after the second phase either to *WHITE* or *BLACK*. At the end of the algorithm, every node will be in either one of the three states namely *WHITE_STATE*, *BLACK_STATE* and *RED_STATE* which indicate that the node is an *Ordinary Node*, a *Cluster Head* or a *Super Cluster Head* respectively.

The basic idea of the algorithm is very similar to the CDSC Algorithm. The finite state machine and the transitions between states differ from the CDSC because of the second level

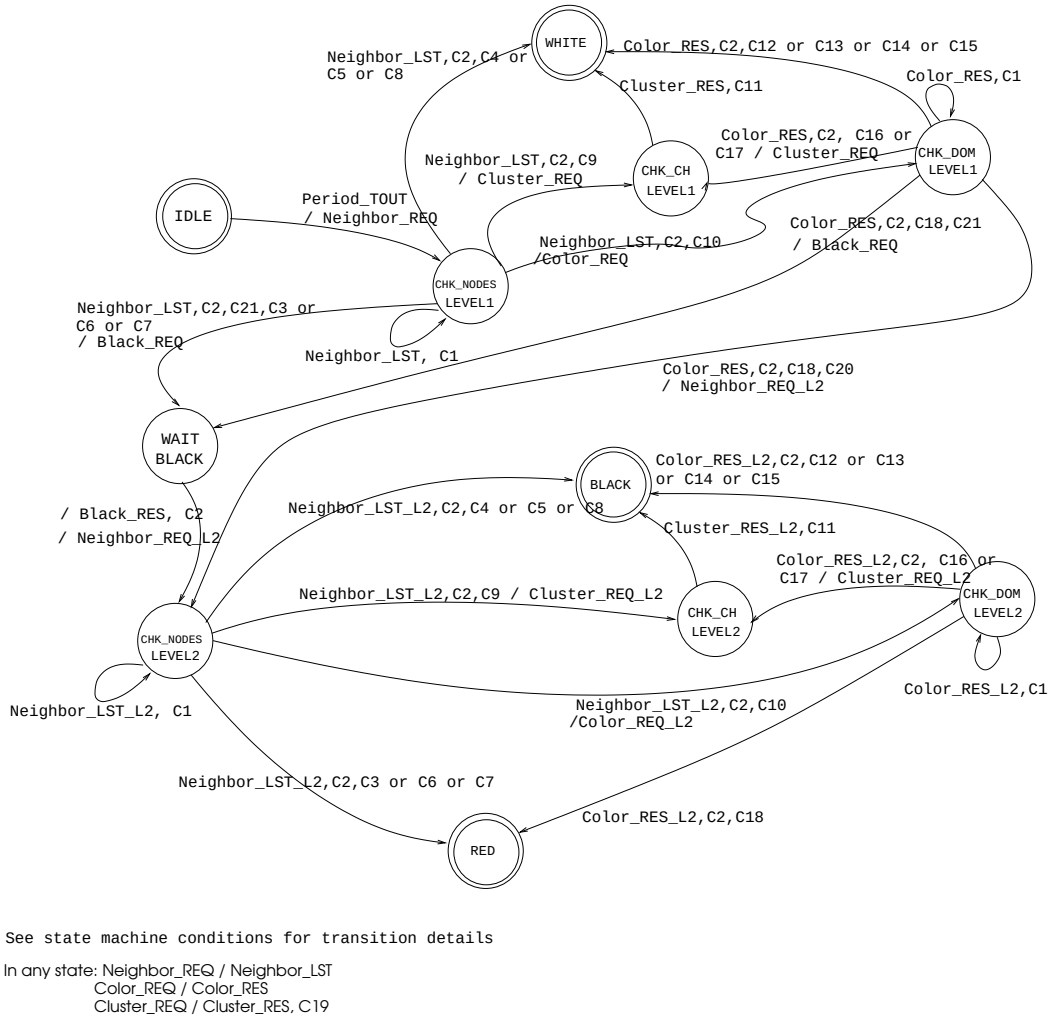

Figure 4.1. Finite State Machine of the TLCDSC Algorithm

clustering.

The message types used in the TLCDSC Algorithm can be classified as first level and second level messages. The first level messages are *Period_TOUT*, *Neighbor_REQ*, *Neighbor_LST*, *Color_REQ*, *Color_RES*, *Cluster_REQ* and *Cluster_RES* which are described in the CDSC Algorithm in detail, *Black_REQ* and *Black_RES* which are described below.

- *Black_REQ*: When a node determines its first level color as *BLACK* it will check its neighbors' first level colors in order to find out which nodes will be in its second level neighborhood. If there are still *GRAY* of *UNDEFINED* colored neighbors, a *Black_REQ* message is multicasted to those neighbors in order to learn their permanent first level colors.

- *Black_RES*: A node which receives a *Black_REQ* message, checks if it is in the multicast list which is provided in the message body. If it is in the list and if it determined its first level color as either *WHITE* or *BLACK* then it sends a *Black_RES* message indicating its first level permanent color.

These messages are exchanged until all nodes determine their first level colors. The message types used during the second level clustering are: *Neighbor_REQ_L2*, *Neighbor_LST_L2*, *Color_REQ_L2*, *Color_RES_L2* and *Cluster_REQ_L2*, *Cluster_RES_L2*. The effect of the messages are the same with the first level messages except they are applied to a subset of the graph which consists of nodes which are *BLACK* colored after the first phase.

Every node in the network performs the same local algorithm periodically. The finite state diagram of the algorithm can be seen in Fig. 4.1. During the runtime of the *TLCDSC Algorithm*, the state machine transition conditions which determine state transitions are very similar to those which are described in the CDSC Algorithm. A modified list of the conditions are described below:

TLCDSC Algorithm Finite State Machine Transition Conditions:

- C1. The responses to the multicasted message are not completely collected.

- C2. The responses to the multicasted message are completely collected.

- C3. The node is isolated, its neighbor is isolated too and node's id is bigger than its neighbor's id.

- C4. The node is isolated, its neighbor is isolated too and the node's id is smaller than its neighbor's id.
- C5. The node is isolated and its neighbor is not isolated.
- C6. The node has at least one isolated neighbor.
- C7. The graph is complete and the node has the biggest id in the graph.
- C8. The graph is complete and the node does not have the biggest id.
- C9. Node's neighbors are all connected and the graph is not complete.
- C10. The node has at least two unconnected neighbors.
- C11. Cluster Head is set to the sender's id.
- C12. CDSC pruning rule 1 is true.
- C13. CDSC pruning rule 2 is true.
- C14. CDSC pruning rule 3 is true and the node has at least one *BLACK* neighbor.
- C15. CDSC pruning rule 4 is true and the node has at least one *BLACK* neighbor.
- C16. CDSC pruning rule 3 is true and the node doesn't have any *BLACK* neighbor.
- C17. CDSC pruning rule 4 is true and the node doesn't have any *BLACK* neighbor.
- C18. Conditions C12 to C17 are all false.
- C19. Node's color is currently *BLACK*.
- C20. Node's neighbors completed the first level and determined their first level color.
- C21. The node still has GRAY colored neighbors in its color list.
- C22. The node does not have any neighbors.

Although basic idea of the first level clustering of the TLCDSC Algorithm is quite similar to the CDSC Algorithm, in order to explain the FSM completely, we explain the whole sequence.

Each node is in the *IDLE* state and colored as *UNDEFINED_COLOR* initially. When the period is timed out, the node sends a *Period_TOUT* message to itself. This message causes the node to switch its state to *CHK_NODES_LEVEL1* and send a *Neighbor_REQ* message to all of its adjacent neighbors. Then the node waits for *Neighbor_LST* messages from all of its adjacent neighbors. When all *Neighbor_LST* messages are collected, the node checks the heuristics *C3* to *C9* which are defined in the state machine transition conditions list, to determine their next state transition. If the node is suitable for conditions C4, C5 or C8, it determines its *First Level* and its *Second Level Colors* as *WHITE* and changes its state to *WHITE_STATE*. In all these three conditions, the node can determine its clusterhead. For conditions C4 and C5, the clusterhead is determined as the node's single neighbor, and in the condition C8, clusterhead is determined as the node with the maximum degree in the complete graph. Such a node completes the algorithm in this step and becomes an *Ordinary Node*.

If condition C9 is true for a node, it marks its *First Level Color* as *WHITE*, switches its state to *CHK_CH_LEVEL1* and multicasts a *Cluster_REQ* message in order to learn which neighbor became its clusterhead. When the first *Cluster_RES* message is received, the node sets its clusterhead as the sender of the message, changes its state to *WHITE_STATE*, sets its *First* and its *Second Level Colors* as *WHITE* and finishes the algorithm.

If the node is suitable for the condition C10, it is potentially a clusterhead candidate. In this case, the node switches its state to *CHK_DOM_LEVEL1*, changes its *First Level Color* to *GRAY* and multicasts a *Color_REQ* message in order to collect its neighbors' colors. When the node switches its state to *CHK_DOM_LEVEL1*, it waits for all of its neighbors to send their colors. When the node v collects all the color information, it starts to apply the CDS pruning rules which are described previously in the CDS Algorithm. The pruning rules are described below:

CDS Algorithm Pruning Rules:

1. $\exists u \in N(v)$ which is marked $BLACK$ such that $N[v] \subseteq N[u]$;
   If the node has a *BLACK* neighbor which covers its closed neighborhood then it should mark itself as *WHITE* because there is already a *BLACK* node which dominates all the closed neighbors.

2. $\exists u, w \in N(v)$which is marked $BLACK$ such that $N(v) \subseteq N(u) \bigcup N(w)$;

If the node has two connected *BLACK* neighbors and if the union of the neighborhoods of these *BLACK* neighbors cover the node's closed neighborhood then the node should mark itself as *WHITE*.

3. $\exists u \in N(v)$ which is marked *GRAY* such that $N[v] \subseteq N[u]$ and $degree(v) < degree(u)$ OR $(degree(v) = degree(u)$ AND $id(v) < id(u))$;
   If the node has a *GRAY* colored neighbor which covers its closed neighborhood, it means both of them are candidate to be in dominating set. In this case, if the node has smaller degree than its *GRAY* colored neighbor, it should mark itself as *WHITE*.

4. $\exists u, w \in N(v)$which is marked *GRAY* OR *BLACK* such that $N(v) \subseteq N(u) \bigcup N(w)$ and $degree(v) < min\{degree(u), degree(w)\}$ OR $degree(v) = min\{degree(u), degree(w)\}$ AND $id(v) < min\{id(u), id(w)\}$;
   If the node has two connected *BLACK* or *GRAY* neighbors which covers its closed neighborhood and if its degree is smaller than these neighbors degrees, then it will mark itself as *WHITE*.

If the node is suitable for conditions C12, C13, C14 or C15, it finishes the algorithm, changes its state to *WHITE_STATE* and sets its *first* and *second level colors* to *WHITE*. If the node is suitable for one of the conditions C12 or C13 which indicate that the node is dominated by one or two *BLACK* neighbors, it sets its clusterhead as one of its dominators. If the node is suitable for the conditions C14 or C15 it selects smallest id *BLACK* colored neighbor as its clusterhead.

If the node is suitable for conditions C16 or C17, it changes its state to *CHK_CH_LEVEL1* and multicasts a *Cluster_REQ* message in order to learn which neighbor become its clusterhead. When the first *Cluster_RES* message is received, the node sets its clusterhead as the sender of the message, changes its state to *WHITE_STATE*, sets its *First* and its *Second Level Colors* as *WHITE* and finishes the algorithm.

If none of the four pruning rules is true, then the node is suitable for the condition C18, it then marks its *first level color* as *BLACK*. In this case, if the node is also suitable for the condition C21, it changes its state to *WAIT_BLACK* and multicasts a *Black_REQ* message to its *GRAY* colored neighbors in order to wait its neighbors to determine their permanent colors. If the node's neighbors have already determined their *First Level Colors*, then the node changes

its state to *CHK_NODES_LEVEL2* and multicasts a *Neighbor_REQ_L2* message to its *Second Level Neighbors*.

If the node is suitable for the conditions C3, C6 or C7, it changes its *First Level Color* as *BLACK* and switches to state *WAIT_BLACK* and broadcasts a *Black_REQ* message to all its neighbors. When a node which is in the *WAIT_BLACK* state, collects all *Black_RES* messages, it changes its state to *CHK_NODES_LEVEL2*.

Nodes which reach the *CHK_NODES_LEVEL2* state finish the *First Level Clustering* and from this point they start to execute the *Second Level Clustering*. The algorithm which is used in the *Second Level Clustering* is the same algorithm which is used during the *First Level Clustering*. The most important difference is the new set of nodes used in the *Second Level Clustering* are the nodes which are *BLACK* colored after the *First Level Clustering*. We create a subset S which is composed of *First Level Cluster Heads*, and apply the *CDS* algorithm to the subset S.

In the *Second Level Clustering*, nodes which are not *Level 2 Cluster Heads* end in the state *BLACK_STATE* indicating that they are *Level 1 Cluster Heads*. *Level 2 Cluster Heads* end in the state *RED_STATE*.

When the *TLCDSC Algorithm* is finished, the nodes are in either one of the *WHITE_STATE*, *BLACK_STATE* or *RED_STATE*. The cluster information for a node is held locally, each node knows only its clusterhead. This makes our algorithm more flexible, thus our algorithm can be easily extended to a K-Level Hierarchical Clustering.

At any state, a node can receive request messages to help other nodes run their algorithms. These messages are *Neighbor_REQ*, *Cluster_REQ*, *Color_REQ*, *Neighbor_REQ_L2*, *Cluster_REQ_L2*, *Color_REQ_L2* and *Black_REQ*. In such a case, the node prepares the required information requested in the received message and continues its current operation. No state changes are performed in these cases.

4.2. An Example Operation

We obtained the resulting connected dominating set in Fig. 4.4 by using our algorithm. On the sample graph of Fig. 4.2 the execution of the algorithm is explained step by step for all nodes.

- *Execution of First Level Clustering:*

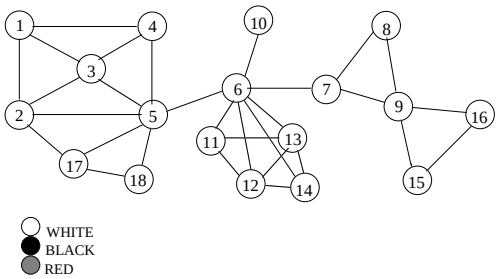

Figure 4.2. TLCDSC Algorithm Sample Graph

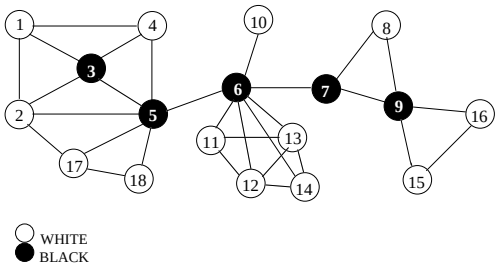

Figure 4.3. TLCDSC Algorithm Samle Graph at the End of First Level

At the end of the first phase of the *First Level Clustering*, nodes 6, 8, 10, 11, 14, 15, 16 and 18 determine their colors permanently. Node 6 satisfies the condition C6, thus changes its *First Level Color* to *BLACK* and finishes its *First Level Clustering* and changes its state to *WAIT_BLACK*. Nodes 8, 11, 14, 15, 16 and 18 satisfy condition C9 and change their *First Level Colors* to *WHITE*. Nodes 8, 15, 16 and 18 change their states to *CHK_CH_LEVEL1* in order to set their clusterheads. Nodes 11 and 14 set their clusterhead as node 6 and finish their execution. Node 10 is an isolated node, therefore it changes its *First Level Color* to *WHITE* and sets its clusterhead as node 6 and finish its execution. Other nodes become *GRAY* colored, because all of them satisfy the condition C10.

In the second phase of the *First Level Clustering*, the CDS algorithm checks the conditions C12 to C18. At the end of this phase, nodes 1, 2, 4, 12, 13 and 17 determine their colors as *WHITE*, because they are suitable for one of the four pruning rules. Nodes 12 and 13 select node 6 as their clusterhead and finish their *First Level Clustering*. Nodes 1, 2, 4 and 17 change their states to *CHK_CH_LEVEL1* in order to set their clusterheads. Nodes 3, 5, 7 and 9 change their colors to *BLACK* as they satisfy condition the C18. At the end of the *First Level Clustering*, the resulting CDS can be seen in Fig. 4.3.

- *Execution of Second Level Clustering:* The *Second Level Clustering* uses the new subset

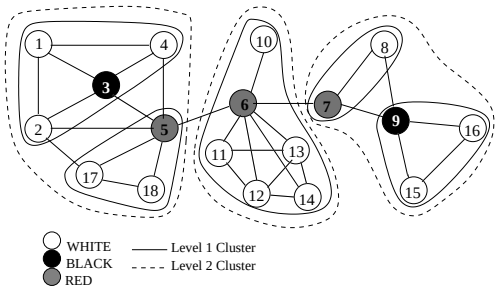

Figure 4.4. TLCDSC Algorithm Resulting Graph

of *First Level Cluster Heads* as its domain, therefore the working set for the *Second Level Clustering* is nodes 3, 5, 6, 7 and 9. When the *Second Level Clustering* starts its execution, nodes 3 and 9 determine their *Second Level Colors* as *WHITE* because they are suitable for the condition C5. Thus nodes 3 and 9 finish their *Second Level Clustering* by determining their *Second Level Colors* as *WHITE* and their end states as *BLACK_STATE*. Nodes 5 and 7 are suitable for the condition C6, thus they finish their *Second Level Clustering* as *Super Cluster Heads* by determining their *Second Level Colors* as *BLACK* and their end states as *RED_STATE*. Node 6 is suitable for the condition C10, thus it changes its state to *CHK_DOM_LEVEL2*. After collecting its neighbor's second level colors, it determines its *Second Level Color* as *BLACK* because it is suitable for the condition C18. The overall result of the *2-Level CDS Algorithm* can be seen in Fig. 4.4.

4.3. Analysis

Theorem 4.3.1. *Time complexity of the TLCDSC Algorithm is* $O(10)$.

Proof. Every node executes the distributed algorithm by the exchange of at most 10 messages (*Neighbor_REQ, Neighbor_LST, Color_REQ, Color_RES, Black_REQ, Black_RES, Neighbor_REQ_L2, Neighbor_LST_L2, Color_REQ_L2, Color_RES_L2*). As explained in Section 4.1., in the worst case scenario, after exchanging of these 10 messages, the members of Two-Level CDS are determined. The time complexity during the local iterations are ignorable compared to the messaging durations, therefore the time complexity of the local iterations are ignored during the time complexity analysis. Since all these communication occurs concurrently in every node, the time complexity of the algorithm is $O(10)$. □

Theorem 4.3.2. *Message complexity of the TLCDSC Algorithm is* $O(n^2)$ *where n is the*

number of nodes in the graph.

Proof. For every mark operation of a node, 10 messages are required (*Neighbor_REQ, Neighbor_LST, Color_REQ, Color_RES, Black_REQ, Black_RES, Neighbor_REQ_L2, Neighbor_LST_L2, Color_REQ_L2, Color_RES_L2*). Assuming every node has n-1 adjacent neighbors, total number of messages sent is $10(n-1)$. Since there are n nodes, total number of messages in the system is $n(10(n-1))$ Therefore messaging complexity of our algorithm has an upperbound of $O(n^2)$. □

4.4. Results

We implemented the TLCDSC Algorithm using C++ on top of the network simulator *ns2*. We generated random scenarios for static and dynamic graphs. In our experiments, we collected test results for the runtime of the algorithm, size of the superclusters and the resulting number of super clusterheads.

During the experiments, we used three parameters which are number of nodes, mobility of nodes and density of the network. We determined 4 "number of nodes scenarios" which have 20, 30, 40 and 50 nodes. We used the degree of the graph as the density parameter. As the surface area decreases the density of the graph increases, it means that the nodes will have greater degrees. We set the surface area such that the degree of our graph will be between 4 and 10. For the mobility parameter we generated three "mobility scenarios" namely static, low speed and high speed. In the static scenario tests, nodes remain still. In the low and high mobility scenarios, respective node speeds are limited from 1 m/s to 5 m/s and from 5 m/s to 10 m/s. The speed of the nodes is determined randomly by the simulation environment within the specified velocity limits. In the dynamic graph experiments, we take into account only the experiments in which nodes are moving but the neighborhoods of the nodes do not change.

The parameters which are described above generate 84 different test cases with the specified values. During the tests we collected an average of 60 test results for each of the 84 different test cases. Total of 5000 samples were collected during the TLCDSC Algorithm tests.

In Fig. 4.5, we tested the runtime of the algorithm in a static network. In this experiment we observed that the runtime of the algorithm is below 25 seconds for the densities nearly below 6. We also observed that the runtime is nearly the same for the nodes 20 to 50 for densities smaller than 5. This is because the algorithm runs distributed in each node and is independent

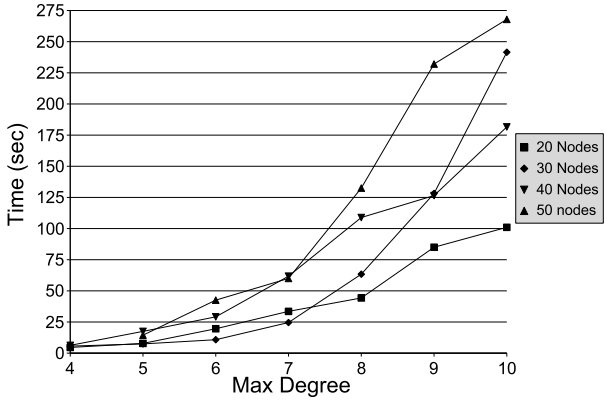

Figure 4.5. TLCDSC Algorithm Runtime Test in a Static Network

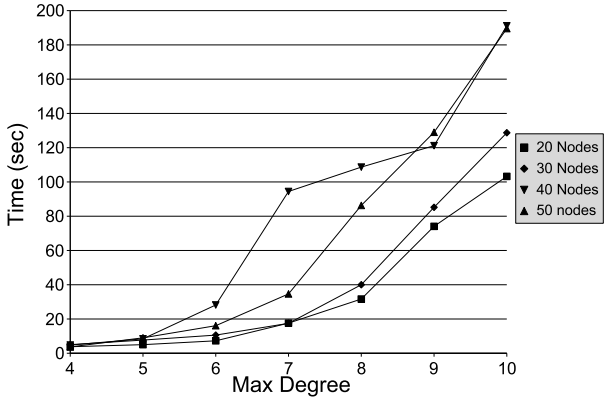

Figure 4.6. TLCDSC Algorithm Runtime Test in a Low Speed Dynamic Network

from the size of the graph. The runtime increases dramatically for the densities above 6. This is because of the conflicting messages in the mobile network.

In Fig. 4.6, we can see that in a low speed mobile network, runtime of the algorithm is similar to the static network as long as the neighborhoods do not change. The only parameter that affects the runtime is the density of the graph which determines the number of messages exchanged between the neighbor nodes. For higher degrees, the message conflicts increase dramatically which results in a sudden increase in the runtime of the algorithm and makes the observations meaningless.

The test results for the high speed dynamic networks can be seen in Fig. 4.7. As can be seen in the figure, we could not collect meaningful data for the networks in which the number

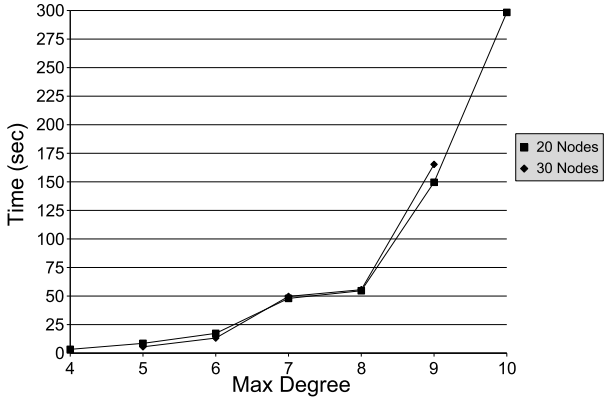

Figure 4.7. TLCDSC Algorithm Runtime Test in a High Speed Dynamic Network

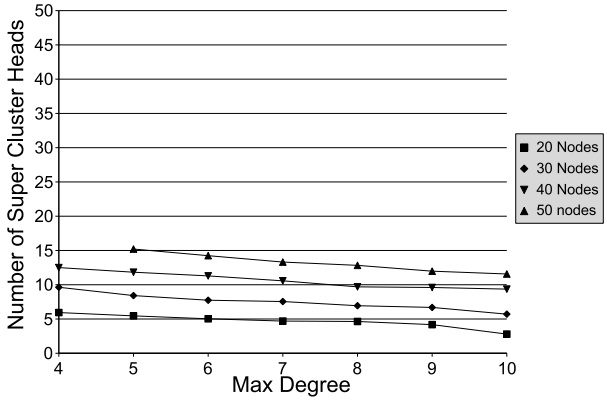

Figure 4.8. Number of Super Cluster Heads in a Static Network

of nodes are above 30. This is because the time required to construct the two level clusters is larger than the duration in which the neighborhoods of the nodes remain constant.

In Fig. 4.8, Fig. 4.9 and Fig. 4.10, we observed the number of super clusterheads in our resulting clustered network. Typically in a network, we expect to have less super clusterheads as density increases. We can see the decrease in the super clusterhead numbers in the graph as the degree value increases as we expected. We can see almost the same decrease in the three mobility scenarios.

Compared to the CDSC Algorithm, the TLCDSC Algorithm selected less super clusterheads which means that the clusters will be more crowded. In fact, sum of the number of clusterheads and the number of super clusterheads in the TLCDSC Algorithm is exactly the

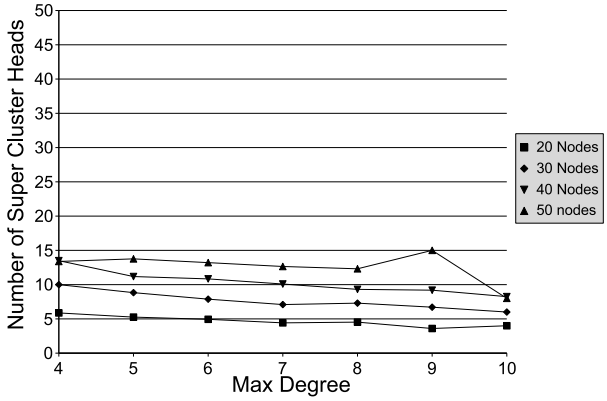

Figure 4.9. Number of Super Cluster Heads in a Low Speed Dynamic Network

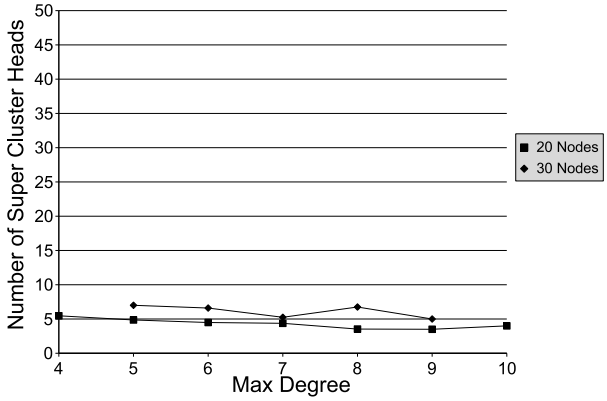

Figure 4.10. Number of Super Cluster Heads in a High Speed Dynamic Network

same with the number of clusterheads in the CDSC Algorithm. But because the TLCDSC Algorithm selects some of the clusterheads as super clusterheads, we can guarantee that the second level CDS which is constructed by TLCDSC Algorithm will consists of less number of super clusterheads.

In Fig. 4.11, we tested the size of the super clusters for varying parameters. Typically as the density increases, the number of clusterheads decreases. Therefore, we expect to have more crowded clusters as the degree increases. We can see this increase in the experiment results. As we can see from the test results, for each graph we have nearly similar cluster sizes. This result shows us that our algorithm is independent from the size of the MANET in terms of the cluster qualities. We also expected to have more crowded clusters than the CDSC Algorithm. When

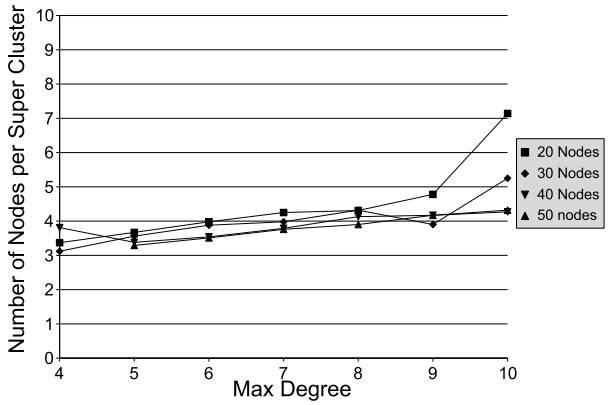

Figure 4.11. Size of the Super Clusters in a Static Network

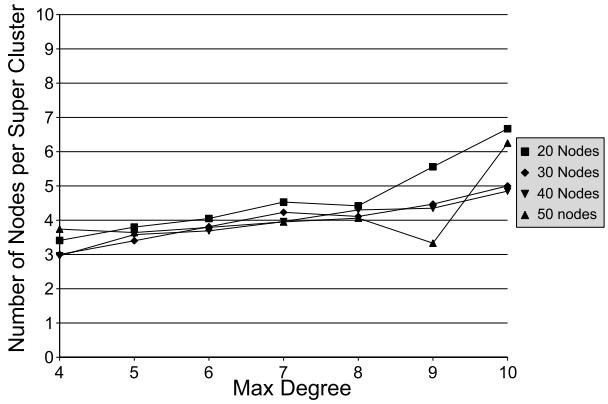

Figure 4.12. Size of the Super Clusters in a Low Speed Dynamic Network

we look at the test results we can see that in CDSC Algorithm the cluster sizes vary between 2 and 3, however the resulting super cluster sizes of the TLCDSC Algorithm vary between 3 and 5.

The Fig. 4.12 and Fig. 4.13 show that the size of the superclusters are also similar for three different mobility scenarios, which means that the algorithm is also independent from the mobility.

The test results of the TLCDSC Algorithm satisfy our expectations. These results show us that the TLCDSC Algorithm is independent from the mobility and the size of the MANET, if we ignore message conflicts. They also prove that the algorithm is independent from the number of nodes in the network. We can say that the algorithm can be preferable in environments in

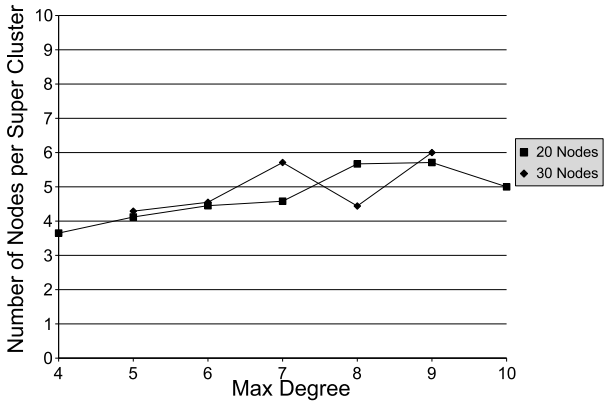

Figure 4.13. Size of the Super Clusters in a High Speed Dynamic Network

which the density value does not exceed the limit values shown in the graphs.

4.5. Discussions

The TLCDSC Algorithm showed us that CDSC Algorithm, which is explained in Chapter 3, can be easily expanded to a k-level clustering algorithm without loosing the CDS property of the resulting clusters. The advantage of using two level clusters is the capability of providing a hierarchical structure to the applications which may require group communication. This hierarchical structure may be used for efficient multicast communication among the group members. Two Level Clustering also provides a more reliable and energy efficient network topology by decreasing the hop-counts, inter-cluster message traffic and message conflicts.

The limitations of the TLCDSC Algorithm are quite similar to those which are discussed in the CDSC Algorithm, because both algorithms rely on the same fundamental logic. Although the message complexity of the TLCDSC Algorithm is greater than the CDSC Algorithm, during the experiments, the runtime of the TLCDSC Algorithm remain nearly in the same time range with the CDSC Algorithm. This is because most of the duration which is required to build the clusters is spent for the conflicting messages. We observed the difference between two algorithms more clearly in high speed dynamic network experiments in which the message conflicts occur more frequently. We couldn't collect meaningful data for the networks consisting of 40 and 50 nodes which are moving at high speed. The message complexity causes an increase in the conflicting messages which make our algorithm run longer in such cases. The runtime of the

algorithm exceeds the time limit which results in alterations in the neighborhoods of the nodes. This situation breaks our first assumption, therefore we can say that the TLCDSC Algorithm is not suitable for high speed dynamic networks without having a MAC level support to the message conflict problem.

CHAPTER 5

CDS FLOODING ALGORITHM

5.1. General Idea and Description of the Algorithm

Routing in MANETs is a very problematic issue because of the dynamicity of the network. In dynamic networks such as MANETs, routing tables should be updated very frequently. Keeping the routing tables up to date may consume a large part of the wireless traffic in the network. This traffic might sometimes be extremely dense which may possibly block the circulation of the messages between nodes. A virtually structured network such as a CDS can be considered as a good solution to make message transfers more efficient. But even in the structured networks a routing protocol is required in order to deliver messages to the destinations. CDS Flooding Algorithm is a cluster based routing algorithm. We first construct a connected dominating set by using our TLCDSC Algorithm, then implement a message flooding mechanism which uses the clusterheads as the gateways of the clusters. In CDS Flooding Algorithm, there are two types of message traffic; traffic between an ordinary node and its clusterhead, and traffic between the clusterheads. Flooding process takes place only between the clusterheads, therefore the CDS Flooding Algorithm significantly reduces the number of flooded messages in the network.

For the CDS Flooding Algorithm, we assume that CDS clusters are already constructed and every node has a unique node id. A node which would like to send a message to any node in the network, sends the message to its clusterhead. From this point, the clusterhead floods the message to the network where only clusterheads are involved in the flooding operation. When a clusterhead realizes that the destination is in its cluster, it stops flooding the message and sends the message directly to the destination. Flooding lasts until all clusterheads receive the message.

For the CDS Flooding algorithm, we used our TLCDSC as the CDS constructor algorithm. By using the TLCDSC Algorithm, we may enhance the CDS Flooding to a multi level flooding algorithm as a future work without modifying the underlying clustering algorithm.

The CDS Flooding algorithm requires that every node knows its clusterhead and ev-

ery clusterhead knows the members of its cluster. In TLCDSC Algorithm, the membership information of the clusters is missing in the clusterheads. Therefore, we modified the first level clustering part of the TLCDSC Algorithm by informing the clusterheads about the members of their clusters. We didn't modify the second level clustering part of the TLCDSC Algorithm as we don't use the second level clusters during the CDS Flooding algorithm. For this purpose we add a *WAIT_MBR_ACK* state to the finite state machine of the TLCDSC Algorithm. We also add two more messages, *Cluster_MBR_INF* and *Cluster_MBR_ACK*, which are described below.

- *Cluster_MBR_INF*: When a *WHITE* colored node determines its *first level clusterhead*, it sends a *Cluster_MBR_INF* message to its *first level clusterhead* informing that it will join the cluster of its clusterhead.

- *Cluster_MBR_ACK*: The *Cluster_MBR_ACK* message is used in order to be sure that the cluster membership information is received by the clusterhead. When a *first level clusterhead* receives a *Cluster_MBR_INF* message, it will add the sender's id to its *cluster members list* and sends a *Cluster_MBR_ACK* message to the sender node.

The modified version of the finite state machine of the TLCDSC Algorithm can be seen in Fig. 5.1. In addition to the TLCDSC Algorithm which is described in Chapter 4, the nodes which determine their first level color as *WHITE* and which have more than one clusterhead possibilities, send a *Cluster_MBR_INF* message to their clusterheads and change their state to *WAIT_MBR_ACK*. By sending this message, an ordinary node informs its clusterhead that it has joined the cluster. Then the node waits for a *Cluster_MBR_ACK* message which ensures that the clusterhead has received the cluster membership information. When a node receives a *Cluster_MBR_ACK* message, it changes its state to *WHITE_STATE* and finishes its modified TLCDSC Algorithm. At the end of the modified version of TLCDSC Algorithm two level clusters are constructed, every node has a clusterhead and every clusterhead knows the list of nodes which are in its cluster. This information satisfies the requirements which are needed by the CDS Flooding Algorithm.

The CDS Flooding Algorithm consists of two distinct procedures, sending a message and receiving a message. Each node calls the related procedure to send or receive a message.

The procedure which is executed to send a message can be seen in the Fig. 5.1. The

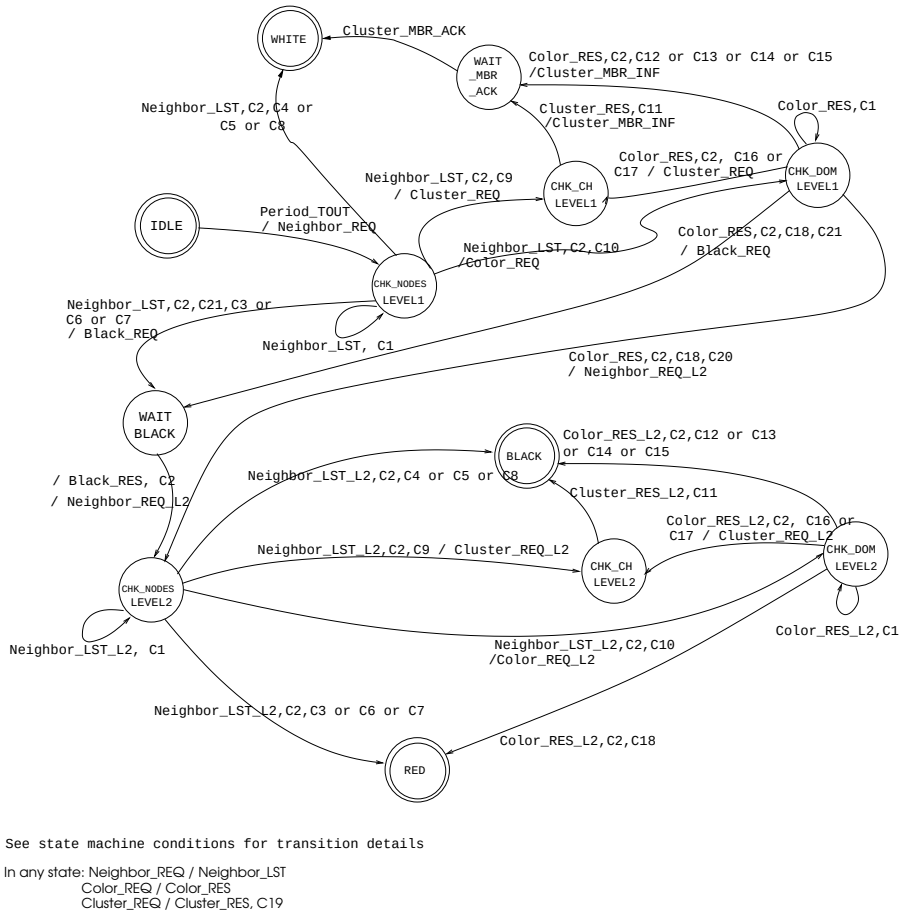

Figure 5.1. Finite State Machine of the 2-Level Hierarchical Clustering Algorithm

procedure depends on the type of the node and the destination of the message. A more detailed explanation of the algorithm can be found in 5.1.1.

```
send_proc

begin

  if i am a clusterhead then

    if the destination is in my cluster then
      send message directly to the destination.

    else
      multicast the message to my BLACK neighbors.

    end

  else
    send message to my clusterhead.

  end

end
```

Algorithm 5.1: Procedure executed by the nodes to send a message

A general description of the CDS Flooding Algorithm which is executed upon receiving a message can be seen in Fig. 5.2. The process which is executed depends on the type of the node, sender and the receiver of the message.

```
receive_proc

begin

  if i am a clusterhead then

    if the message is received for the first time then

      if i am the destination then
        process the message.

      else

        if destination is in my cluster then
          send message directly to the destination.

        else
          multicast the message to my BLACK neighbors.

        end

      end

    end

  else
    if sender of the message is my clusterhead AND i am the destination AND

    the message is received for the first time then
      process the message.

    end

  end

end
```

Algorithm 5.2: Procedure executed by the nodes upon receiving a message

The messages which are transferred during the CDS Flooding algorithm have unique message ids which are determined by the sender nodes. The message ids are calculated locally in each node according to the formula: "*MessageId = (NodeId * maximumMessageNumber) + Index*" where *NodeId* is the unique id of the sender node and *Index* is a modular incremental value which is calculated in base *maximumMessageNumber*. The formula prevents duplicate *MessageId*s in a network up to *maximumMessageNumber* messages. There is no need to have infinitely many unique message ids. After a message is completely flooded in the network, its message id can be reused. For that, we determine the *maximumMessageNumber* such that it

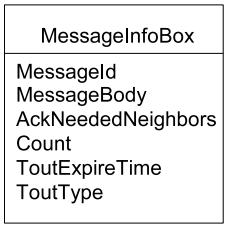

Figure 5.2. Data Structure for the MessageInfoBox

will be at least the maximum number of messages which can be flooded to the network in a duration of flooding of a single message.

Each node records the processed messages to a table which is called *MessageInfoBox*. The data structure for the *MessageInfoBox* can be seen in the Fig. 5.2. The entities in the data structure are described below:

- *MessageId*: Unique id of the messages which are calculated locally in each node according to the algorithm which is explained above.
- *MessageBody*: *MessageBody* includes the whole message object. *MessageBody* is required when the message is needed to be resent.
- *AckNeededNeighbors*: A list of nodes to which the message is sent. The list is particularly used in order to keep track of the *ACK* messages. When an *ACK* message is received, the sender of the *ACK* message is deleted from the list.
- *Count*: Count holds the size of the *AckNeededNeighbors* list and is used in order to check if there are still neighbors from which acknowledgements are expected.
- *ToutExpireTime*: The expiration time of the message inside the node. When the node reaches the *ToutExpireTime* in its local clock, it processes the message according to the *ToutType*. The *ToutExpireTime* is calculated by adding a predefined period value to the current time. Predefined periods are *AckPeriod* and *DeletePeriod*. The *AckPeriod* should be set to at least the duration of two message transfers between two adjacent neighbors, *DeletePeriod* should be set to at least the duration of one message transfer time between two adjacent nodes multiplied by the number of clusterheads, which can be considered as the total duration of the flooding operation in the network.

- *ToutType*: Expiration type of the message determines the process which should be executed when the local clock reaches the *ToutExpireTime*. The *ToutTypes* are *AckTout* and *DeleteTout*. *AckTout* is used to remark that an *ACK* message is expected for the message. *DeleteTout* is used to remark that at the end of the *ToutExpireTime*, the entry which belongs to the message should be deleted from the *MessageInfoBox*.

5.1.1. Sending a Message

In CDS Flooding algorithm the behavior of the nodes which want to send a message changes according to the type of the sender node. When an ordinary node wants to send a message, it prepares the message and records it in its *MessageInfoBox*. The node sets *ToutExpireTime* to *AckPeriod* and sets *ToutType* to *AckTout*. It then sends the message to its clusterhead. When the sender node receives an *ACK* message before the *ToutExpireTime*, it sets the *ToutExpireTime* to *DeletePeriod* and sets *ToutType* to *DeleteTout*. If the *ACK* message is not received before the *ToutExpireTime*, the node sends the same message again. The node re-sends the message periodically until the *ACK* is received.

If the sender node is a clusterhead, it checks the members of its cluster. If the destination is in its cluster, then the node sets *ToutExpireTime* to *AckPeriod* and sets *ToutType* to *AckTout*. It then sends the message to the destination. If the destination is in another cluster, it prepares the message as a multicast message and records the message to its *MessageInfoBox*. It puts its *BLACK* colored neighbors to the *AckNeededNeighbors* list, sets *Count* to the size of the list, sets *ToutExpireTime* to *AckPeriod* and sets *ToutType* to *AckTout*. The *AckNeededNeighbors* list is also copied into the message body as *MulticastNeighbors*. Then the node floods the message to the network by multicasting the message. The sender node removes a node from its AckNeededNeighbors list upon receiving acknowledgement from the corresponding node. When the local clock reaches the *ToutExpireTime*, the node checks the *Count* variable if there are still *ACK* needed neighbors. If there are, the node resends the message to the neighbors which are still in *AckNeededNeighbors* list. When all the *ACK* messages are collected, the node sets the *ToutExpireTime* to *DeletePeriod* and sets *ToutType* to *DeleteTout*.

5.1.2. Receiving and Processing Messages

During the CDS Flooding algorithm, ordinary nodes are not involved in the flooding operation. An ordinary node accepts a message if and only if it is the destination of the message received. Moreover, in a CDS, an ordinary node can only receive a message from its clusterhead. Therefore, when an ordinary node receives a message, it first checks if the message is sent by the node's clusterhead. Then it checks if it is the destination of that message. If one of these conditions is false then the node ignores the message and does nothing. If both conditions are true, it processes the message.

When a clusterhead receives a message, it checks the following conditions.

- C1: The message is received for the first time and the node is the destination of the message
- C2: The message is received for the first time and the destination of the message is in the node's cluster
- C3: The message is received for the first time and the destination of the message is outside of the node's cluster
- C4: The message is already received before and the node is the destination of the message
- C5: The message is already received before and the destination of the message is in the node's cluster
- C6: The message is already received before and the destination of the message is outside of the node's cluster

If the condition C1 is true then the node records the message in its *MessageInfoBox*, sets the *ToutExpireTime* to *DeletePeriod*, sets *ToutType* to *DeleteTout*, sends an *ACK* message to the sender of the message and processes the message. If the message is suitable for the condition C2, then the node records it in its *MessageInfoBox*. The node sets *ToutExpireTime* to *AckPeriod* and sets *ToutType* to *AckTout*. Then it sends the message to the destination node and sends an *ACK* message to the sender. If condition C3 is true then the node prepares the message as a multicast message and records the message to its *MessageInfoBox*. It puts its *BLACK* colored neighbors to the *AckNeededNeighbors* list, sets *Count* to the size of the list, sets *ToutExpireTime* to *AckPeriod* and sets *ToutType* to *AckTout*. The *AckNeededNeighbors*

list is also copied into the message body as *MulticastNeighbors* in order to indicate which nodes should process the message. Then the node floods the message to the network as a multicast packet. If the one of the conditions C4, C5 or C6 is true, the node ignores the message an does nothing. If the received message is an *ACK* message, the node searches its *MessageInfoBox* and checks if the sender is in the *AckNeededNeighbors* list, if it is in the list then the node deletes the sender node from the *AckNeededNeighbors* list for that message.

When a clusterhead delivers a message which suits to the condition C1, it stops flooding the message. But the flooding operation continues in the network until the message is distributed to all clusterheads. If a node receives a message for more than once, it does not reply to this message. But in order to acknowledge the sender node, the echoes to the flooding messages are needed to be sent. If the sending node needs an *ACK* and receives nothing, there might be three possibilities:

1. The message isn't received by the destination
2. The message is received by the destination but the ACK message is lost in the network
3. The message is already received by the destination, therefore is ignored by the receiver

If an *ACK* message is not received before the *ToutExpireTime* in order to ensure the message delivery, the node sends a special type of message, *ForceMessage*, which forces the receiver to send an *ACK* message in any case. If a clusterhead receives a message which suits to one of the conditions C4, C5 or C6, it checks if the message is a *ForceMessage*. If the message is a *ForceMessage*, the node sends an *ACK* message to the sender of the message. The nodes which reply to a *FroceMessage* do not expect any acknowledgement for the replied message.

5.2. An Example Operation

We explain the example operation of the CDS Flooding Algorithm on top of the example CDS which can be seen in the Fig. 5.3 . A node which wants to send a message sends the message to the *BLACK* node in the cluster. For example nodes 1, 2 and 4 should send the outgoing message to node 3, nodes 10, 11, 12, 13 and 14 should send the outgoing message to the node 6 and so on.

- *Intra-cluster Communication Towards a Clusterhead:* When node 1 wants to send a message to node 3, it prepares the message and sends it to its clusterhead which is node 3. The node

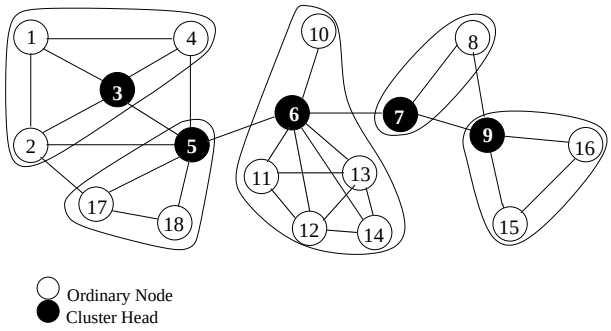

Figure 5.3. An Example CDS Based Clusters

3 receives the message and finds out that it is the destination. Therefore it does not flood the message, just sends an *ACK* message to the node 1 and finishes the operation. Routing of the message can be seen in Fig. 5.4.a

- *Intra-cluster Communication Between Ordinary Nodes:* When node 4 wants to send a message to node 2, it sends the message to its clusterhead which is node 3. Node 3 finds that the message is destined to a node which is in its cluster, therefore it does not flood the message to the CDS, instead, node 3 sends the message to the destination node, node 2, and finishes the operation. Routing of the message can be seen in Fig. 5.4.b

- *Inter-Cluster Communication from an Ordinary Node to a Clusterhead*: When node 2 wants to send a message to node 6 it sends the message to its clusterhead, node 3. When node 3 receive the message it finds out that the destination is outside of its cluster. Therefore it floods the message. Node 5 receives the flooded message and it checks its cluster, it sees that the destination is not in its cluster, therefore it floods the message too. Node 6 receives the message and finds out that it is the destination. Therefore it stops flooding and finishes the operation. Routing of the message can be seen in Fig. 5.4.c

- *Inter-cluster Communication Between Ordinary Nodes:* When node 1 wants to send a message to node 12, it sends the message to its clusterhead. The message reaches to the node 6 in the same fashion with the previous description. But this time node 6 finds out that the destination is in its cluster. Therefore it sends the message directly to the destination node, node 12, and stops flooding the message. Routing of the message can be seen in Fig. 5.5.a

- *Intra-cluster Communication from a Clusterhead to an Ordinary Node:* When node 3 wants to send a message to node 2 it checks its cluster members, it finds out that destination node

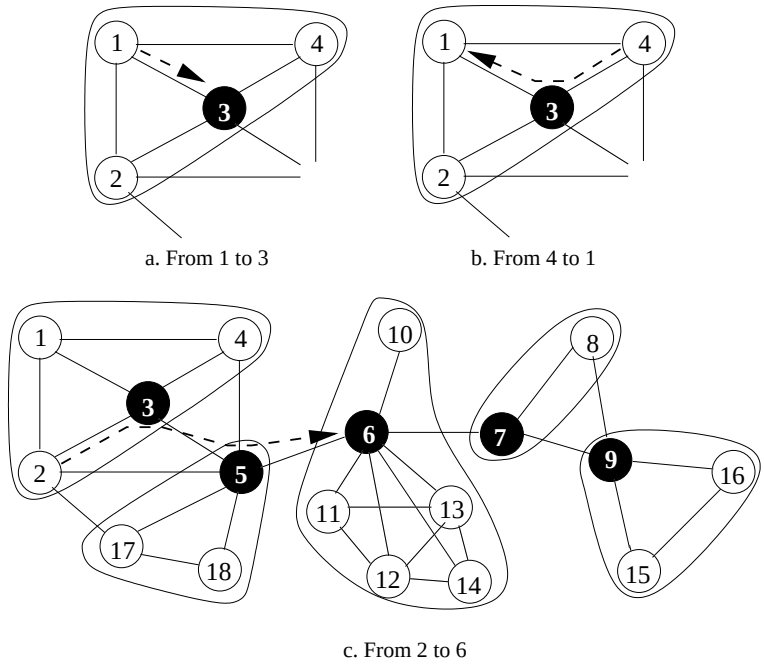

Figure 5.4. CDS Flooding Examples

is in its cluster. Therefore it directly sends the message to the destination and finishes its operation. Routing of the message can be seen in Fig. 5.5.b

- *Inter-cluster Communication from a Clusterhead to Any Type of Node:* When node 3 wants to send a message to node 6 it floods the message to the CDS. Node 5 receives the flooded message, it checks its cluster and finds out that the destination is not in its cluster. Therefore, node 5 continues to flood the message, then node 6 receives the message and finds out that it is the destination of the message. Routing of the message can be seen in Fig. 5.5.c

5.3. Analysis

Theorem 5.3.1. *Time complexity of the CDS Flooding Algorithm has a lowerbound of* $\Omega(2)$ *and an upperbound of* $O(n)$ *where* n *is the number of clusterheads in the network.*

Proof. The best case scenario of the algorithm is the intra-cluster communication. In the intra-cluster communication scenario the node sends the message to its clusterhead and the clusterhead redirects the message to the destination. In this case 2 message exchange is required, thus the lowerbound complexity of the CDS Flooding Algorithm is $\Omega(2)$. In the worst case scenario, source and destination nodes are located in the most distant clusters, and the

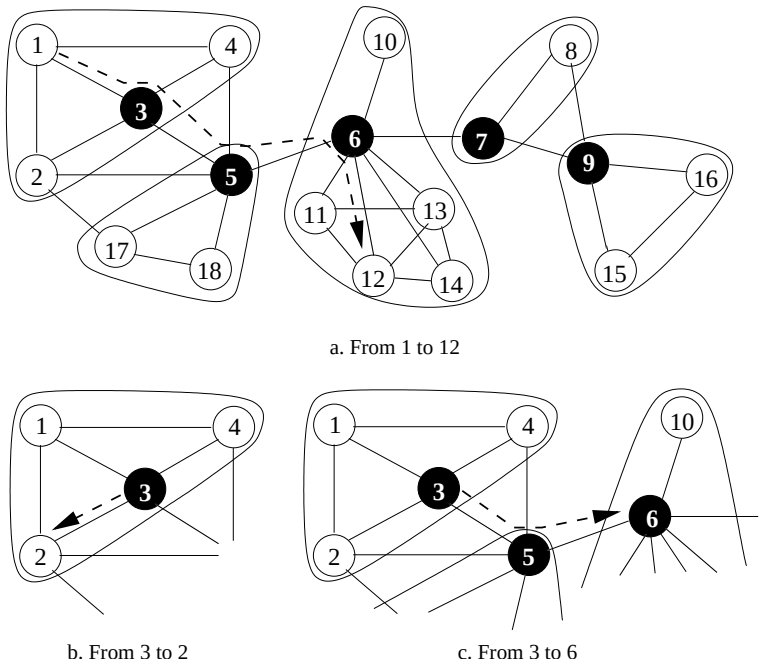

Figure 5.5. CDS Flooding Examples

clusterheads have at most 2 clusterhead neighbors. Source node sends the message to its clusterhead first, then the message is flooded to all the clusterheads by broadcasting the message sequentially $n-1$ times. Finally destination node's clusterhead sends the message to the destination node. Since the distribution of the message is sequential, the time complexity of the algorithm in the worst case can be expressed as $O(n)$ where n is the number of clusterheads in the network. The time complexity during the local iterations are ignorable compared to the messaging durations, therefore the time complexity of the local iterations are ignored during the time complexity analysis. □

Theorem 5.3.2. *Message complexity of the CDS Flooding Algorithm is $O(n)$ where n is the number of the clusterheads in the network.*

Proof. In the worst case, a message is flooded to all clusterheads in order to reach the destination. The message is first sent to the first clusterhead and then flooded to the CDS. A message is flooded to the network by using the broadcast messages. In the worst case, each clusterhead has maximum of two BLACK colored neighbors, therefore each time the message is flooded, it is received by only one clusterhead which receives the flooded message for the first time. Each pair of clusterheads communicate by the exchange of one broadcast message. Therefore the total messaging complexity of the algorithm is $O(n+1)$ in the worst case which

can be expressed as $O(n)$ in the big O notation where n is the number of clusterheads in the network. □

5.4. Results

We implemented the CDS Flooding Algorithm using C++ on top of the network simulator *ns2*. In order to get reliable results for the message transfer times, we used the same network scenarios for each test. We generated 8 random network scenarios for static and low speed dynamic graphs for the number of nodes as 20, 30, 40 and 50. We first constructed the CDSs with these scenarios using the modified TLCDSC Algorithm which is described in Section 5.1. Therefore, during our tests we assumed that we already have underlying CDS based clusters. We tested our algorithm for random message traffics for 0.5, 1, 2, 4 and 8 messages per second.

In our experiments we collected test results for the duration of the delivery of messages. The duration is measured between the creation of the message at the application level and the delivery to the destination node at the application level. We add a timestamp to the message body. When the target node receives the message it calculates the duration and stores the result. Each node sends a message to a random destination at a random time. We have collected data for a total of 2800 samples. We have generated 40 different scenarios for 5 message density, 4 number of nodes and 2 mobility parameters and we collected 70 sample result for each scenario. We classified the samples according to the number of nodes and message traffic densities.

In the dynamic graph experiments, random movements are generated for each simulation. For the low mobility scenarios, node speeds are limited from 1 m/s to 5 m/s. In dynamic graph experiments, we take into account only the experiments in which, nodes are moving but the neighborhoods of the nodes do not change.

The results for the static network experiments can be seen in Fig. 5.6. We expect that the message times won't be affected from the traffic density under the ideal conditions. But because of the message conflicts, the message times increase as the message traffic increases. We also expect that as the number of nodes increases the message times will also increase. In a larger network, our CDS will have more clusterheads. Therefore the messages will be flooded to a larger set of nodes. We can see the increase in the Fig. 5.6 as the number of nodes increases.

The test results for the low speed dynamic network can be seen in Fig. 5.7. The CDS

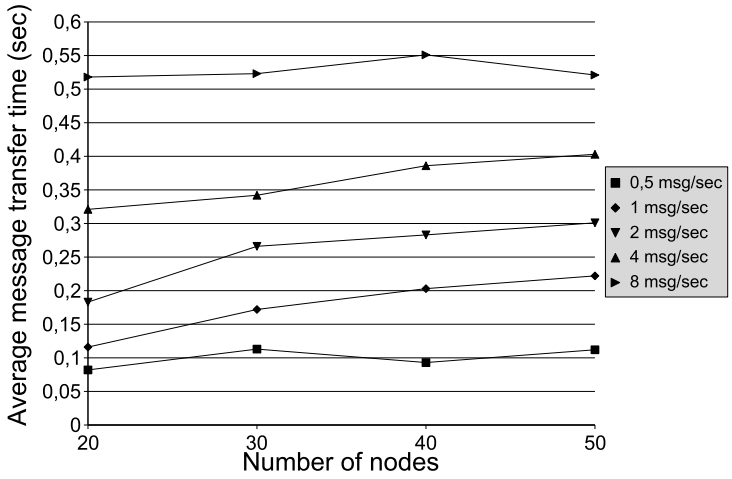

Figure 5.6. CDS Flooding Test Results for Static Network

Flooding Algorithm is independent from the mobility of the network. As it can be seen in the Fig. 5.7 the results are quite similar to the results of the static network. Both results vary between approximately 0.05 and 0.5 seconds. We can observe nearly the same behaviors against the number of nodes and the traffic load.

5.5. Discussions

The CDS Flooding Algorithm satisfies the expectations according to the experimental results. The limitations with the algorithm are generally caused by the conflicting messages. Under the high traffic load, the results became unstable. For traffic loads greater than 8 msg/sec, we cannot collect meaningful readouts. This is because when the traffic load exceeds a limit, the message conflicts increases and blocks the message traffic.

Another limitation that we encountered during the tests is the mobility of the network. During our experiments we cannot collect meaningful data for the high speed dynamic networks. In order to generate random traffic, we set the test duration between 200 and 2000 seconds. In high speed dynamic networks the neighborhoods change in such durations and one of our assumptions is not valid anymore. This limitation is not related to the CDS Flooding Algorithm, it is a result of our test scenarios. In ideal conditions, as the TLCDS Algorithm runs periodically before the neighborhoods change, the period is determined according to the speed of the nodes. Therefore this limitation can be ignored. Moreover, this problem should be

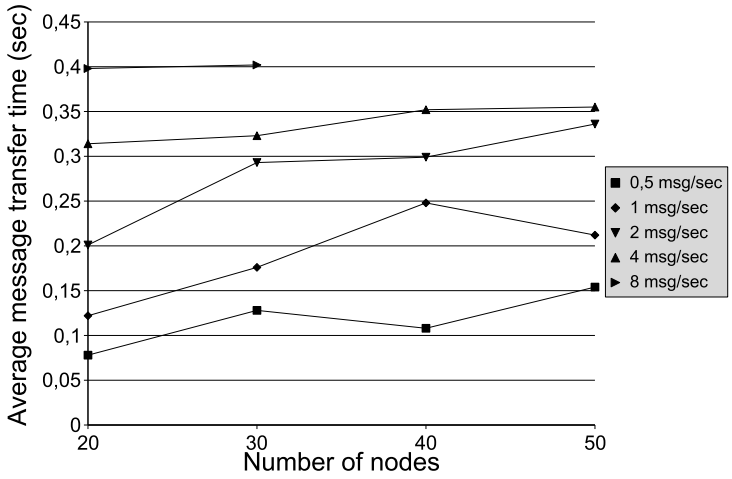

Figure 5.7. CDS Flooding Test Results for Low Speed Dynamic Network

left to the TLCDSC Algorithm as the CDS Flooding Algorithm assumes that the underlying CDS topology is managed by the underlying TLCDSC Algorithm independently from our CDS Flooding Algorithm. Therefore we didn't make tests for the high speed dynamic networks.

For the future work some modifications and optimizations can be applied to the CDS Flooding Algorithm in order to decrease message complexity. For example, some additional rules may be applied such that if two nodes are adjacent, they may communicate directly without sending the message to the clusterhead.

The CDS Flooding Algorithm shows us that our TLCDSC Algorithm may be easily applied to many applications. It also shows that TLCDSC Algorithm meets the expectations under an application which is using the resulting CDS.

CHAPTER 6

CONCLUSION

We proposed, designed and implemented three protocols in order to solve clustering and routing problems in MANETs. We implemented our protocols in two layers. The first layer is the clustering layer which divides the unstructured MANET into clusters. In this layer, clustering algorithms implemented are Connected Dominating Set Based Clustering (CDSC) Algorithm and Two Level Connected Dominating Set Based Clustering (TLCDSC) Algorithm. The CDSC Algorithm is focused to find minimal connected dominating sets in MANETs. The CDSC Algorithm is based on Wu's CDS Algorithm "(Wu and Li 2002)" which finds a crowded dominating set and prunes the MANET by eliminating redundant clusterheads. We added extra heuristics and pruning rules to the Wu's CDS Algorithm in order to reduce the size of the resulting CDS. The CDSC Algorithm also contributes to Wu's CDS Algorithm by informing the nodes about their clusterheads. At the end of the CDSC Algorithm, every node determines their clusterheads and are ready to implement any cluster based application. We describe the algorithm by using finite state machine design. We analyzed the time and message complexities of the algorithm. We tested the CDSC Algorithm on top of the network simulator $ns2$. We measured the time to build the clusters, cluster sizes, number of clusterheads per cluster and coefficient of variances in the cluster sizes. We tested our algorithm in different scenarios in which the number of nodes varies between 20 and 50, and mobilities vary between 0 m/s and 10 m/s. The implementation results confirm with the theoretical analysis and show that the CDSC Algorithm is scalable in terms of the size of the network and the mobility. The second algorithm which is designed and implemented in the clustering layer is the TLCDSC Algorithm. At the end of the evaluation period of the CDSC Algorithm we realized that in some situations in which inter-cluster communication is the main part of the entire communication in the network, the cluster sizes are needed to be larger than the sizes of clusters that CDSC Algorithm builds. In order to build more crowded multi-level clustered environments, we improved the CDSC Algorithm so that it runs recursively on top of the resulting CDS of the previous recursion. We implemented the TLCDSC Algorithm to build two level clusters. We extended the finite state

machine of the CDSC Algorithm and described the algorithm in detail. We analyzed the time and message complexities of the algorithm and implemented on top of the network simulator $ns2$. We tested the TLCDSC Algorithm with the same scenarios which are used during the tests of the CDSC Algorithm and compared the results of the two clustering algorithms. As it is expected, the first level clustering results are similar in both algorithms, but the TLCDSC Algorithm extracted a second level CDS which consists of a subset of CDS elements of the first level clustering. The test results of the TLCDSC Algorithm are very similar to the expected behavior and confirm with the theoretical analysis. They also show that the CDSC Algorithm can easily be extended to a multi-level hierarchical clustering algorithm.

The second layer protocol which is proposed in this thesis is a Flooding based routing algorithm for clustered MANETs (CDS Flooding). The CDS Flooding Algorithm assumes that the network is clustered by an underlying clustering algorithm. It also assumes that the clusterheads create a backbone between the clusters. In order to meet the assumptions, we used the TLCDSC Algorithm as the clustering algorithm. We build the CDS based clusters and design the CDS Flooding Algorithm. The CDS Flooding Algorithm is based on the idea of reverse-path multicast algorithms. In CDS Flooding Algorithm, the flooding operation takes place between the clusterhead nodes instead of the whole network. Therefore, the algorithm reduces the flooding complexity directly by using a smaller set of nodes. We analyzed, implemented and tested the CDS Flooding Algorithm by using the network simulator environment $ns2$. The test results met our expectations and showed that the CDS Flooding algorithm can be used in clustered MANETs especially if the clustering algorithm is based on connected dominating sets. The CDS Flooding Algorithm also showed that the TLCDSC Algorithm is a very suitable algorithm for the development of distributed applications which are designed to run on top of clustered environments.

In summary, we proposed three algorithms for the distributed applications in mobile ad hoc networks. This study shows that the proposed architecture may be preferable in implementing distributed applications which require CDS based clustered networks with a built-in routing capabilities. The test results show that our algorithms are scalable in terms of the number of nodes in the MANET. The clustering algorithms perform similarly in different test scenarios in which the number of nodes varies. Cluster sizes are not affected by the size of the network. The CDS Flooding Algorithm also performs similarly in the different test scenarios. Test results show that end-to-end message transfer times are proportional to the size of the

CDS which is built at the end of the TLCDSC Algorithm, instead of the number of nodes in the entire network. The protocols implemented are also preferable for static and low speed networks, but they are not very suitable for the high speed dynamic networks. Test results show that performance values of the three algorithms are nearly constant in the different mobility scenarios as long as the neighborhoods remain still.

Our algorithms have better performance than many other similar algorithms such as Wu's CDS Algorithm in crowded MANETs in which the degrees of the nodes do not exceed the specified values in the test results.

We are planning to make enhancements in the CDS Flooding Algorithm and implement a proactive routing algorithm for MANETs as a future work in order to speed up the algorithms. We are planning to get help from a MAC level protocol in order to decrease the number of conflicting messages which can be considered as the main problem encountered during the tests. We are also planning to adapt the proposed communication architecture to the Wireless Sensor Networks (WSN), which is an important research area in today's wireless technology. We are planning to enhance the algorithms by adding some critical wireless sensor node constraints such as energy levels, signal strengths and lifetime of the nodes.

REFERENCES

M. Ahuja and Y. Zhu. A distributed algorithm for minimum weight spanning trees based on echo algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 2–8, 1989.

B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 230–240, 1987.

D. Baker and A. Ephremides. The architectural organization of a mobile radio network via a distributed algorithm. *IEEE Transactions*, 29:1694–1701, 1981.

S. Banerjee and S. Khuller. A clustering scheme for hierarchical routing in wireless networks. Technical Report CS-TR-4103, University of Maryland, 2000.

S. Basagni, I. Chlamtac, V.R. Syrotiuk, and B.A. Woodward. A distance routing effect algorithm for mobility (dream). In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking, MobiCom98*, pages 76–84. ACM Press, 1998.

G. Chen, F.G. Nocetti, J.S. Gonzalez, and I. Stojmenovic. Connectivity based k-hop clustering in wireless networks. In *Proceedings of the 35th Annual Hawaii International Conference*, pages 2450–2459, 2002.

Y.P. Chen and A.L. Liestman. Approximating minimum size weakly-connected dominating sets for clustering mobile ad hoc networks. In *Proceedings of 3rd ACM International Symposium Mobile Ad Hoc Net. and Comp.*, pages 165–72, 2002.

Y.P. Chen and A.L. Liestman. A zonal algorithm for clustering ad hoc networks. *International Journal of Foundations of Computer Science*, pages 305–322, 2003.

Y.P. Chen, A.L. Liestman, and J. Liu. Clustering algorithms for ad hoc wireless networks. *Nova Science Publishers*, 2004.

C.C. Chiang, H.K. Wu, W. Liu, and M. Gerla. Routing in clustered multihop, mobile wireless networks with fading channel. In *Proceedings of IEEE Singapore International Conference on Networks SICON'97*, pages 197–211. IEEE, 1997.

D. Cokuslu, K. Erciyes, and O. Dagdeviren. A dominating set based clustering algorithm for mobile ad hoc networks. In *Proceedings of International Conference on Computational Science 2006*, volume LNCS 3991, pages 571–578. Springer-Verlag, 2006.

O. Dagdeviren, K. Erciyes, and D. Cokuslu. A merging clustering algorithm for mobile ad hoc networks. In *Proceedings of International Conference on Computational Science and Its Applications 2006*, volume LNCS 3981, pages 681–690. Springer-Verlag, 2006.

F. Dai and J. Wu. An extended localized algorithm for connected dominating set formation in ad hoc wireless networks. *IEEE Transactions On Parallel and Distributed Systems*, 15(10), 2004.

B. Das and V. Bharghavan. Routing in ad-hoc networks using minimum connected dominating sets. In *IEEE International Conference on Communications*, volume 1, pages 376–380, 1997.

B. Das, R. Sivakumar, and V. Bhargavan. Routing in ad hoc networks using a spine. In *Proceedings of Sixth IEEE Int. Conf. Computers Comm. and Networks*, pages 1–20, 1997.

M.K. Denko and H. Lu. An aodv-based clustering and routing scheme for mobile ad hoc networks. In *Ad-Hoc Networking*, pages 83–97. Springer Boston, 2006.

K. Fall and K. Varadhan. The ns manual. available at http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf, 2006.

R.G. Gallagher, P.A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.

J.A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Proceedings 34th Annual Symposium on Foundations of Computer Science*, pages 659–668, 1993.

M. Gerla and J.T.C. Tsai. Multicluster, mobile, multimedia radio network wireless networks. *ACM/Baltzer Journal of Wireless Networks*, 1(3):255–265, 1995.

R.P. Grimaldi. Discrete and Combinatorial Mathematics, An Applied Introduction. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0201896842.

S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 2450–2459. Springer-Verlag, 1998.

M. Halvardsson and P. Lindberg. Reliable group communication in a military ad hoc network. Technical report, Vaxjo University, 2004.

T.W. Haynes, S.T. Hedetniemi, and P.J. Slater. *Domination in graphs: Advanced Topics*. Dekker, 1978. ISBN 0824700341.

M. Joa-Ng and I.T. Lu. A peer-to-peer zone-based two-level link state routing for mobile ad-hoc wireless networks. *IEEE Journal on Selected Areas in Communications*, pages 1415–1425, 1999.

D.B. Johnson and D.A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

L. Kleinrock and K. Faroukh. Hierarchical routing for large networks. *Computer Networks*, 1: 155–174, 1997.

Y.N. Lien. A new node-join-tree distributed algorithm for minimum weight spanning trees. In *Proceedings of the 8th International Conference on Distributed Computing System*, pages 334–240, 1988.

C. Liu and J. Kaiser. A survey of mobile ad hoc network routing protocols. *University of Ulm Technical Report Series*, (2003-08), 2005.

H. Liu, Y. Pan, and C. Jiannong. An improved distributed algorithm for connected dominating sets in wireless ad hoc networks. In *Proceedings of Parallel and Distributed Processing and Applications: Second International Symposium*, page 340, 2004.

J. Mingliang, J. Li, and Y. C. Tay. Cluster based routing protocol (cbrp). 1999.

S. Murthy and J.J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2):183–197, 1996.

T. Ohta, S. Inoue, and Y. Kakuda. An adaptive multihop clustering scheme for highly mobile ad hoc networks. In *Proceedings of 6th ISADS 03*, pages 2450–2459, 2003.

OPNET and Technologies Corporation. Opnet users manual. http://www.opnet.com/, 2006.

G. Pei, A. Iwata, C. Chiang, M. Gerla, and T. Chen. Scalable routing strategies for ad-hoc wireless networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1369–1379, 1999.

G. Pei, M. Gerla, and T.W. Chen. Fisheye state routing in mobile ad hoc networks. In *ICDCS Workshop on Wireless Networks and Mobile Computing*, pages D71–D78, 2000.

C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *Conference on Communications Architectures, Protocols and Applications, SIGCOMM94*, pages 234–244. ACM, 1994.

C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications*, pages 90–100. IEEE, 1999.

E.M. Royer and C.K. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Magazine Personal Commun.*, 8:46–55, 1999.

Scalable and Network Technologies Corporation. Qualnet user's guide. http://www.scalable-networks.com/, 2005.

S. Srivastava and R.K. Ghosh. Distributed algorithms for finding and maintaining a k-tree core in a dynamic network. *Information Processing Letters*, 88(4):187–194, 2003.

I. Stojmenovic, M. Seddigh, and J. Zunic. Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 13:14–25, 2002.

P.J. Wan, K. M. Alzoubi, and O. Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. *Mobile Networks and Applications*, 9(2):141–149, 2004.

D. West. Introduction to Graph Theory. Prentice Hall, second edition, 2001. ISBN 0130144002.

J. Wu. Extended dominating-set-based routing in ad hoc wireless networks with unidirectional links. *IEEE Trans. Parallel and Distributed Systems*, 9(3):189–200, 2002.

J. Wu and H. Li. A dominating-set-based routing scheme in ad hoc wireless networks. *Telecommunication Systems*, 18(1-3):13–36, 2002.

Z. Xu and S. Dai. Hierarchical routing using link vectors. In *Proceedings of Infocom*, pages 702–710, 1998.

X. Yan, Y. Sun, and Y. Wang. A heuristic algorithm for minimum connected dominating set with maximal weight in ad hoc networks. In *Proceedings of Grid and Cooperative Computing: Second International Workshop*, pages 719–722, 2003.

X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the Principles of Advanced and Distributed Simulation Conference 1998*, pages 154–161, 1998.

APPENDIX A

SIMULATION SETUP

The protocols which are described in this thesis are implemented and tested on top of the network simulation environment *ns2* "(Fall and Varadhan 2006)". There are lots of other network simulation environments such as *GloMoSim* "(Zeng et al. 1998)", *QualNet* "(Scalable and Corporation 2005)", and *OPNET* "(OPNET and Corporation 2006)". They all provide simulation platforms for MANETs. Before the implementation of our protocols, we examined these simulation environments and chose *ns2* as the simulation environment for our protocols. We find out that *ns2* is broadly used in the academic studies in order to simulate various networks. In contrast, *OPNET* and *QualNet* are mostly considered to be commercial products. *GloMoSim* is available for free of charge for the academic applications, but it does not have a wired network support. The *ns2* can be downloaded free of charge, it has both wired and wireless network support and can be compiled on different operating systems. The source code can be modified and some protocols can be added easily. Moreover, many wireless extensions have been contributed from the UCB Daedelus, the CMU Monarch projects and Sun Microsystems "(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. *MobileNode* is the basic *ns2 Node* object with added functionalities like ability to move, transmit and receive on a channel which allows it to be used to create mobile, wireless simulation environments. During our implementations, we used *MobileNode* class with its interfaces which are required to send-receive message, create a list of adjacent neighbors etc. The network components such as link layer and MAC layer components are created together in *OTcl* language "(Fall and Varadhan 2006)".

The protocol stack is implemented in *C++*. We firstly modified the existing *UDP* and *MAC* layer implementations to communicate with our protocols. *UDP_CDS* class header can be seen in Figs. A.1 and 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.

The class header of the *CDSC* Algorithm can be seen in Figs. A.4, A.5 and A.6, class header of the TLCDSC Algorithm can be seen in Figs. A.7, A.8, A.9 and A.10 and class header of the CDS Flooding Algorithm can be seen in Figs. A.11, A.12 and A.13.

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 are 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 set the surface area in order to ensure the density constraints, therefore the surface areas varied relative to the number of nodes. We generated simulation surfaces varied from 850m × 850m to 850m × 3000m. Static, low and high mobility scenarios are generated with the respective node speeds 0m/s, 1.0m/s to 5.0m/s and 5.0m/s to 10.0m/s. Pause time is set to 0.

After protocols are implemented in *C++*, the scenario files are written in *Otcl*. A complete scenario file is given in Figs. A.14, A.15 and A.16.

```
//
// Author:    Deniz Cokuslu
// File:      udp-cds.h
// Date:      10/09/06 (for CDSC Algorithm)
//
#ifndef ns_udp_cds_h
#define ns_udp_cds_h
#include "udp.h"
#include "ip.h"
#include "cds_protocol.h"
// Message Header Structure
struct hdr_cdsClus {
    int ack;                // is it ack packet?
    int seq;                // sequence number
    int nbytes;             // bytes for pkt
    double time;            // current time
    int32_t source;         // source address
    int32_t destination;    // next hop destination address
    int32_t real_destination; //final destination address
    int32_t 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 message_propagation_type;//Message Propagation Type
    int protocol_type;      //Type of the protocol, CDSC, TLCDSC or CDSFlooding
    int priority;           //Priority of the message
//Added for CDSC Algorithm
    int  D_neighbors;      //A list of neighbors of the sending node
    int  D_color;          //Color of the sender
    int  D_degree;         //Degree of the sender
    int  *multicast_neighbors;//A list of neighbors to which the message is sent
//Added for CDS Flooding Algorithm
    double message_id;     //Unique id of the message
    double sentTime;       //sent time of the message
// Packet header access functions
    static int offset_;
    inline static int& offset() { return offset_; }
    inline static hdr_cdsClus* access(const Packet* p) {
    return (hdr_cdsClus*) p->access(offset_);
        }
};
```

Figure A.1. Header File of the udp-cds Class

```
// UdpCDSClusAgent Class definition
class UdpCDSClusAgent : public UdpAgent {
public:
    UdpCDSClusAgent();
    UdpCDSClusAgent(packet_t);
    Application* getApplication();
    virtual int supportCDSClus() { return 1; }
    virtual void enableCDSClus() { support_cdsclus_ = 1; }
    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*);
    void notification(int src, double message_id);
};
```

Figure A.2. Header File of the udp-cds Class

```
void Mac802_11::sendACK(int dst, double message_id)
{
    Packet *p = Packet::alloc();
    hdr_cmn* ch = HDR_CMN(p);
    struct ack_frame *af = (struct ack_frame*)p->access(hdr_mac::offset_);
    assert(pktCTRL_ == 0);
    ch->uid() = 0;
    ch->ptype() = PT_MAC;
    ch->size() = phymib_.getACKlen();
    ch->iface() = -2;
    ch->error() = 0;
    bzero(af, MAC_HDR_LEN);
    //Modified for CDSFlooding Algorithm
    af->src=addr();
    af->message_id = message_id;
.
.
.
    UdpMergClusAgent* udp=(UdpMergClusAgent*)UDPMCContainer::instance().getElementById(my_addr);
    //Modified for CDSFlooding Algorithm
    udp->notification(source, message_id);
.
.
.
```

Figure A.3. Modification Made on mac-802_11.cc

```
// Author:    Deniz Cokuslu
// File:      cdsclus-app.h
// Date:   10/09/06 (for CDSC Algorithm)

#include "timer-handler.h"
#include "packet.h"
#include "app.h"
#include "mobilenode.h"
#include "timeque.h"

class CDSClusApp;

// CDS Clustering Protocol uses this timer to
// schedule next app data packet transmission time
class CDSTimer:public TimerHandler
{
  public:
    CDSTimer(CDSClusApp* 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:
    CDSClusApp* t_;
    double tout_seconds;
    int tout_state;
};

// CDSClus Application Class Definition
class CDSClusApp:public Application
{
  public:
    CDSClusApp();

    //Sends a cds clustering packet
    void send_cdsclus_pkt(int source,int destination,int message_type, int neighbors[],
                          int color, int degree, int packet_type);
    //Generic send method
    void send_msg(int nbytes,const char* msg);
    void recordNode(MobileNode *node);
    void init();
    void externalInit();
    void init_fsm();
    //Sets the methods to the FSM transitions
    void CDSClusApp::fsm_jump_next_state(void *message);
    //Generic receive message method
    virtual void recv_msg(int nbytes, const char *msg = 0);
```

Figure A.4. CDSClusApp Class Header File

```
//Global variables
    int D_color;                                //Color of the node
    int D_count;    //Count of the message sent, used to collect ACK messages
    int D_degree;                               //Number of node's neighbors
    int D_neighbor_matrix[N_NODES][N_NODES];    //Node's 2 hop neighborhood list
    int D_neighbors_colors[N_NODES];            //List of node's neighbors colors
    int D_neighbors_degrees[N_NODES];           //List of node's neighbors' degrees
    int D_message_sent_neighbors[N_NODES];      //List of nodes to which the multicast message
                                                //is sent
    int D_lost_message_type;                    //Type of the ACK needed message
    int myNeighbors[N_NODES];                   //list of node's neighbors
    int D_my_cluster_head;                      //Node's clusterhead

protected:
    int command(int argc, const char*const* argv);
    void start();  // Start method of the protocol
    void stop();   // Stop method of the protocol

//States of the Finite State Machine
    #define D_IDLE 0
    #define D_CHK_NODES 1
    #define D_CHK_DOMINATION 2
    #define D_CHK_CH 3 //Check Cluster Head

//Message types
    #define D_Period_TOUT 0
    #define D_Neighbor_REQ 1
    #define D_Neighbor_LST 2
    #define D_Color_REQ 3
    #define D_Color_RES 4
    #define D_Cluster_REQ 5
    #define D_Cluster_RES 6

//Colors
    #define D_UNDEFINED_COLOR -1
    #define D_WHITE 0
    #define D_BLACK 1
    #define D_GRAY  2

//Timeout constants
    #define D_tout_period 0.010
    #define D_tout_timer 0.200

//FSM Methods                   State - Message Received
    int Act00(void *message);//IDLE  - D_Period_TOUT
    int Act01(void *message);//IDLE  - D_Neighbor_REQ
    int Act03(void *message);//IDLE  - D_Color_REQ
    int Act05(void *message);//IDLE  - D_Cluster_REQ
    int Act11(void *message);//D_CHK_NODES - D_Neighbor_REQ
```

Figure A.5. CDSClusApp Class Header File (con.)

```
    int Act12(void *message);//D_CHK_NODES - D_Neighbor_LST
    int Act13(void *message);//D_CHK_NODES - D_Color_REQ
    int Act15(void *message);//D_CHK_NODES - D_Cluster_REQ
    int Act21(void *message);//D_CHK_DOMINATION - D_Neighbor_REQ
    int Act23(void *message);//D_CHK_DOMINATION - D_Color_REQ
    int Act24(void *message);//D_CHK_DOMINATION - D_Color_RES
    int Act25(void *message);//D_CHK_DOMINATION - D_Cluster_REQ
    int Act31(void *message);//D_CHK_CH - D_Neighbor_REQ
    int Act33(void *message);//D_CHK_CH - D_Color_REQ
    int Act35(void *message);//D_CHK_CH - D_Cluster_REQ
    int Act36(void *message);//D_CHK_CH - D_Cluster_RES
    int ActNA(void *message);//Method Not Applicable

//CDS Application methods
    //Used to multicast a message to a list of neighbors
    int multicast_message(int id, int type, int *neighbors, int color, int degree);
    //Returns the degree of the node
    int getMyDegree(int *my_neighbors, int numberOfNodes);
    //Finishes the execution of the algorithm
    int finish_phase(int id, int color);
    //Selects an appropriate clusterhead if possible
    int selectMyClusterHead();
};
```

Figure A.6. CDSClusApp Class Header File (con.)

```
// Author:    Deniz Cokuslu
// File:      cdsclus-app.h
// Date:   28/03/07 (for TLCDSC Algorithm)

#include "timer-handler.h"
#include "packet.h"
#include "app.h"
#include "mcfque.h"
#include "mobilenode.h"
#include "timeque.h"

class TLCDSCApp;

// TLCDSC Protocol uses this timer to
// schedule next app data packet transmission time
class TLCDSCTimer : public TimerHandler
{
   public:
    TLCDSCTimer(TLCDSCApp* 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:
    TLCDSCApp* t_;
    double tout_seconds;
    int tout_state;
};

// TLCDSC Application Class Definition
class TLCDSCApp : public Application
{
   public:
    TLCDSCApp();
    void send_cds_pkt(int source,int destination, int message_type, int *neighbors,
                      int color, int degree, int *multicast_neighbors, int packet_type);
    hdr_tlcdsc generate_packet(int root_source, int source, int destination,
                               int final_destination, double message_id,
                               int message_type, int *neighbors, int color,
                               int degree, int multicast_neighbors[N_NODES], int packet_type,
                               int protocol_type, double sentTime);

    void send_tlcdsc_pkt(hdr_tlcdsc mh_buf);
    void send_msg(int nbytes,const char* msg);
    void notification(int source, double message_id);
    virtual void upper_notification(int ntf_src, double mesasge_id);
```

Figure A.7. TLCDSCApp Class Header File

```
    void recordNode(MobileNode *node);
    void init();
    void externalInit();
    void init_fsm();
    void TLCDSCApp::fsm_jump_next_state(void *message);

    virtual void recv_msg(int nbytes, const char *msg = 0); // (Sender/Receiver)

//Level 1 Global variables
    int D_color;                            //Level 1 Color of the node
    int D_count;    //Count of the message sent, used to collect ACK messages
    int D_degree;                           //Number of node's Level 1 neighbors
    int D_neighbor_matrix[N_NODES][N_NODES];//Node's 2 hop Level 1 neighborhood list
    int D_neighbors_colors[N_NODES];        //List of node's Level 1 neighbors colors
    int D_neighbors_degrees[N_NODES];       //List of node's Level 1 neighbors' degrees
    int D_message_sent_neighbors[N_NODES];  //List of nodes to which the multicast message
                                            //is sent
    int D_lost_message_type;                //Type of the ACK needed message
    int myNeighbors[N_NODES];               //list of node's Level 1 neighbors
    int D_cluster_members[N_NODES];         //A list of nodes which are in the node's cluster
                                            //if the node is a clusterhead
    int D_my_cluster_head;                  //Node's clusterhead

//Level 2 Level Global Variables
    int D_degree_L2;                        //Number of node's Level 2 neighbors
    int D_neighbor_matrix_L2[N_NODES][N_NODES];//Node's 2 hop Level 2 neighborhood list
    int D_neighbors_colors_L2[N_NODES];     //List of node's Level 2 neighbors colors
    int D_neighbors_degrees_L2[N_NODES];    //List of node's Level 2 neighbors' degrees
    int myNeighbors_L2[N_NODES];            //list of node's Level 1 neighbors
    int D_color_L2;                         //Level 2 Color of the node
    int D_multicast_neighbors[N_NODES];     //Multicasted neighbors list

protected:
    int command(int argc, const char*const* argv);
    void start();       // Start sending data packets (Sender)
    void stop();        // Stop sending data packets (Sender)

//FSM States
//Level 1 States
#define D_IDLE 0
#define D_CHK_NODES 1
#define D_CHK_DOMINATION 2
#define D_CHK_CH 3 //Check Cluster Head

//Level 2 States
#define D_WAIT_BLACK 4 //A Black node waits for its GRAY colored neighbors' permanent colors
#define D_CHK_NODES_L2 5
#define D_CHK_DOMINATION_L2 6
#define D_CHK_CH_L2 7 //Check Cluster Head
#define D_WAIT_MEMBERSHIP_ACK 11
```


Figure A.8. TLCDSCApp Class Header File (con.)

```
//End States
#define D_WHITE_STATE 8
#define D_BLACK_STATE 9
#define D_RED_STATE 10

//Message types

//Level 1 Messages
#define D_Period_TOUT 0
#define D_Neighbor_REQ 1
#define D_Neighbor_LST 2
#define D_Color_REQ 3
#define D_Color_RES 4
#define D_Cluster_REQ 5
#define D_Cluster_RES 6

//Level 2 Messages
#define D_Black_REQ 7
#define D_Black_RES 8
#define D_Neighbor_REQ_L2 9
#define D_Neighbor_LST_L2 10
#define D_Color_REQ_L2 11
#define D_Color_RES_L2 12
#define D_Cluster_REQ_L2 13
#define D_Cluster_RES_L2 14
#define D_Cluster_MBR_INF 15
#define D_Cluster_MBR_ACK 16

//Level 1 Colors
#define D_UNDEFINED_COLOR -1
#define D_WHITE 0
#define D_BLACK 1
#define D_GRAY  2

//Level 2 Colors
#define D_BLUE 3 // Gray of Level2
#define D_RED 4  //Supernode

//Timer constants
#define D_tout_period 0.010
#define D_tout_timer 0.200

//FSM Methods
int ActNA(void *message);   //Not Applicable
                            //  STATE   - Received Message Type
int ActX1(void *message);   //Any State - D_Neighbor_REQ
int ActX3(void *message);   //Any State - D_Color_REQ
int ActX5(void *message);   //Any State - D_Cluster_REQ
```

Figure A.9. TLCDSCApp Class Header File (con.)

```
int ActX7(void *message);   //Any State - D_Black_REQ
int ActX9(void *message);   //Any State - D_Neighbor_REQ_L2
int ActX11(void *message);  //Any State - D_Color_REQ_L2
int ActX13(void *message);  //Any State - D_Cluster_REQ_L2
int ActX15(void *message);  //Any State - D_Cluster_MBR_INF

int Act00(void *message);   //D_IDLE - D_Period_TOUT
int Act12(void *message);   //D_CHK_NODES - D_Neighbor_LST
int Act24(void *message);   //D_CHK_DOMINATION - D_Color_RES
int Act36(void *message);   //D_CHK_CH - D_Cluster_RES
int Act48(void *message);   //D_WAIT_BLACK - D_Black_RES
int Act510(void *message);  //D_CHK_NODES_L2 - D_Neighbor_LST_L2
int Act612(void *message);  //D_CHK_DOMINATION_L2 - D_Color_RES_L2
int Act714(void *message);  //D_CHK_CH_L2 - D_Cluster_RES_L2
int Act1116(void *message); //D_WAIT_MEMBERSHIP_ACK - D_Cluster_MBR_ACK

//TLCDSCApp methods

//Broadcasts the message
int broadcast_message(int id, int type, int *neighbors, int color, int degree);

//Multicasts the message to the specified list of neighbors
int multicast_message(int id, int type, int *neighbors, int *multicast_neighbors,
                      int color, int degree);

//Get the node's degree
int getMyDegree(int *my_neighbors, int numberOfNodes);

//Finises the application
int finish_phase(int id, int color, int color_L2, int state);

//Selects an appropriate clusterhead if possible
int selectMyClusterHead();
};
```

Figure A.10. TLCDSCApp Class Header File

```
// Author:    Deniz Cokuslu
// File:      cds-flooding.h
// Date:   07/03/07 (for cds flooding)

#include "timer-handler.h"
#include "packet.h"
#include "app.h"
#include "mobilenode.h"
#include <time.h>

class CDSFloodingApp;

//t_message_info_box is a structure that holds
// message_id: unique message id
// ack_needed_neighbors: From who the node is waiting the ACK
// count: The number of neighbors to which the message is sent
// expire_time: When the ACK will be expired
// message: the message object
// tout_type: type of the expiration, this variable
//            is either Delete_TOUT or ACK_TOUT
//            which indicates the action to be done
//            when the time as come.
typedef struct t_message_info_box {
    double message_id;
    int ack_needed_neighbors[N_NODES];
    int count;
    double expire_time;
    hdr_tlcdsc message;
    int tout_type;
}t_message_info_box;

// CDSFlooding Protocol uses this timer to
// schedule next app data packet transmission time
class CDSFloodingTimer : public TimerHandler {
 public:
    CDSFloodingTimer(CDSFloodingApp* 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:
    CDSFloodingApp* t_;
    double tout_seconds;
    int tout_state;
};
```

Figure A.11. CDSFloodingApp Class Header File

```
// Application Class Definition
class CDSFloodingApp : public Application
{
 public:
    CDSFloodingApp();

    //CDSFlooding packet send method
    void send_cds_flooding_pkt(hdr_tlcdsc mh_buf);

    //Packet generating method
    hdr_tlcdsc generate_packet(int root_source, int source, int destination,
                               int final_destination, double messageId,
                               int message_type, int *multicast_neighbors,
                               int packet_type, double sentTime);

    //Generic send message method
    void send_msg(int nbytes,const char* msg);

//Constants
    #define D_tout_timer 0.200
    #define DELETE_PERIOD 50    //delete messages from the message_info_box in this period
    #define ACK_PERIOD 0.800    //if ACK is not received in this time interval then expire
    #define MSG_BUFFER_SIZE 1000//length of the message_info_box

//Type of the timeouts
    #define DELETE_TOUT 0
    #define ACK_TOUT 1

    #define Normal_MSG 0
    #define Forced_MSG 1//Forces the receiver to reply the message even it is already received

    //
    void notification(int source);
    virtual void upper_notification(int ntf_src, double message_id);
    void recordNode(MobileNode *node);
    void init();
    //Unique message id generator
    double message_id_generator();
    //Records the message into the message_info_box
    int recordMessage(double message_id, int *ack_needed_neighbors, int count,
                      double expire_time, hdr_tlcdsc *message, int tout_type);
    void externalInit();
    //finds the index of the specified message in the message_info_box
    int findMessage(double message_id);
    //Process message is the heart of the CDSFlooding algorithm
    void processMessage(void *message);
    //Generic receive message method
    virtual void recv_msg(int nbytes, const char *msg = 0);
```

Figure A.12. CDSFloodingApp Class Header File (con.)

```
    int D_color;            //Color of the node
    int *D_neighbor_matrix; //List of node's 2 hop neighbors
    int *D_neighbors_colors;//List of nodes' neighbors' colors
    int *myNeighbors;       //List of node's neighbors
    int *D_cluster_members; //List of members of the cluster if the node is a clusterhead
    int D_my_cluster_head;  //Id of the node's clusterhead
    t_message_info_box message_info[MSG_BUFFER_SIZE];//message_info_box

   protected:
    int command(int argc, const char*const* argv);
    void start();       // Start sending data packets (Sender)
    void stop();        // Stop sending data packets (Sender)
};
```

Figure A.13. CDSFloodingApp Class Header File (con.)

```
# ======================================================================
# 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)             20                         ;# number of mobilenodes
set val(rp)             DumbAgent                  ;# routing protocol
set val(mp)             "../../ns-2.28/indep-utils/cmu-scen-gen/setdest/CDSFlooding20_"
#set val(mrequest)       "../mrequests/mrequest10M"
set val(start_time)     0
set val(mmapp_end_time)           100.56
set val(cds_flooding_start_time)  101.00
set val(cds_flooding_end_time)  2000.00
set val(end_time)          2000.56
set val(halt_time)          2000.57
set val(tr_file)            out20lt.tr
set val(log_time1)          1.751763
set val(log_time2)          5.0
set val(log_time3)          8.10
# ======================================================================
# Main Program
# ======================================================================

#
# Initialize Global Variables
#
set ns_ [new Simulator]
set tracefd [open $val(tr_file) w]
$ns_ trace-all $tracefd
# set up topography object
set topo [new Topography]
#20 nodes
$topo load_flatgrid 850 950 #Surface area width * height
#
# Create God
#
set god_ [create-god $val(nn)]
```

Figure A.14. An example Scenario File

```
#
#  Create the specified number of mobilenodes [$val(nn)] and "attach" them
#  to the channel.
#  Here all the nodes
# configure node
        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 $topo \
             -agentTrace OFF \
             -routerTrace OFF \
             -macTrace OFF \
             -movementTrace ON
    for {set i 0} {$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_($i) [new Agent/UDP/tlcdsc]
                set cds_flooding($i) [new Application/CDSFloodingApp]
                $ns_ attach-agent $node_($i) $udp_($i)
                $udp_($i) set packetSize_ 1000
                $udp_($i) set fid_ 1
        }

        for {set i 0} {$i < $val(nn) } {incr i} {
         for {set j 0} {$j < $val(nn) } {incr j} {
                  if {$i != $j} {
                     $ns_ connect $udp_($i) $udp_($j)
                  }
             }
        }
        for {set i 0} {$i < $val(nn) } {incr i} {
         set mmapp_($i) [new Application/TLCDSCApp]
             set cds_flooding($i) [new Application/CDSFloodingApp]
             $mmapp_($i) attach-agent $udp_($i)
             $mmapp_($i) set pktsize_ 1000
        }
```

Figure A.15. An example Scenario File (con.)

```
        for {set i 0} {$i < $val(nn) } {incr i} {
         for {set j 0} {$j < $val(nn) } {incr j} {
               $mmapp_($i) record $node_($j)
             }
         $cds_flooding($i) record-cdsclus $mmapp_($i)
        }
#
# Define node movement model
#
puts "Loading movement pattern..."
source $val(mp)
#
# Tell nodes when the simulation ends
#
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(start_time) "$node_($i) log-movement";
    $ns_ at $val(log_time1) "$node_($i) log-movement";
    $ns_ at $val(log_time2) "$node_($i) log-movement";
    $ns_ at $val(mmapp_end_time) "$node_($i) log-movement";
    $ns_ at $val(end_time) "$node_($i) log-movement";
}
#
# Tell nodes when the simulation ends
#
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(end_time) "$node_($i) reset";
}
$ns_ at $val(end_time) "stop"
$ns_ at $val(halt_time) "$ns_ halt"
proc stop {} {
    global ns_ tracefd
    $ns_ flush-trace
    close $tracefd
}

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 $val(cds_flooding_start_time) "$cds_flooding($i) start"
  $ns_ at $val(cds_flooding_end_time) "$cds_flooding($i) stop"
}
$ns_ run
```

Figure A.16. An example Scenario File (con.)