Modified Merging Clustering and Backbone Formation Algorithms for Mobile Ad hoc Networks

[1]Orhan Dagdeviren, [1]Kayhan Erciyes, [2]Deniz Cokuslu

[1] Izmir University Computer Eng. Dept., Uckuyular, Izmir 35350, Turkey
{orhan.dagdeviren,kayhan.erciyes}@izmir.edu.tr
[2] Izmir Institute of Technology, Computer Eng. Dept., Urla, Izmir 35430, Turkey
denizcokuslu@iyte.edu.tr

Abstract

Clustering and backbone formation are widely used techniques to manage the routing operation in mobile ad hoc networks (MANET)s. In this work, we provide algorithms to form a backbone that is highly resilient to mobility and topology variations in mobile ad hoc networks. The first algorithm forms clusters of nodes in the mobile network each with a leader. The clusters are constructed in a balanced way to distribute the network load evenly. The second algorithm builds a ring network among the leaders of the clusters. The ring backbone is constructed in a fault tolerant and energy efficient way. These two algorithms are integrated in a communication architecture. To the best of our knowledge, our algorithms are the first attempts that constructs balanced clusters with a ring backbone. We show the operation of the algorithms, analyze their proof of correctness, time and message complexities and provide the simulation results in ns2 environment against the density, number of clusters and mobility of the network. We compare our proposed algorithms with the existing algorithms and show that our algorithms create controllable number of balanced clusters and robust ring backbone infrastructures while providing low message count and run-time.

Keywords : clustering, backbone formation, mobile ad hoc networks, layered architecture, spanning tree, message complexity, time complexity.

1 Introduction

Mobile ad hoc networks consist of mobile nodes that communicate over packet radios and do not have a fixed infrastructure unlike cellular networks [23]. Due to the nature of the fundamental applications of MANETs such as the military and rescue operations, efficient and timely cooperation between the nodes is of paramount importance. An efficient method to manage this difficult task would be the provision of some hierarchical mechanism and also another mechanism to provide communication between these hierarchical levels. Clustering the network to construct a robust communication structure is a significant research area in MANETs. Clustering and backbones using clusters are provided in MANETs in order to decrease the number of messages and total time spent for routing. In clustering schemes, each node is classified as either cluster head or cluster member. Cluster members are ordinary nodes whereas cluster heads perform various task on behalf of the members of the clusters.

A MANET can be modeled as a graph $G(V,E)$ where V is the set of vertices (nodes of MANET) and E is the set of edges (communication links between nodes). Graph theoretic algorithms use results from graph theory and develop algorithms to solve various difficult problems. Graph theoretic clustering algorithms, similarly, assume that the underlying network is modeled as a graph and provide clusters of the network using this property.

Various algorithms are proposed in literature for clustering and backbone formation in MANETs. Existing clustering algorithms generally do not focus on balanced clustering. This may result unbalanced load distribution and hot spots may occur in the network area. Besides, existing backbone formation algorithms do not mention about virtual ring formation although ring backbone can be very suitable to give services for upper layers [14, 12]. Moreover, some of the existing algorithms may have $O(N^2)$ message complexity which may lead to high energy consumption for battery constraint mobile nodes.

In this study, we provide algorithms that will first partition the MANET network to balanced clusters and then provide a ring architecture among the leaders of the clusters dynamically in a highly mobile environment. The ring architecture is chosen mainly because it provides better performance under heavy load, circulation of a token eases many distributed control functions that may be required by the upper layers such as mutual exclusion and also it may be used as a framework for full distributed applications in MANETs. Our designed ring architecture is fault tolerant which means that ring is recovered when a node is failed.

Our intended contributions in this work are given below:

- The first algorithm produces balanced clusters and spanning trees within clusters in a MANET. Clusters are enlarged in an asynchronous manner and controlled by upper and lower bound parameters. The algorithm is fully distributed in nature making it suitable for large scale applications.
- Proposed backbone formation as a ring architecture in a MANET is the first algorithm for this purpose in the literature.
- We integrated these algorithms into a communication architecture and show the operation of the algorithms. In this manner, our communication architecture is the first which provides a robust infrastructure which consists of balanced clusters and ring backbone.
- We theoretically analyzed our algorithms where proof of correctness, time complexity and message complexity are studied. We also showed from simulation results that our algorithms outperforms existing studies.

The rest of the paper is organized as follows. Section 2 summarizes related work on spanning tree formation and clustering and backbone formation in MANETs. In Section 3, the proposed architecture is given. The MCA is described, analyzed and simulation results are outlined in Section 4. The BFA is described, analyzed and simulation results are given in Section 5. Finally, the last section provides conclusions and future work anticipations.

2 Related Work

2.1 Spanning Tree Formation and Clustering in MANETs

Clustering is a useful method to manage scarce resources in MANETs and in general computer networks. Various clustering algorithms in literature have been proposed for energy efficiency [2, 29], routing [28] and topology control [5, 21]. The graph theoretic clustering algorithms aim to maintain graph theoretic structures for especially unicast and multicast routing [13]. The two important graph theoretic clustering techniques are spanning tree based clustering and dominating set based clustering.

Spanning tree formation, especially MST, is an efficient and useful method for obtaining a communication infrastructure in networks that can be modeled as graphs. Gallagher et. al. [17] proposed a distributed MST algorithm for undirected graphs with distinct finite weights for every edge. The MST is obtained by merging small fragments to larger fragments on outgoing edges by using some predefined rules. A fragment of an MST is defined as a subtree of the MST. The minimum weighted edge that combines two fragments define the combination strategy. Initially the level of a fragment including a single node is 0. The level of the fragments are updated

while combining. The edge that combines two fragments is called the core of the new fragment. The nodes at the endpoint of the core decides for the new combination by exchanging messages. The total message complexity is $5Vlog_2 V + 2E$ and the time complexity of the algorithm is $O(E+Vlog_2 V)$. Although algorithm provides a spanning tree with fragments, balanced clustering is not mentioned.

Awerbuch [3] proposed a distributed MST algorithm that has two stages: Counting Stage and MST Stage. The Counting Stage provides the knowledge of $|V|$. The MST Stage has two phases. In the first phase, an algorithm identical to Gallagher et. al.'s algorithm is used and terminated when all trees reach the size of $\Omega(V/logV)$. In the second phase, the new algorithmic ideas are introduced for updating the levels of fragments to prevent waiting. The algorithm runs in $O(V)$ time and requires $O(E+Vlog_2 V)$ messages. Same as Gallagher's algorithm, balanced clustering is not mentioned in this study.

Gallagher et. al.'s and Awerbuch's distributed MST algorithms are based on Tree-join-tree approach. The distributed MST algorithm proposed by Lien [24] uses Node-join-tree approach. The algorithm is started at a single node in contrast to Gallagher et. al.'s algorithm. An MST fragment(M) starts from a single node and iteratively grows. In each iteration, each terminal node of M sends a *Follow-me* message to its neighbors. Neighbors decide to join itself to M. The algorithm ends when there is no node that waits to hook to M which indicates MST is formed. The upper bound of total number of messages is $(2E + V(V-1)/4)$ and the time complexity is $O(V^2)$. In the best case, the algorithm needs E^2 messages in $O(Vlog_2V)$ time. Balanced clustering is not focused in this work same as Gallagher's and Awerbuch's algorithms.

Ahuja and Zhu [1] proposed a distributed MST algorithm based on Gallagher et. al.'s algorithm. The improved algorithm needs at most $(2E + 2(V-1)log_2(V/2))$ messages and $2Dlog_2V$ time where D is the diameter of the network. In the best case, it needs $2E$ messages in $2D$ time. On the average case, the algorithm needs $O(E)$ message in $O(D)$ time. Garay et. al. [18] provide a controlled algorithm based on Gallagher et. al.'s algorithm which has the time complexity of $O(D+V^{0,614})$. The deficiency of this algorithm is same with Gallagher's, Awerbuch's and Lien's algorithms.

Maintenance of the spanning tree for clustering in ad hoc networks is challenging and hot research topic. Banerjee and Khuller [4] proposed a spanning tree based protocol for hierarchical routing in wireless networks. They defined the clusters as the subset of vertices whose induced graph is connected. While forming the clusters, the cluster size and the maximum number of clusters that a node can belong is considered. The algorithm finds a rooted spanning tree and a Breadth First Search(BFS) tree. Time complexity of the algorithm is $O(|E|)$. The deficiency of this algorithm is its time complexity when $E{=}N^2$. In this case, the time complexity of the algorithm has an upperbound of $O(N^2)$.

Srivastava and Ghosh [32] proposed a distributed algorithm for forming a rooted spanning tree in dynamic graphs. The root of this tree is aimed to be located near the center of the graph. The algorithm has two phases. In the first phase, a spanning forest is formed and the separate trees of spanning forests are connected to create the rooted spanning tree in the second phase. Authors emphasize the rooted spanning tree formation instead of clustering operation. In this manner, balanced clustering property is not satisfied.

Fernandes and Malkhi [16] proposed a spanning tree based clustering algorithm to limit the amount of routing information needed by hosts. They defined the k-clustering in wireless ad hoc networks is to divide the network into non-overlapping sub networks, also referred as clusters, in which every two hosts in the same cluster are at most k hops apart from each other. They modeled the ad hoc network as an unit disk graph. The algorithm has two phases where a spanning tree is formed in the first phase and the spanning tree is partitioned in the second phase. The algorithm has $O(k)$ time and message complexity. Although unit disk graph resembles wireless transmission pattern, the physical channel model can vary which leads to erroneous algorithm design. This case especially occurs when network area includes obstacles.

Gentile et. al. [19] proposed a routing algorithm by using the clustering approach. Their aim is to reduce the number of overhead messages necessary for maintaining minimum power routing. The degree of clustering and maximum cluster size are controlled by a parameter.

The algorithm minimizes the proposed multi-objective function which controls the degree of clustering and minimizes the power routing. The message complexity of the algorithm is O(N^2) where energy consumption can be high. Wang and Olariu [33] proposed a cluster maintenance algorithm based on the properties of diameter-2 graphs. Their algorithm is tree-based, and they defined their algorithm as cluster-centric. On the other hand, the algorithm does not aim to provide balanced clusters.

Erciyes [15] proposed a distributed spanning tree based clustering algorithm (DSTA) for sensor networks. The depth parameter is provided by the algorithm to adjust the diameter of the clusters. The sink periodically sends *PARENT(nhops)* message to its neighbors to reinitiate the operation. Each node sends the *PARENT((nhops + 1) mod depth)* message to its neighbors upon first reception of the *PARENT(nhops)* message. The recipient of the message with *nhops* = 0 are the *SUBROOTS*, $nhops < depth$ are the *INTERMEDIATE* nodes, $nhops = depth$ are *LEAF* nodes. The algorithm both provides cluster and backbone formation at the same time. Although depth parameter controls cluster size, cluster sizes may vary leading to an unbalanced clustering scheme.

Table 1: Comparison of Clustering Algorithms

	Time Complexity	*Message Complexity*
Srivastava	not given	not given
Gentile	not given	$O(N^2)$
Lien	$O(N^2)$	$O(N\ log_2(N))$
Gallagher	$O(N\ log_2(N))$	$O(N\ log_2(N))$
Awerbuch	$O(E)$	$O(N\ log_2(N)$
Ahuja	$O(D\ log_2(N))$	$O(N\ log_2(N)$
Fernandes	$O(k)$	$O(k)$
Banerjee	$O(E)$	$O(N)$
DSTA	$O(N)$	$O(N)$

A comparison of clustering algorithms are given in Table 1. In this Table, N is the number of nodes, E is the number of edges and D is the network diameter. Algorithms are ordered from worst to best suitable to proposed algorithm. Besides, each algorithm's time and message complexities are summarized in this table.

2.2 Backbone Formation in MANETs

The construction of a path between cluster heads can be defined as backbone formation. It is a useful method for network-wide efficient routing. Rubin et. al [30] classified the nodes as high capacity and low capacity nodes according to their power status. High capacity nodes include Backbone Nodes(*BNs*) and Backbone Capable Nodes(*BCNs*). They present a topological synthesis algorithm that selects a subset of high capacity nodes to form a backbone network. Each backbone node manages the allocation of resources for transport of messages from/to itself and among regular nodes(*RN*) that reside in its managed cluster of nodes. Backbone nodes also interact to coordinate the allocation of MAC layer communications assets such as time slots in their access nets to prevent interferences. They introduced the TBONE protocol which consists of three algorithms: the *Anet* association algorithm, the *BN* election algorithm, and the time slot allocation algorithm. The *Anet* association algorithm provides a mechanism that associates an unassociated low power node with exactly one *BN*. Every unassociated low power node instigates the *Anet* association algorithm by sending *join_request* message to a *BN* or associated *BCN*. The purpose of the *BN* election algorithm is to elect eligible *BCNs* and convert them into *BNs* in order to satisfy the covering requirement. The dynamic weighted labels of *BCNs* determine their eligibility. The unassociated *BCN* that initiates the *BN* election algorithm broadcasts its *ID* and

dynamic weighted label request to all power link associated *BCN* neighbors. All *BCNs* collect data of others. The one with maximum dynamic label will convert itself to a *BN*. The purpose of the *BN-BCN* conversion algorithm is to provide a mechanism to determine redundant *BNs* and convert them into *BCNs* in order to support minimality. The time slot algorithm provides a mechanism for allocation of time slots by *BNs* among their associated low power nodes. The time and message complexity is not given in the article. Although the algorithm is power efficient, it does not provide a ring backbone.

Ya-Feng et. al [36] focused on the formation of the optimal Virtual Multicast Backbone(VMB) with the fewest forwarding nodes to decrease overhead and cost, due to the scarce resource in ad hoc networks. Instead of conventional Steiner tree model, the optimal shared VMB in ad hoc networks is modeled as Minimum Steiner Dominating Set (MSCDS) in Unit-Disk Graphs(UDG), which is NP-hard. One-hop algorithm and *d-hop* algorithm are proposed for approximating MSDCS. One-hop algorithm is divided into steps below:

1. Find a maximal independent set I in *G(V)*
2. In G, apply the Steiner tree algorithm in [31] to find a Steiner tree T for the subset I, with all edges having unit weight. The final solution is the set of the nodes of T.

The *One-hop* Algorithm forms a hierarchical VMB. However, when deployed in sparse UDG, where most multicast nodes are two or more hops apart from each other, it mostly results in trivial single-node multicast clusters and consequently a flat VMB. This implies that *One-hop* Algorithm is not suitable for VMB formation in sparse ad hoc networks. To address this issue, an extended *d-hop* Algorithm is proposed. The *d-hop* Algorithm has a time complexity $O(DV)$ where D is the graph diameter and a message complexity of $O(Vlog(V))$ if d equals to 1, otherwise $O(V^d)$. Same as TBONE protocol, this algorithm does not provide a ring architecture.

Haitao and Gupta [20] proposed the Selective Backbone Formation Algorithm (SBC) for energy efficiency in MANETs. SBC forms backbone in two steps. In the first step, one or more backbone seed nodes are elected. Next, they choose their neighbor nodes into backbone to connect the whole network. When SBC starts, every node computes its priority and broadcasts it in its neighborhood. It also broadcasts the identities of its direct neighbors that it has discovered. Thus, each node gets to know the topology information in its two-hop neighborhood. Backbone seeds are also elected based on two-hop neighborhood information. When electing backbone seeds, they consider two factors. An ideal backbone seed should have high priority. In addition, to speed up the process of backbone formation, it is desirable to have backbone seed nodes chosen from an area of high node density so that more nodes can be covered quickly. They use node degrees as the indicator of node density. Considering these requirements, every node first compares its degree with the degrees of its neighbors based on the two-hop topology information. If its degree is the highest, it picks the neighbor with highest priority as backbone seed. Otherwise, it depends on nodes in other neighborhoods to pick backbone seeds. Time and message complexities are not given in the article. This algorithm does not provide a ring backbone for communication.

In Min et. al's scheme(RVBSM) [25], they assumed that every node records its own location at every second during the period. And whenever a node selects a node from its neighbors, it chooses the one with the highest rank. Every message contains color, rank and locations of both the sender and the 1 hop backbone neighbors of the sender. The algorithm has message complexity of $O(DV)$ and time complexity of $O(V)$, where D is the maximum degree. Same with the previous algorithms, ring formation is not provided.

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 [34]. If the nodes in the dominating set is connected then the dominating set is called connected dominating set (CDS). A two-phased CDS algorithm is proposed in [35], in which initially each vertex marks itself as dominator due to two rules given below (dominator nodes are black):

1. Initially all nodes are white.
2. If the node has two unconnected neighbors it marks itself as black.

3. If the node's neighbors with greater id cover all neighbors of the node, node marks itself as white.

Although CDS backbone has many advantages in network applications such as ease of broadcasting and forming virtual backbones however, when we try to obtain a CDS, we may have undesirable number of cluster heads. Besides, ring formation is not provided by this algorithm.

A comparison of backbone algorithms are given in Table 2 where algorithms are ordered from worst to best suitable to proposed algorithm. In this Table, N is the number of nodes, E is the number of edges, D is the network diameter and Δ is the maximum node degree. Each algorithm's time and message complexities are summarized in this table.

Table 2: Comparison of Backbone Formation Algorithms

	Time Complexity	*Message Complexity*
SBC	not given	not given
TBONE	not given	not given
d-hop algorithm	$O(DN)$	$O(N^d)$
RVBSM	$O(N)$	$O(\Delta N)$
EBS	$O(N)$	$O(N)$
CDS	$O(\Delta^2)$	$O(N)$
DSTA	$O(D)$	$O(N)$

3 The Architecture

Our architecture with three layers to maintain a communication architecture for MANETs is shown at Fig. 1.a. The lowest layer is the routing layer where AODV [27] is used which is a frequently used ad hoc network routing protocol that has a stable ns2 release. Any other routing protocol can be used instead of AODV. The second layer is the clustering layer in which MCA is used which provides balanced non-overlapping clusters. Any other clustering algorithm that provides non-overlapping clusters can also be used in this architecture. The third layer is the backbone formation layer which is responsible for forming a directed ring architecture. BFA is used to maintain the ring architecture as backbone between cluster heads. To our knowledge, BFA is the first attempt for distributed ring formation in MANETs among cluster heads.

In the lowest layer, routing paths are formed for MANETs. Although there are stable routing algorithms like AODV, it is very hard to manage communication links when the mobility is high. The message count for route discovery may be high for geographically distant nodes. With regarding these issues, we designed the second layer where balanced clusters are formed. MCA constructs clusters from nearby nodes in order to decrease route discovery messages and packet routes for especially networks with highly mobile nodes. Although MCA provides an intracluster communication infrastructure, intercluster communication is not handled. In the third layer, BFA connects cluster heads in order to provide intercluster packet transfers. By applying these layers, network wide communication is achieved.

The contributions of this architecture is listed as follows:

- Most routing algorithms may use many messages for transmission between two distant nodes, by designing the second layer, we aim to reduce message transmissions. When our second layer (MCA) is applied on top of the first layer, geographically close nodes are involved in message transmissions.

- Existing clustering and backbone algorithms do not use a routing layer as a sub layer. These algorithms refresh communication paths periodically. When mobility is high and period

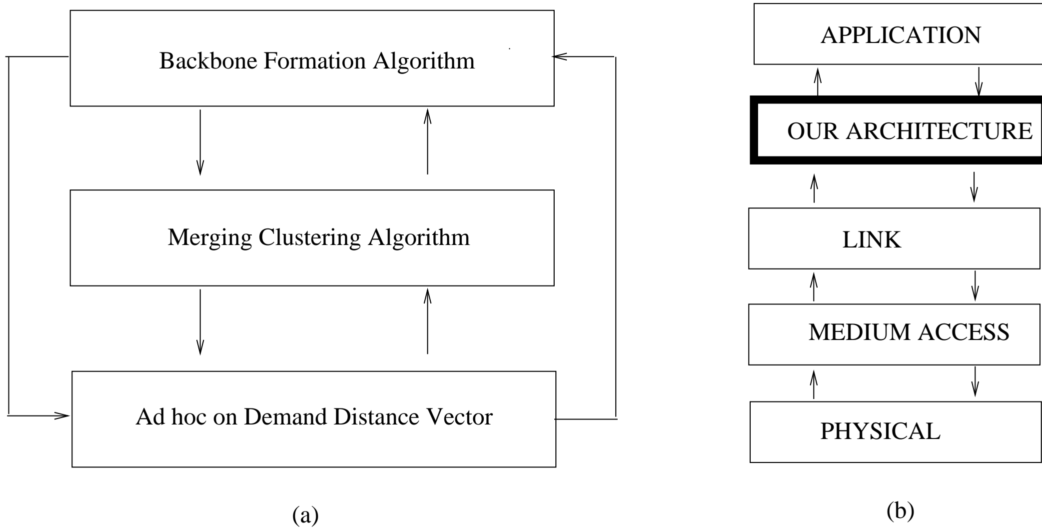

Figure 1: (a)Proposed Architecture (b)MANET Protocol Stack with Our Proposed Architecture

time is low, links can be broken frequently and message transfers may stop. Different than these algorithms, our communication architecture tries to discover new links by applying routing algorithm. By integrating first and second layer, nodes continue intracluster message transfers even under highly mobile conditions. The intercluster message transmission is achieved by integrating third and first layer.

- Existing backbone formation algorithms do not mention about balanced clustering and ring formation. By integrating second and third layer, we provide these issues efficiently.

The general MANET protocol stack with our architecture is shown in Fig. 1.b. Our architecture is located between application and link layers. Various wireless transmission protocols and applications can be used with our architecture. The architecture is worked as follows. When a packet is generated from an application layer of the source node, it first enters to the third layer (backbone layer). If the source node is an ordinary cluster member node, then the packet is passed to the second layer (clustering layer). In the clustering layer, the destination of the message is set to the cluster head and the packet is passed to the routing layer then link layer and so on. When cluster head receives this message from source node, the message enters to physical layer and moves up to clustering layer. In this layer, node first checks whether the destination node of this message resides in the same cluster. If destination resides in the same cluster, the message is sent to the routing layer. Otherwise, the message is sent to the third layer where ring is used to route messages.

4 The Clustering Algorithm

4.1 General Idea of the Algorithm

The Merging Clustering Algorithm(MCA) finds clusters in a MANET by merging the clusters to form higher level clusters as in Gallagher et. al.'s algorithm [17]. However, we emphasize the clustering operation which reduces the message complexity as explained in Section 4.3. Our second contribution is to use upper and lower bound parameters for clustering operation which results in controlled number of nodes in the clusters formed. MCA aims to select the nodes with strong communication links for the same cluster as the third contribution. The fourth contribution is the formation of tree-based routing structure similar to Gallagher et. al.'s algorithm. The

last contribution is the cluster head(leader) selection method as an alternative to the core of the fragment in [17].

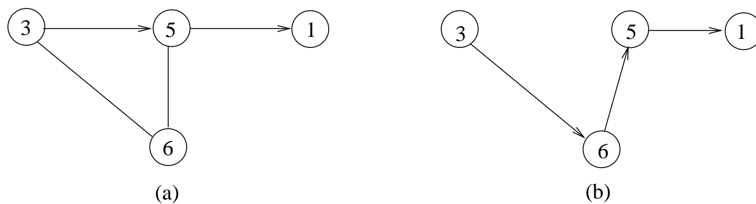

Figure 2: (a) Initial Network (b) Final Network.

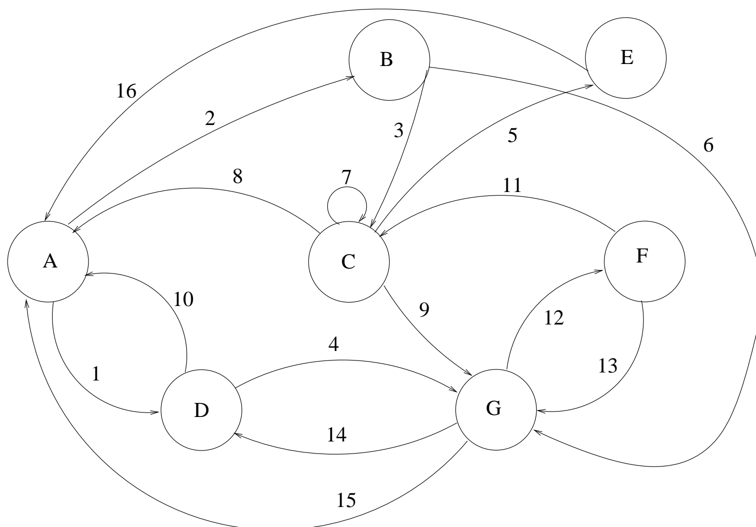

Figure 3: Finite State Machine of the Merging Clustering Algorithm.

Table 3: States of Merging Clustering Algorithm.

State	**Name**	**Description**
A	*IDLE*	Initially all nodes are in *IDLE* state.
B	*WT_TIME*	A node in *WT_TIME* state waits for *Clustering_TOUT*.
C	*WT_CON*	A node in *WT_CON* state waits for *Connect_Mbr* message.
D	*WT_INFO*	A node in *WT_INFO* state waits for *Node_Info* message.
E	*MEMBER*	A node which is a member of a cluster, is in the *MEMBER* state.
F	*LDR_WT_TIME*	A node in *LDR_WT_TIME* state waits for *Clustering_TOUT*.
G	*LEADER*	A node which is a leader of a cluster, is in the *LEADER* state.

Previous MCA is a solely clustering algorithm which aims to construct routing paths. As mentioned in Section 3, links can be broken when mobility is high and period time is low. This may lead to stop message transfers until the next period begins. To solve this problem we proposed modified MCA [10, 9]. Our new algorithm is not a solely routing algorithm, it is used on top of an ad hoc routing algorithm like AODV [27], DSR [22] or TORA [26]. Our aim is to cluster the nearby nodes to decrease the routing overhead and to maintain a robust structure for upper layers. Nodes continue intracluster message transfers even under highly mobile condition by integrating routing layer and clustering layer. To achieve this goal, we redesign the MCA and integrated to our communication architecture. An example operation to show the improvements are given in Fig. 2. Assume that a routing path between node 3 and node 1 is formed by the

Table 4: State Transitions of Merging Clustering Algorithm for $node_j$.

Transition	Name	Description
1	*Wake_TOUT* or my_id = max(neighbor_ids[])	If a node's *node_id* is greater than all of its neighbors or a *WAKE_TOUT* occurs, it sends a *Poll_Node* message to its neighbors and will make a state transition to *WT_INFO* state.
2	*Poll_Node*	A node in *IDLE* state changes its state to *WT_TIME* after receiving *Poll_Node* message.
3	*Clustering_TOUT*	When a *Clustering_TOUT* occurs for a node in *WT_TIME*, it sends a *Node_Info* message and changes its state to *WT_CON*.
4	*Node_Info*	If a node in *WT_INFO* state receives a *Node_Info*, it will send a *Connect_Mbr* and will make a state transition to *LEADER* state.
5	*Connect_Mbr*(destination=j)	A node in *WT_CON* state changes its state to *MEMBER* after receiving *Poll_Node* message.
6	*Connect_Mbr*(destination=my target leader) or *Node_Info*(destination=my target leader)	When a node in *WT_TIME* state, overhears a *Connect_Mbr* or *Node_Info* message which's destination is equal to its target leader, it makes a state transition to *LEADER* state.
7	*State_TOUT* and *node_info_number* $<$ max	When a *State_TOUT* occurs in a node in *WT_CON* state it sends a *Node_Info* message and increments *node_info_number* if *node_info_number* $<$ max.
8	*State_TOUT* and *node_info_number* $>$ max	When a *State_TOUT* occurs in a node in *WT_CON* state it sends a *Node_Info* message and increments *node_info_number* if *node_info_number* $>$ max.
9	*Connect_Mbr*(destination=my target leader) or *Node_Info*(destination=my target leader)	If a node in *WT_CON* state, overhears a *Connect_Mbr* or *Node_Info* message which's destination is equal to its target leader, it makes a state transition to *LEADER* state.
10	*State_TOUT*	When a *State_TOUT* occurs in a node in *WT_INFO* state it makes a transition to *IDLE* state.
11	*Clustering_TOUT*	When a *Clustering_TOUT* occurs for a node in *LDR_WT_TIME*, it sends a *Node_Info* message and changes its state to *WT_CON*.
12	*Poll_Node* and *cluster_level* $<$ *upper_bound*	A node in *LEADER* state changes its state to *WT_TIME* after receiving *Poll_Node* message if its *cluster_level* is smaller than *upper_bound*.
13	*Connect_Mbr*(destination=my target leader) or *Node_Info*(destination=my target leader)	When a node in *LDR_WT_TIME* state, overhears a *Connect_Mbr* or *Node_Info* message which's destination is equal to its target leader, it makes a state transition to *LEADER* state.
14	*State_TOUT* and *cluster_level* $<$ *lower_bound*	When a *State_TOUT* occurs in a node in *WT_INFO* state it makes a transition to *IDLE* state if its *cluster_level* $<$ *lower_bound*.
15	*Period_TOUT*	When a *Period_TOUT* occurs in a node in *LEADER* state it makes a transition to *IDLE* state.
16	*Period_TOUT*	When a *Period_TOUT* occurs in a node in *MEMBER* state it makes a transition to *IDLE* state.

previous MCA which is shown in Fig. 2.a. Also assume that node 5 moves far from node 3 and their communication link is broken as shown in Fig. 2.b. In this case, previous MCA can not recover the broken link until new period begins. On the other hand, our new architecture can handle this link failure as shown in Fig. 2.b.

We assume that each node has distinct *node_id*. Moreover, each node knows its *cluster-*

head_id, *cluster_id* and *cluster_level*. *Cluster_level* is identified by the number of the nodes in a cluster. Clusterhead node is the node with maximum *node_id*. *Clusterhead_id* is equal to the *cluster_id*. The local algorithm consists of sending messages over adjoining links, waiting for incoming messages and processing messages. We also assume each node creates the neighbor list by exchanging *BEACON* messages with its neighbors. The algorithm is initiated by the nodes which have greatest *node_id* from their neighbor's. By using this method we try to eliminate the case in which great number of neighboring nodes try to access the medium at the same time by polling their neighbors. The finite state machine of the algorithm is shown in Fig. 3.

Firstly, the node which has greatest *node_id* among its neighbors sends a *Poll_Node* message to its neighbors. The neighbor nodes that receive *Poll_Node* message sleep for an amount of time. Each node adjusts this time by using a simple function that makes signal strength and *node_id* as parameters. The neighbor node which wakes up earlier than others multicasts a *Node_Info* message. When the neighbor nodes that are sleeping receive the *Node_Info* message they quit from the current clustering session. Originator of the *Poll_Node* message replies the *Node_Info* message with *Connect_Mbr* message. The other nodes quit from current clustering session when they receive *Connect_Mbr* message. After a successful clustering session, the originator of *Poll_Node* message becomes a member of that cluster. The algorithm always tries to cluster the nodes that have strong communication links between each other.

Messages can be transmitted independently in both directions on an edge and arrive after an unpredictable but finite delay, without error and in sequence. Message types are *Poll_Node*, *Node_Info* and *Connect_Mbr*. In addition to message types, there are four types of timeouts: *Period_TOUT*, *State_TOUT*, *Wake_TOUT* and *Clustering_TOUT*. States of the MCA are given in Table 3, transitions of the MCA are given in Table 4. A formal description of the algorithm is shown in Alg. 1 using these states.

4.2 An Example Operation

Assume the snapshot of a mobile network in Fig. 4. Cluster lower bound parameter is given as 4 and upper bound parameter is given as 7. Initially all the nodes are in *IDLE* state. Node 19 which has the greatest *node_id* among its neighbors, multicasts a *Poll_Node* message its neighbors and sets its state to *WT_INFO*. Each neighbor of node 19(node 3, node 7, node 0 and node 17) receives the *Poll_Node* message , sets its state to *WT_TIME* and sets its timer reversely proportional with received signal strength. *Clustering_TOUT* is expired in node 3 earlier than other neighbors of node 19, it multicasts *Node_Info* message to all of its neighbors and changes its state to *WT_CON*. Node 7 receives the *Node_Info* message, sets its state to *IDLE* and quits from the current clustering session. Concurrently, node 19 receives the *Node_Info* message, multicasts *Connect_Mbr* message and sets its state to *LEADER*. Each of node 0, node 17 and node 3 receives the *Connect_Mbr* message destined to node 3 and sets its state to *IDLE*. Node 3 receives the *Connect_Mbr* message, changes its state to *MEMBER*. At the same time, node 11 and node 5, node 16 and node 6, node 18 and node 15 makes clustering operations same as node 3 and node 19.

State_TOUT is expired in node 19 that is in *LEADER* state. Node 19 compares its cluster level that is equal to 2 with the lower bound parameter and decides that it can continue clustering operation until its cluster level reaches to lower bound. Node 19 sends a *Poll_Node* message to all of its neighbors and changes its state to *WT_INFO*. Node 7, node 3, node 0 and node 17 receives the message. Node 3 is in *MEMBER* state, so it does not respond. Other neighbors set their state to *WT_TIME*. Node 7's timer is first expired, it sends *Node_Info* and changes its state to *WT_CON*. Node 19 receives the *Node_Info* message and replies with *Connect_Mbr*. Node 0 and node 17 quits from current clustering session either by receiving *Node_Info* or *Connect_Mbr* message. Concurrently node 11 and node 9, node 16 and node 8, node 18 and node 10 makes clustering operations similar to above mentioned clustering scheme.

Wake_TOUT is expired in node 14 that is in *IDLE* state. Node 14 multicasts a *Poll_Node* message to its neighbors. Node 2, node 4 and node 16 receives the *Poll_Node* message. The *Clustering_TOUT* is first expired in node 2. Node 2 sends a *Node_Info* message to start clustering

Algorithm 1 Merging Clustering Algorithm for node$_j$

```
1: initially current_state_j=IDLE
2:     Legend : □ STATE ∧ input_message ⟶ actions
3:     A message is received by all neighbors of the sender.
4: loop
5:   □ IDLE ∧ Wake_TOUT                ⟶ send Poll_Node(j)
6:                                          current_state_j ← WT_INFO
7:          ∧ Poll_Node(j)             ⟶ current_state_j ← WT_TIME
8:   □ WT_INFO ∧ Node_Info(i,j)        ⟶ send Connect_Mbr(j,i)
9:                                          current_state_j ← LEADER
10:              ∧ State_TOUT          ⟶ current_state_j ← IDLE
11:  □ LEADER ∧ State_TOUT             ⟶ if (cluster_size < lower) then
12:                                            send Poll_Node(j)
13:                                            current_state_j ← WT_TIME
14:             ∧ Period_TOUT          ⟶ current_state_j ← IDLE
15:             ∧ Poll_Node(i)         ⟶ if (cluster_size < upper) then
16:                                            current_state_j ← LDR_WT_TIME
17:  □ LDR_WT_TIME ∧ Clustering_TOUT   ⟶ send Node_info(j,i)
18:                                          current_state_j ← WT_CON
19:             ∧ Connect_Mbr(k,t)     ⟶ if t=my_target_leader then
20:                                            current_state_j ← LEADER
21:  □ WT_CON ∧ State_TOUT             ⟶ if (node_info_number < max) then
22:                                            current_state_j ← IDLE
23:                                          else send Node_Info(j,i)
24:                                          nodeinfo_number++
25:             ∧ Connect_Mbr(i,j)     ⟶ current_state_j ← MEMBER
26:             ∧ Connect_Mbr(k,t)     ⟶ if( t=my_target_leader ∧ my_id=my_leader_id ) then
27:                                            current_state_j ← LEADER
28:  □ WT_TIME ∧ Clustering_TOUT       ⟶ send Node_info(j,i)
29:                                          current_state_j ← WT_CON
30:             ∧ Connect_Mbr(k,t)     ⟶ if( t=my_target_leader )then
31:                                            current_state_j ← LEADER
32:             ∧ State_TOUT           ⟶ if (node_info_number > max) then
33:                                            current_state_j ← IDLE
34:  □ MEMBER ∧ Period_TOUT            ⟶ current_state_j ← IDLE
35: end loop
```

operation with node 14. At the same time, node 11 and node 1, node 19 and node 0, node 18 and node 12 merges and enlarges current clusters.

State_TOUT is expired in node 14 that is in *LEADER* state. Node 14 has 2 elements in its cluster, so it multicasts a *Poll_Node* and changes its state to *LDR_WT_TIME*. Node 16 changes its state to *LDR_WT_TIME*. All of other neighbors do not change their states. After *Clustering_TOUT* is occurred in node 16, it sends a *Node_Info* message to node 14 and changes its state to *WT_CON*. Node 14 replies with *Connect_Mbr* and becomes the cluster head of a cluster with 6 nodes.

4.3 Analysis

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

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

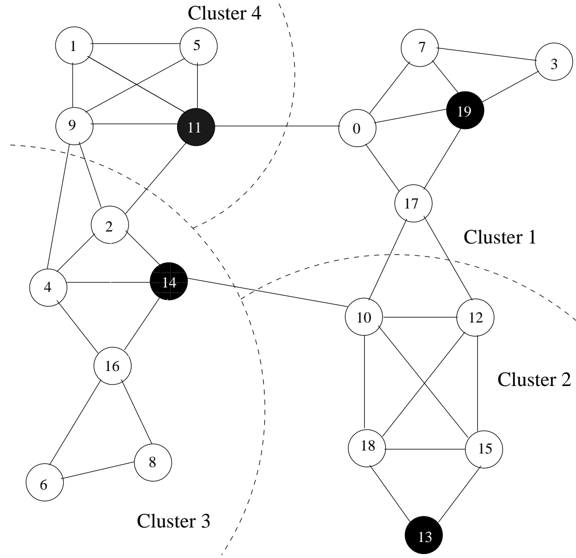

Figure 4: Clusters obtained using Merging Clustering Algorithm.

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

□

Theorem 2. *Message complexity of the clustering algorithm is $O(n)$.*

Proof. Assume that we have n nodes in our network. For every merge operations of two clusters, 3 messages (Poll_Node, Node_Info and Connect_Mbr are required. In the worst case n clustering operation is accomplished. Total number of messages in this case is *3*n* giving the message complexity of $O(n)$.

□

Theorem 3. *MCA is free from deadlock and unbounded waiting.*

Proof. A clustering session between $\text{node}_A$and node_B is the flow of the messages in Fig. 5. node_A initiates the operation by sending a $Poll_Node$ message.

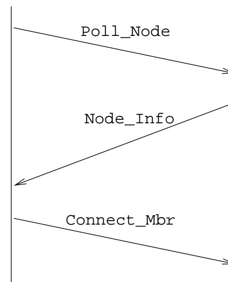

Figure 5: Message Flow Diagram of Merging Clustering Algorithm.

Assume that the algorithm is not free from deadlock or unbounded waiting. This requires at least one of the two cases listed below to be true:

1. node_A is blocked after sending $Poll_Node$ message, waiting for $Node_Info$.

2. node_B is blocked after sending $Node_Info$ message, waiting for $Connect_Mbr$.

Assume that case 1 is true. This requires that node_A must be blocked in WT_INFO state which can be obviously seen in Fig. 3. If node_A receives a $Node_Info$ message from node_B, then node_A makes a state transition to $LEADER$ state. If node_A does not receive a $Node_Info$ message from node_B, then a $STATE_TOUT$ occurs in node_A and node_A makes a state transition to $IDLE$ or $LEADER$ depending on its previous state. Node_A will not block in this case, case 1 contradicts with our assumption.

Assume that case 2 is true. In this case, node_B must be blocked in WT_CON state. If node_B receives a $Connect_Mbr$ from node_A, then node_B makes a state transition to $MEMBER$ state. If node_A does not receive a $Connect_Mbr$ message, then it resends the $Node_Info$ message for $node_info_number$ times and makes a state transition to $IDLE$ or $LEADER$ state as shown in Fig. 3. Both cases contradict with our assumptions. The algorithm is free from deadlock and unbounded waiting. □

4.4 Results

The modified MCA is implemented with the *ns*2 simulator. Total number of nodes are selected from 10 to 50 nodes. Different size of flat surfaces are chosen for each simulation to create very small, small and medium distances between nodes, as well as, high dense, dense and medium connected topologies. Surface areas vary from 120m × 600m to 600m × 600m, 130m × 650m to 650m × 650m, 140m × 700m to 700m × 700m respectively. Average degree of the network is approximately $N/4$ for the medium connected, $N/3.5$ for the dense connected and $N/3$ for the highly dense connected networks where N denotes the total number of nodes in the network. Random movements are generated for each simulation and random waypoint model is chosen as the mobility pattern. Low, medium and high mobility scenarios are generated and respective node speeds are limited from 1.0m/s to 5.0m/s, 5.0m/s to 10.0m/s, 10.0m/s to 20.0m/s. Upper bound and lower parameters is changed to obtain different size of clusters. Beside upper and lower bound parameters, *State_TOUT* parameter is fixed to 250ms and *Wake_TOUT* is defined as 1000ms+(node_id·10). When calculating Clustering_TOUT, we use distance between nodes instead of received signal strength. In this case, all nodes must be equipped with GPS, or use localization techniques. *Clustering_TOUT* parameter is defined as (distance/5)ms. AODV is used as routing layer and other routing protocols can also be used. AODV is chosen since it is a widely used routing protocol which is stable in varying mobility and density conditions [6] and it has a stable *ns*2 release.

Fig. 6 and Fig. 7 display the run-time results of the merging clustering algorithm ranging from 10 to 50 nodes against mobility and density. Run-time values increase linearly and stable results are gained against density and mobility as seen in these results. The recorded run-time values vary between 4s to 7s approximately. Since clusters are enlarged by merging operations, creating clusters with high level can be more time consuming than low level cluster creation. For a MANET with 30 nodes, upper and lower bound parameters are varied to obtain 3 to 7 clusters. Fig. 8 shows the effect of the number of clusters to run-time values. Generally, creating MANETs with more clusters take less time.

We measure the number of messages used for the clustering operation against the density, mobility and number of clusters in Fig. 9 and Fig. 10 where the number of messages increase linearly and change slightly against the mobility and density. Averagely, for a MANET with 10 nodes, 50 messages are exchanged whereas 200 messages are exchanged totaly when the network size is equal to 200 nodes. Fig. 11 shows the number of messages exchanged for a MANET with 30 nodes partitioned into 3 to 7 clusters. It is expected that creating clusters with high levels consume more messages than creating clusters with low levels. However, as seen in Fig. 11 fluctuations occurred due to collisions, and lack of connectivity between nodes ready for clustering.

One of the important parameters of the clustering is the edge-cut. Average edge-cut values are recorded against density, mobility and number of clusters in Fig. 12 and Fig. 13. The

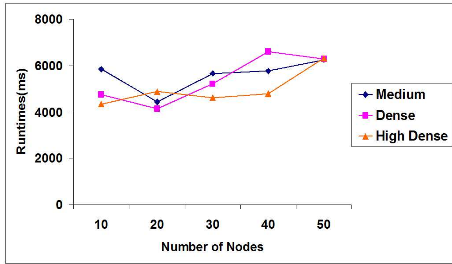

Figure 6: Run-time of MCA against Density.

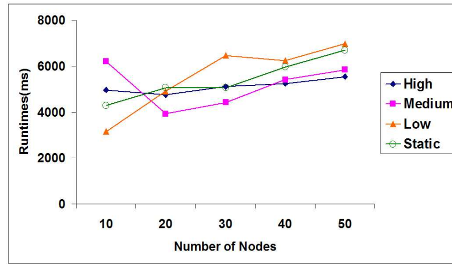

Figure 7: Run-time of MCA against Mobility.

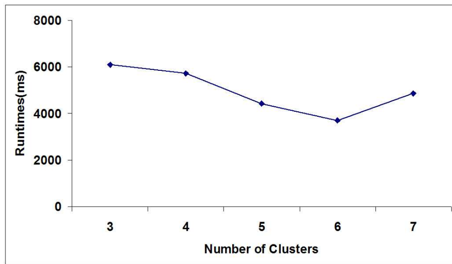

Figure 8: Run-time of MCA against Number of Clusters.

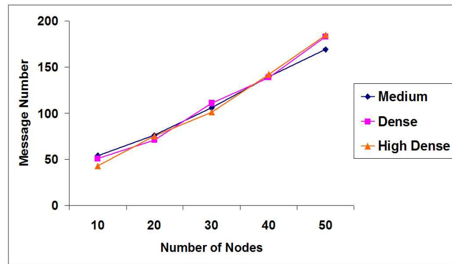

Figure 9: Total Message Number of MCA against Density.

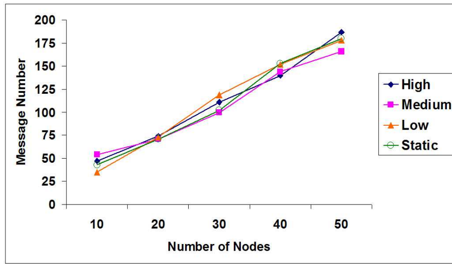

Figure 10: Total Message Number of Merging Clustering Algorithm against Mobility

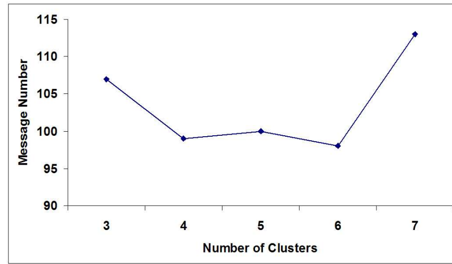

Figure 11: Total Message Number of Merging Clustering Algorithm against Number of Clusters.

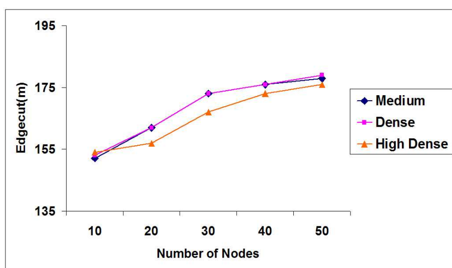

Figure 12: Average Edgecut of Merging Clustering Algorithm against Density

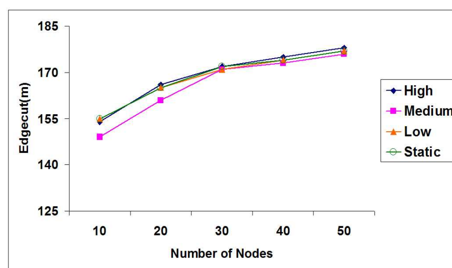

Figure 13: Average Edgecut of Merging Clustering Algorithm against Mobility.

values between 145m to 180m increase linearly and behave stable under different density and mobility conditions as can be seen. Generally, for more partitioned networks, the clusters are more balanced and edge-cut values are smaller as seen in Fig. 14.

The number of nodes in the clusters are controlled by the upper and lower bound parameters. It is strictly defined that cluster size can not exceed the upper bound parameter because cluster heads do not accept this type of clustering requests. So to make comment about clustering quality we need to find the percentage of clusters having node count below lower bound parameter. We measure the percentage of clusters having node count below lower bound parameter or erroneous clusters for MANETs varying from 10 to 50 nodes. The lower and upper bound parameter pairs for 10 to 50 node are (2,4), (3, 6), (4, 9), (5, 12), (5, 15) respectively. Fig. 15 shows the effect of density to clustering. Number of erroneous clusters are fewer in densely connected networks due to fact that clusters are enlarged around cluster heads like a star network. Fig. 16 shows the percentage of erroneous clusters against mobility. As seen in Fig. 16, results are stable. Averagely 20% of the clusters, at worst 38% of the clusters are erroneous approximately as seen in Fig. 15 and Fig. 16.

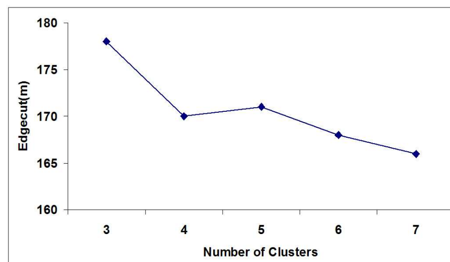

Figure 14: Average Edgecut of Merging Clustering Algorithm against Number of Clusters.

Table 5: Percentage of Clusters Under Lower Bound against Parameters.

Parameters	*Percentage*
2, 6	15
2, 7	9
4, 9	22
6, 11	31

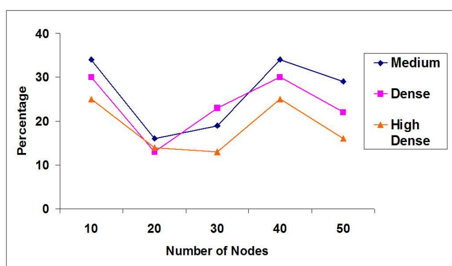

Figure 15: Percentage of Clusters Under Lower Bound against Density.

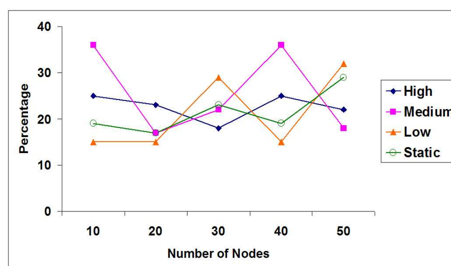

Figure 16: Percentage of Clusters Under Lower Bound against Mobility.

Lastly, we measure the percentage of erroneous clusters against upper and lower bound parameters as seen in Table 5. When parameter pairs are selected as (2, 7), number of erroneous clusters are minimized among others. Number of erroneous clusters are maximized when parameter pairs are selected as (6, 11). Generally for larger lower bounds and smaller upper bounds, percentage of erroneous clusters increase.

Consequently, our results conform with the analysis that run-time values and message counts grow linearly. Also, algorithm is stable under different mobility and density conditions. Upper

and lower bound parameters change the cluster sizes and its selection is very important. Average edge cut values are stable and increases linearly when number of nodes are increased linearly.

4.5 Performance Comparison

In this section, we provide performance comparison of MCA with the existing algorithms. As mentioned in Section 2, two important graph theoretic clustering techniques are spanning tree based and dominating set based clustering. Because of this, we implemented DSTA and Wu's CDS algorithm in ns2 simulator. In Wu's CDS algorithm, we connect each ordinary node to the nearest cluster head and form clusters in this manner. We implemented DSTA algorithm with depth=2 and depth=3 in order to measure the performance of the algorithm.

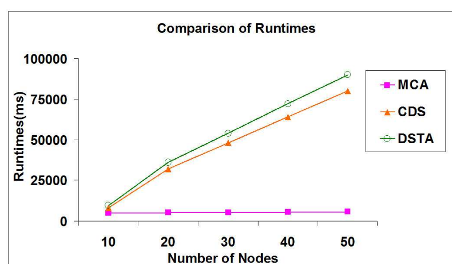

Figure 17: Runtime Performance Comparison.

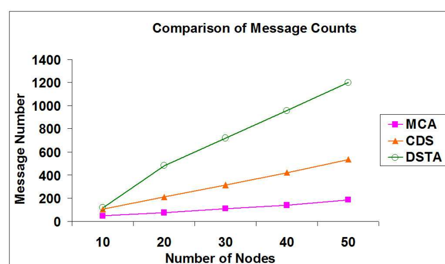

Figure 18: Message Count Performance Comparison.

In order to evaluate and compare the performances of the clustering algorithms, we measure run-time and message counts. In Fig. 17, run-time performances of the algorithms are shown. In CDS, the nodes decide their states by learning the states of their neighbors. In DSTA, each message should be forwarded to all neighbors. In this type of algorithms, collisions may frequently occur since each node may exchange messages with its neighbors in the same time intervals. To prevent these collisions, IEEE 802.11 MAC uses an exponential backoff timer so that the execution time of the algorithms increases. MCA outperforms these algorithms where its run-time performance is approximately 9 times better than CDS, 10 times better than DSTA on the average. Due to same reason, MCA's message count performance is better than its counterparts. On the average, MCA's message count is 3 times smaller than CDS, 6 times smaller than DSTA.

To evaluate quality of the produced clusters, we used two metrics: Number of clusters and the node count in clusters. The number of clusters must be controllable in a preferable clustering algorithm. In MCA, low level clusters merge to form higher level clusters such that the number of clusters decrease as the time passes. A comparison of the number of clusters produced by algorithms is seen in Fig. 19. Since MCA merges lower level clusters to form higher level clusters, its cluster count is more controllable and smaller than its counterparts. DSTA's depth parameter can not adjust the number of cluster in high mobile conditions.

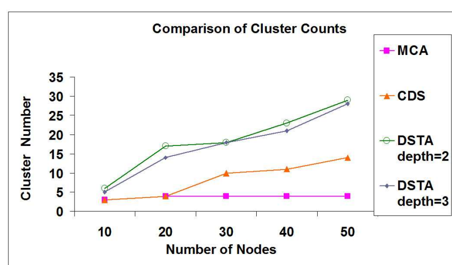

Figure 19: Cluster Count Performance Comparison.

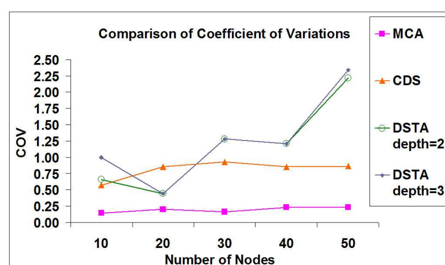

Figure 20: Coefficient of Variation Performance Comparison.

The second criteria of the clustering performance is the balancing of the clusters. Our balance metric is the coefficient of variation (COV) which is computed as standard deviation / mean. If COV < 1 is then the distribution is considered to be of low variance else it is of high variance. We measure the COV values of the algorithms against number of nodes as shown Fig. 20. The average value of COV measurements of MCA is 0.19, CDS is 0.81, DSTA with depth=2 is 1.16 and DSTA with depth=3 is 1.25. In all experiments MCA outperforms other algorithms.

5 Backbone Formation Algorithm

In this section, we describe, analyze and give results for the backbone formation algorithm as the second layer above the clustering algorithm.

5.1 General Idea and Description of the Algorithm

Backbone Formation Algorithm(BFA) [8] forms a directed ring architecture between cluster heads to support an infrastructure for distributed ring algorithms running on upper layers. Distributed mutual exclusion [14, 12], total order multicast [11], leader election [7] are some types of distributed ring algorithms that can benefit from ring backbone. BFA is not responsible for cluster maintenance unlike CDS based algorithms that both creates clusters and backbone at the same time. Creation of clusters and backbone concurrently is an hard job for MANETs under high mobile conditions. In some cases, CDS based algorithms can sacrifice clustering quality parameters like balanced clustering for backbone formation. To prevent this inequality, BFA can be used as upper layer as any clustering algorithm. BFA can be tolerant to some faults since it maintains a global network wide information for each cluster head. We will mention this property in the following paragraphs.

After clustering operation is completed, cluster heads may have no information about each other depending on the characteristics of the clustering algorithm. At this stage, independent from clustering algorithm, BFA supports cluster heads to be aware of each other. The basic idea is each cluster head floods *Clusterhead_Info* message to network. During flooding operations, cluster member nodes act as routers, by just forwarding the messages. BFA is a semi-distributed algorithm, in which each cluster head collects all other's informations, executes same algorithm, and finds its next cluster head on the ring. The first step of the ring formation is MST construction. This operation is achieved by executing a central process using the collected Clusterhead_Info messages.

Algorithm has two modes of operation: Hop-based backbone formation and position-based backbone formation. According to selected mode of the algorithm, *Clusterhead_Info* message can contain two types of information: hop information or position information. In hop-based backbone formation, minimum number of hops counted during flooding operation is used in MST formation. The hop information between two cluster heads must be same since they must form the same MST. In highly mobile scenarios, an agreement between two cluster heads must be made to guarantee the knowledge of same hop count. The other mode of operation is position-based backbone formation. In this scheme, the cluster heads insert their position information in *Clusterhead_Info* message. The position information is used for central MST formation. It is obvious that hop count is a better information for MST formation, assuming the nodes are located uniformly, the position of the nodes can also be accepted as a good measure for MST formation. In position-based backbone formation there is no need for an agreement although nodes are moving. But in this mode, a position tracker like a GPS receiver or a localization technique is needed.

The finite state machine of the algorithm is shown in Fig. 21 and its pseudocodes are given in Alg. 2 with the helper procedures in Alg. 3, Alg. 4, Alg. 5, Alg. 6 and 7. There are four types of states and three types of messages. The states are summarized in Table 6 and transitions are summarized in Table 7.

The first step is the formation of MST using total received cluster head information. After forming MST, the main idea is to divide the ring into two parts and directing these two parts

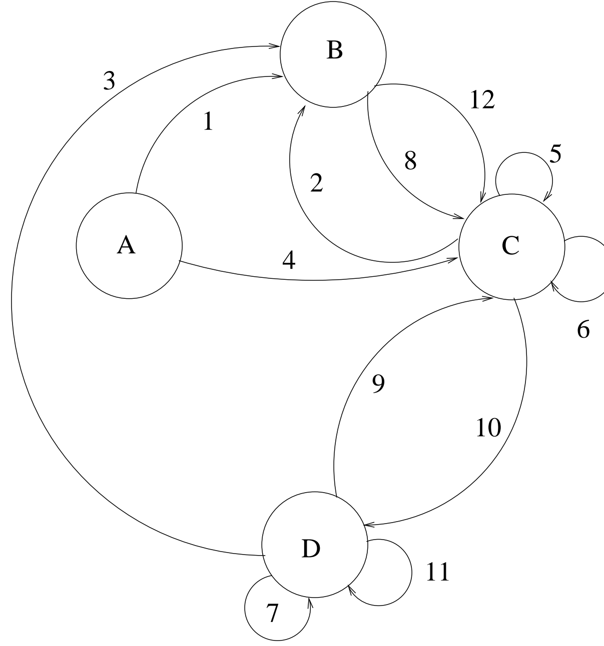

Figure 21: Finite State Machine of the Backbone Formation Algorithm.

Table 6: States of Backbone Formation Algorithm.

State	Name	Description
A	*IDLE*	Initially all cluster head nodes are in *IDLE* state.
B	*WT_INFO*	A node in *WT_INFO* state waits for other *Clusterhead_Info* message.
C	*LEAF*	A cluster head in *LEAF* state has a degree of 1 in its MST.
D	*BACKBONE*	A cluster head which has a degree greater than 1 in its local MST is in *BACKBONE* state.

Algorithm 2 *Backbone Formation Algorithm* for node$_j$

1: **initially** $current_state_j$=IDLE
2: $Legend$: □ STATE ∧ $input_message$ ⟶ actions
3: r : $input_message$ sender node
4: **loop**
5: □ IDLE ∧ $Period_TOUT$ ⟶ **broadcast** $Clusterhead_Info(j)$
6: $current_state_j$ ← 'WT_INFO'
7: □ WT_INFO ∧ $Clusterhead_Info$ ⟶ $current_state_j$ ← LEAF
8: next_clusterhead$_j$=id$_r$
9: □ WT_INFO ∧ $TOUT$ ⟶ $current_state_j$ ← LEAF
10: □ !WT_INFO ∧ $Clusterhead_Info$ ⟶ **call** Ring_Construct($Clusterhead_Info(r)$)
11: □ (LEAF ∨ BACKBONE) ∧ state$_{next_clusterhead_j}$=CRASHED ⟶ **call** Recovery()
12: □ (LEAF ∨ BACKBONE) ∧ $Period_TOUT$ ⟶ **broadcast** $Clusterhead_Info(j)$
13: $current_state_j$ ← 'WT_INFO'
14: **end loop**

to each other to form ring. The first part is directed path of $BACKBONE$ cluster heads, the second part is the directed path of $LEAF$ cluster heads.

The vital part of our backbone formation is the first part, the path connecting $BACKBONE$ cluster heads. Firstly we choose the starting $BACKBONE$ cluster head. Starting backbone cluster head has minimum connected $BACKBONE$ cluster heads in its MST. After choosing starting $BACKBONE$ cluster head, we choose the nearest $BACKBONE$ cluster head as its

Table 7: State Transitions of Backbone Formation Algorithm for node$_j$.

Transition	Name	Description
1, 2, 3	$Period_TOUT$	When a $Period_TOUT$ occurs in a node in $IDLE$, $LEAF$ or $BACKBONE$ states, it broadcasts a $Clusterhead_Info$ message and makes a state transition to WT_INFO state.
4	$Clusterhead_Info$	If a node in $IDLE$ state receives a $Clusterhead_Info$, it will make a state transition to $LEAF$ state.
5	$Clusterhead_Info$	When a node in $LEAF$ state receives a $Clusterhead_Info$, it will first reconstruct the ring. Then, it will make a state transition to $LEAF$ state if it is a $LEAF$ node in the constructed ring.
6, 7	Detect Next Clusterhead Crash	When a node which is in $LEAF$ or $BACKBONEstate$,detects its crash of next cluster head, then the node starts executing recovery procedure.
8	$TOUT$	When a $TOUT$ occurs in a node in WT_INFO state, it makes a transition to $LEAF$ state.
9	$Clusterhead_Info$	When a node in $BACBONE$ state receives a $Clusterhead_Info$, it will first reconstruct the ring. Then, it will make a state transition to $LEAF$ state if it is a $LEAF$ node in the constructed ring.
10	$Clusterhead_Info$	If a node in $LEAF$ state receives a $Clusterhead_Info$, it will first reconstruct the ring. Then, it will make a state transition to $BACKBONE$ state if it is a $BACKBONE$ node in the constructed ring.
11	$Clusterhead_Info$	When a node in $BACKBONE$ state receives a $Clusterhead_Info$, it will first reconstruct the ring. Then, it will make a state transition to $BACKBONE$ state if it is a $BACKBONE$ node in the constructed ring.
12	$Clusterhead_Info$	If a node in WT_INFO state receives a $Clusterhead_Info$, it will make a transition to $LEAF$ state.

Algorithm 3 *Ring_Construct* procedure for node$_j$

1: construct minimum spanning tree by total received cluster head information.
2: **if** degree$_j$=1 **then**
3: call *ordinary_leaf_proc*.
4: **else**
5: $current_state_j \leftarrow$ BACKBONE.
6: **end if**
7: **if** $current_state_j$=BACKBONE or ($state_j$=LEAF $\wedge$ $next_clusterhead$=$\emptyset$) **then**
8: call *backbone_proc*.
9: **end if**

Algorithm 4 *Recovery* procedure for node$_j$

1: **multicast** $Clusterhead_Crashed(next_clusterhead_j)$ to all nodes with cluster head = $next_clusterhead_j$.
2: $next_clusterhead_j = next_clusterhead_{next_clusterhead_j}$.

next cluster head on ring. This operation iteratively continues in this way. Intuitively, this connection policy of $BACKBONE$ cluster heads results in smaller hops and reduces routing delay. Ending $BACKBONE$ cluster head chooses its $LEAF$ with the smallest id. Alg. 5 shows

Algorithm 5 *Backbone* procedure for node$_j$

1: *starting_backbone* ← The unmarked BACKBONE cluster head such that its connectivity to other BACKBONE nodes is smallest between all other BACKBONE cluster heads.
2: $next_clusterhead_{starting_backbone}$ ← The next unmarked BACKBONE cluster head in the MST.
3: **if** $next_clusterhead_{starting_backbone} \neq \emptyset$ **then**
4:     $temporary_backbone \leftarrow next_clusterhead_{starting_backbone}$.
5: **else**
6:     $next_clusterhead_{starting_backbone}$ ← The smallest $LEAF$ node of *starting_backbone*.
7:     mark the *starting_backbone*.
8: **end if**
9: **if** $node_j = starting_backbone$ **then**
10:     $next_clusterhead_j \leftarrow next_clusterhead_{starting_backbone}$
11: **else**
12:     call *backbone_connect* procedure.
13: **end if**

Algorithm 6 *Backbone_Connect* procedure for node$_j$

1: **while** all BACKBONE nodes are not marked **do**
2:     $next_clusterhead_{temporary_backbone}$ ← The unmarked BACKBONE node with the smallest distance to the *temporary_backbone*.
3:     **if** $next_clusterhead_{temporary_backbone} \neq \emptyset$ **then**
4:         $next_clusterhead \leftarrow temporary_backbone$.
5:     **else**
6:         $next_clusterhead_{temporary_backbone}$ ← The smallest $LEAF$ node of *temporary_backbone*.
7:     **end if**
8:     mark the $next_clusterhead_{temporary_backbone}$.
9: **end while**
10: **if** $current_state_j = LEAF \wedge next_clusterhead_j = \emptyset$ **then**
11:     $next_clusterhead_j$ ← The $LEAF$ node with smallest *node_id* from one of the previous BACKBONE cluster heads (nearest in MST) of parent BACKBONE cluster head.
12:     **if** $next_clusterhead_j = \emptyset$ **then**
13:         $next_clusterhead_j \leftarrow starting_backbone$.
14:     **end if**
15: **end if**

Algorithm 7 *Ordinary_Leaf* procedure for node$_j$

1: $current_state_j \leftarrow LEAF$.
2: $leaf_clusterhead$ ← The $LEAF$ cluster head with same parent and nearest greater *node_id*.
3: **if** $leaf_clusterhead \neq \emptyset$ **then**
4:     $next_clusterhead_j \leftarrow leaf_clusterhead$.
5:     mark $leaf_clusterhead$.
6: **end if**

the pseudocode for forming $BACKBONE$ path.

Secondly, $LEAF$ cluster heads are connected to each other. $LEAF$ cluster heads of same $BACKBONE$ cluster head are connected to each other as defined in $ordinary_leaf_proc$ in Alg. 7. The aim of connecting $LEAF$ cluster heads with the same $BACKBONE$ cluster heads to each other is to make the routing process over the same $BACKBONE$ cluster heads to reduce delay. $LEAF$ cluster heads which can't find its next cluster head search a $LEAF$ cluster head from the previous $BACKBONE$ cluster heads of their parent to find a $LEAF$ cluster head seen in Alg. 6. Aim of this connection is the maintenance of routing operation by using $BACKBONE$ cluster heads.

Each cluster head can insert its cluster member's *node_id*s in $Clusterhead_Info$ message. After the ring formation, if a cluster head detects the crash of its next cluster head, it can multicast a *Clusterhead_Dead* message to crashed cluster head's members to restart clustering

operation. This is the fault tolerant aspect of our algorithm. The procedure for recovery is given in Alg. 4. This can also be seen in finite state machine in Fig. 21. To support this functionality, clustering algorithm running below must be updated. If this crash occurs during a real time operation, cluster head updates its next cluster head to next-next cluster head and continues its operation.

5.2 An Example Operation

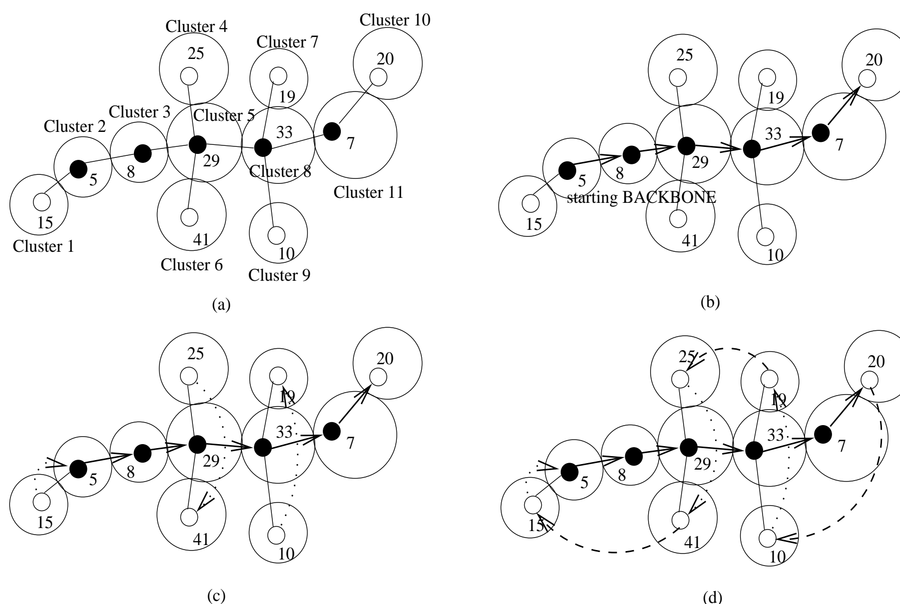

Figure 22: An Example Operation for Backbone Formation Algorithm.

A MANET with cluster heads are obtained using MCA in Fig. 22.a. Nodes 15, 5, 8, 41, 29, 25, 33, 19, 10, 7 and 20 are cluster heads of clusters 1 to 11, respectively. Each cluster head floods the *Clusterhead_Info* message to the network. After each cluster head receives the *Clusterhead_Info* message of the others, minimum spanning tree in Fig. 22.a is formed by all cluster heads. Nodes 15, 41, 25, 10, 19 and 20 identify themselves as *LEAF* cluster heads since their degrees are all 1. Node 5, 8, 29, 33 and 7 identify themselves as *BACKBONE* cluster heads since their degrees are greater than 1. *BACKBONE* cluster heads are filled with black and *LEAF* cluster heads are left unfilled as shown in Fig. 22.a. To connect *BACKBONE* nodes, a starting *BACKBONE* cluster head must be chosen. The criteria is to select the *BACKBONE* node which has the smallest connection to other *BACKBONE* cluster heads. Node 5 is connected to node 15, node 7 is connected to node 20. Either of node 5 and node 7 can be the choice for the starting *BACKBONE* cluster head. Node 5 is selected because its *node_id* is smaller. Node 5 selects the next cluster head as node 8, node 8 selects the next cluster head as node 29 and operation continues in this way. Ending *BACKBONE* cluster head points to its *LEAF* with the smallest *node_id*. These directions can be seen in Fig. 22.b with bold directed lines. *LEAF* cluster heads of a *BACKBONE* cluster head are directed to each other from smallest to greatest. Node 10 is directed to node 19, node 25 is directed to node 41 as seen in Fig. 22.c with dotted directed arcs. Lastly, *LEAF* cluster heads of different *BACKBONE* cluster heads are connected as in Fig. 22.d. Each *LEAF* cluster head which can not find next cluster head, searches a *LEAF* cluster head from children of the previous *BACKBONE* cluster head of its parent *BACKBONE* cluster head. Node 20 is connected to node 10, node 19 is connected to node 25, node 41 is connected to node 15 as shown with dashed arcs in Fig. 22.d.

5.3 Analysis

Theorem 4. *Message complexity of the backbone formation algorithm is $O(Kn)$.*

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

Theorem 5. *Time complexity of the backbone formation algorithm is $O(Kn)$.*

Proof. Assume that we have n nodes in our network. Flooding of K messages to the network takes Kn time. □

Proposition 1. *Two $LEAF$ cluster heads without any parent $BACKBONE$ cluster head, directs to each other.*

Proof. There are 2 cases for a $node_j$:

1. The id_j is not maximum among other $LEAF$ nodes belonging to the same cluster and $next_clusterhead_j$ is not defined.

2. The id_j is maximum among other $LEAF$ nodes belonging to the same cluster.

Both of the cases are covered in Leaf procedure shown in Alg. 7, thus makes the proposition true. □

Proposition 2. *The ending $BACKBONE$ cluster head directs to its $LEAF$ with smallest id.*

Proof. The case is handled in line 6 of Backbone procedure in Alg. 5. □

Proposition 3. *The $LEAF$ cluster head with id_j connects to the $LEAF$ cluster head with id_i if these conditions are satisfied:*

1. *Their parents are same.*

2. *$id_i > id_j$ and no other node with id_k satisfies $id_i > id_k > id_j$ exists.*

Proof. Execution of line 2 in Leaf procedure in Alg. 7 satisfies the proposition. □

Proposition 4. *The $LEAF$ cluster head $node_j$ with maximum id of starting $BACKBONE$ cluster head connects to starting $BACKBONE$ cluster head.*

Proof. Execution of line 11 in Backbone_Connect procedure in Alg. 6 satisfies the proposition. □

Proposition 5. *$BACKBONE$ cluster heads are connected from starting $BACKBONE$ cluster head to ending.*

Proof. Starting cluster head is selected in Backbone Procedure in line 2 given in Alg. 5 and iterative construction of path of $BACKBONE$ cluster heads is maintained in Backbone_Connect procedure from line 1 to line 9. □

Proposition 6. *The $LEAF$ cluster head with maximum id of a $BACKBONE$ cluster head connects to the $LEAF$ cluster head with minimum id of a previous $BACKBONE$ cluster head, if previous $BACKBONE$ cluster head has a $LEAF$ cluster head. Previous $BACKBONE$ cluster heads are searched iteratively.*

Proof. The case is handled in line 11 of Backbone_Connect procedure given in Alg. 6. □

Theorem 6. *BFA ensures ring formation.*

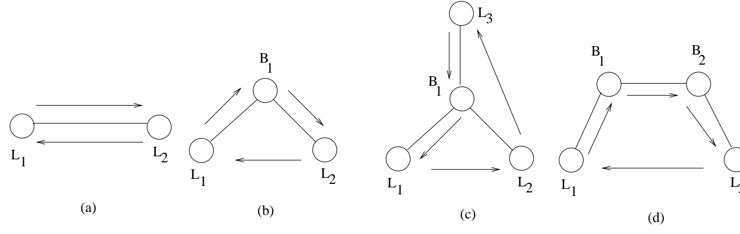

Figure 23: (a) MST with 2 nodes (b) MST with 3 nodes (c) MST with 4 nodes (first case) (d) MST with 4 nodes (second case)

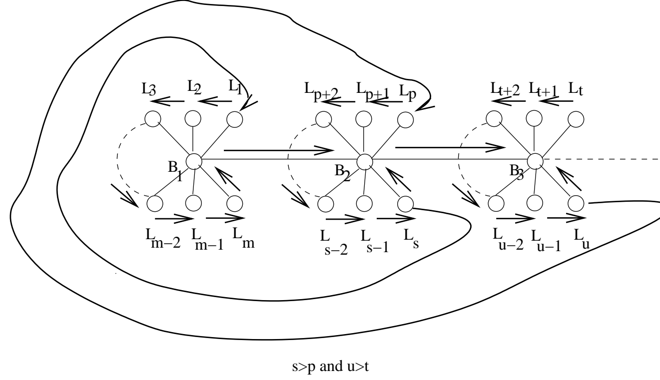

Figure 24: MST with n nodes.

Proof. We will proove the ring formation by induction. Assume that we have k cluster heads in our network and the indices of cluster heads are their corresponding ids. 'B' is the shortening of the $BACKBONE$ and 'L' is the shortening of the $LEAF$ in Fig. 23 and Fig. 24.

- Base case(k=2):

  When there are 2 cluster heads, the MST in Fig. 23.(a) is obtained. Since there is no $BACKBONE$ cluster head, two $LEAF$ cluster heads form ring from Proposition 1.

- Case(k=3):

  There is one $BACKBONE$ cluster head and two $LEAF$ cluster heads when there are 3 cluster heads in MST as shown in Fig. 23.(b). $BACKBONE_1$ is directed to $LEAF_2$ from Proposition 2. $LEAF_2$ is directed to $LEAF_1$ from Proposition 3. $LEAF_1$ is directed to $BACKBONE_1$ from Proposition 4.

- Case(k=4):

  There are two possible cases for MST with 4 nodes as shown in Fig. 23.(c) and Fig. 23.(d). The MST in Fig. 23.(c) is the star graph and is the extended version of Fig. 23.(c) with the same topology. So Proposition 2, Proposition 3 and Proposition 4 are applied to form ring. The MST in Fig. 23.(d) is the second possible case. $BACKBONE_1$ and $BACKBONE_2$ is connected by applying Proposition 5, Proposition 6.

- Induction step:

  It can be obviously seen that the MST with k=n cluster heads in Fig. 24 can be obtained by extending the trees in Fig. 23. Assume that BFA finds the ring for MST with n nodes. Now we will add a new node(node$_j$) to this MST and show that ring is preserved. To obtain a MST with n+1 nodes, we have 2 possibilities:

1. Adding as a *BACKBONE* cluster head as and $B_1 < B_2 ... < B_j < ... < B_m$: Proposition 5 preserves the ring formation.
2. Adding as a *LEAF* cluster head: Proposition 1, Proposition 2, Proposition 3, Proposition 4, Proposition 6 preserves the ring formation.

□

5.4 Results

The position-based backbone formation algorithm is implemented with the *ns*2 simulator and the clusters are obtained using the modified MCA algorithm. Number of nodes in the clusters can be adjusted by the parameters of MCA and density and mobility is varied as in the MCA implementation.

Run-time performance of the algorithm is measured against mobility, density and number of cluster heads. As seen in Fig.25 and Fig.26 run-time values increase linearly and are stable. The algorithm runs for 3.5s on a MANET with 10 nodes and 7.5s on a MANET with 50 nodes approximately. We use data aggregation technique instead of blind flooding. By using this technique, a message can include information of more than one cluster head. Fig. 27 shows the run-time values against number of cluster heads for a MANET with 30 nodes. By using data aggregation technique, close run-time values are gained.

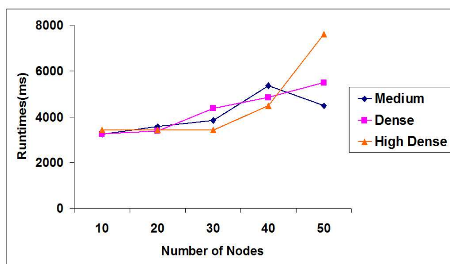

Figure 25: Run-time Performance for Backbone Formation Algorithm against Density.

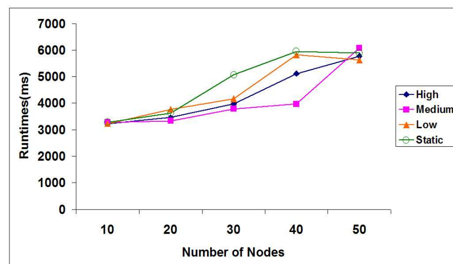

Figure 26: Run-time Performance for Backbone Formation Algorithm against Mobility.

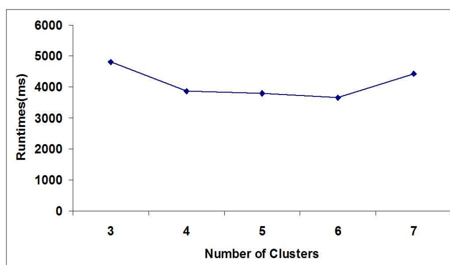

Figure 27: Run-time Performance for Backbone Formation Algorithm against number of Clusters.

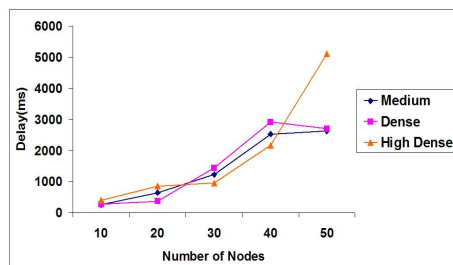

Figure 28: Roundtrip Delay for Backbone Formation Algorithm against Density.

Round-trip delay as measured against number of cluster heads, total number of nodes, mobility and density area are recorded. As shown in Fig. 28 and 29, the round trip values increases linearly and at worst, backbone formation scheme is completed in 5.2s for a MANET with 50 nodes. Our algorithm results in approximate round-trip delay values for high mobile and high dense scenarios as shown in Fig. 28 and Fig. 29. The cluster heads and ordinary routing nodes form the ring. The round-trip delay against number of cluster heads values are measured for the

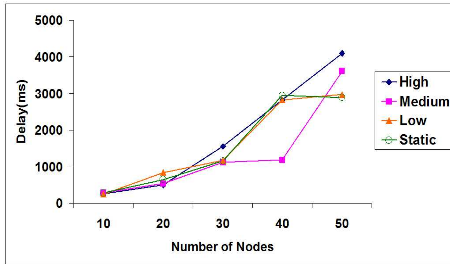

Figure 29: Roundtrip Delay for Backbone Formation Algorithm against Mobility.

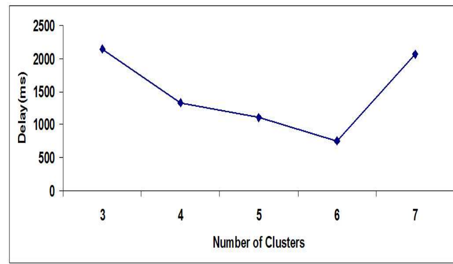

Figure 30: Roundtrip Delay for Backbone Formation Algorithm against Clusterhead Number.

MANET with 30 nodes. Fig. 30 shows that not only number of cluster heads are important for round-trip delay, but also number of ordinary routing nodes. Consequently, our results conform with the analysis that run-time values increase linearly with the increased number of nodes. Round-trip delays also increase linearly against the total number of nodes and cluster number in MANET. Round-trip delay values are stable under different mobility conditions and different densities.

5.5 Performance Comparison

In this section, we give the performance comparisons of the backbone formation algorithms. We implemented MCA and BFA together, to form clusters and backbone at the same. To compare the performance of our algorithms, we choose DSTA and Wu's CDS algorithm, since they achieve clustering and backbone formation operations at the same time. DSTA algorithm forms spanning tree backbone, Wu's CDS algorithm provides a CDS backbone. We measure the run-times of the algorithms and their roundtrip delays. Runtime performances of the algorithms are given in Fig. 31. Although MCA and BFA is implemented together, the run-times of these algorithms are much more smaller than those of CDS and DSTA. The main reason of this situation is collisions as mentioned in Section 4.5. MCA and BFA's total run-time is 9532 ms, CDS's run-time is 46400 ms, DSTA's run-time 52297 ms on the average. Roundtrip delay performances of the algorithms are given in Fig. 32. Although node to sink communication performance of DSTA algorithm is good, its roundtrip delay is high since a cluster head should flood its message to the network in order to reach all cluster heads. In CDS algorithm, each cluster head should be acknowledged by its neighbor cluster heads, this operation increases roundtrip delay. On the other hand, each node should only be acknowledged by its next node on the ring during message traversal, thus BFA has lowest roundtrip delay. BFA's roundtrip delay is 1847 ms, CDS's roundtrip delay is 3600 ms and DSTA's roundtrip delay is 4500ms on the average. In all measurements, our proposed backbone formation scheme outperforms existing algorithms.

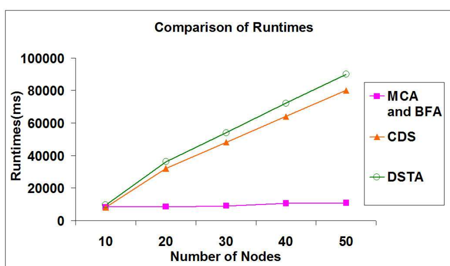

Figure 31: Runtime Performance Comparison.

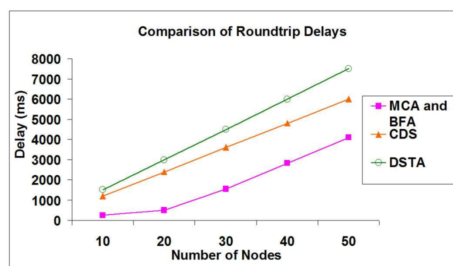

Figure 32: Roundtrip Delay Performance Comparison.

6 Conclusions

We provided two algorithms that can be used to effectively form a communication architecture in a highly mobile network. Our first algorithm, MCA, produces clusters asynchronously. We used upper and lower bound parameters to balance the clusters. MCA is not a solely routing algorithm, it is located on top of a routing algorithm. We gave the finite state machines and pseudocodes of the algorithm. We exemplified the algorithm on a sample instance. We analyzed the proof of correctness, time and message complexities. We implemented the algorithm in simulation environments. We measured the run-time, total message number and cluster quality of MCA and show that algorithm performs well. We compared MCA's performance with those of CDS and DSTA algorithms. From our measurements, we found that on the average MCA is 9 times faster than CDS, 10 times faster DSTA. Besides, MCA uses 3 times less than CDS and 6 times less message than DSTA. Moreover, from our measurements we observed that, MCA may merge lower level clusters to form higher level clusters, thus cluster count is controllable. This also results to balanced clustering where MCA's COV values are 4 times less than CDS, 6 times less than DSTA. From all experiments, we found that MCA outperforms other algorithms.

Our second algorithm, BFA forms a directed ring architecture between cluster heads. BFA is not designed to construct clusters, instead BFA constructs a backbone on top of a clustered network. We gave the description of the algorithm with its finite state machine and algorithmic representation. We exemplified the algorithm on a sample instance. We analyzed the proof of correctness, time and message complexities of BFA algorithm. We implemented the algorithm in ns2 simulator. We measured run-times and roundtrip delays of BFA algorithm and observed that algorithm performs well. We compared BFA algorithm with CDS and DSTA algorithms. From our measurements, we observed that backbone formation with BFA is approximately 4.8 times faster than CDS, 5.4 times faster than DSTA. Moreover, roundtrip delay of BFA is approximately 2 times less than CDS, 2.5 times less than DSTA. We observed that BFA outperforms existing algorithms.

Provision of a ring is a step towards the realization of many other middleware system related tasks such as distributed mutual exclusion, time synchronization, message ordering and consensus. Out of these, distributed consensus is a very critical task in a distributed mobile application such as military or rescue operations where nodes should agree on what to perform next in a real-time. We think the proposed architecture can be favorably used as the framework for distributed consensus. Various non-mobile applications such as the facility location problem using vertex coloring could also benefit from this framework. A simple approximation algorithm for facility location provides a vertex set V_c to cover the graph. The vertices are added to V in the order of their degrees and at each iteration, the added vertex and its incident edges are removed from the graph. This algorithm can suitably be implemented using this framework by providing a simple consensus algorithm over the ring to choose which vertex to be deleted by the cluster heads. Our work is ongoing and we are planning to implement various distributed algorithms at systems or application level such as above on top of this architecture.

References

[1] Ahuja, M., and Zhu, Y. A Distributed Algorithm for Minimum Weight Spanning Trees Based on Echo Algorithms, in Proc. of the 9th International Conference on Distributed Computing Systems, 1989.

[2] Amis, A. D., and Prakash, R. Load-Balancing Clusters in Wireless Ad Hoc Networks, in Proc. of the 3rd IEEE ASSET00, 2000, pp.25-32.

[3] Awerbuch, B. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and related problems, in Proc. of the 9th Annual ACM Symposium on Theory of Computing, 1987, pp.230-240.

[4] Banerjee, S., and Khuller, S. A Clustering Scheme for Hierarchical Routing in Wireless Networks, Tech. Report CS-TR-4103, University of Maryland, College Park, 2000.

[5] Belding-Royer, E. M. Hierarchical Routing in Ad Hoc Mobile Networks, Wireless Commun. and Mobile Comp., V.2, N.5, 2002, pp.515-32.

[6] Broch, J., Maltz, D. A., Johnson, D. B., Hu, Y. C., and Jetcheva, J. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols, in Proc. of the Fourth Annual ACM/IEEE International Conference on Mobile and Networking (MobiCom 98), 1998, pp.85-97.

[7] Chang, E. J., and Roberts, R. An Improved Algorithm for Decentralized Extrema Finding in Circular Arrangements of Processes. ACM Com., 1979, pp.281-283.

[8] Dagdeviren, O., and Erciyes, K. A Distributed Backbone Formation Algorithm for Mobile Ad hoc Networks. ISPA 06, Springer Verlag LNCS, V.4330, 2006, pp.219-230.

[9] Dagdeviren, O., Erciyes, K., and Cokuslu, D. A Merging Clustering Algorithm for Mobile Ad hoc Networks, ICCSA 06, Springer Verlag LNCS, V.3981, 2006, pp.681-690.

[10] Dagdeviren, O., Erciyes, K., and Cokuslu, D. Merging Clustering Algorithms in Mobile Ad hoc Networks, ICDCIT 05, Springer Verlag LNCS, V.3816, 2005, pp.56-61.

[11] Erciyes, K., and Sahan, A. A Real-time Total Order Multicast Protocol, ICCS 04, Springer-Verlag LNCS, V.3036, 2004, pp.357-364.

[12] Erciyes K. Cluster-based Distributed Mutual Exclusion Algorithms for Mobile Networks, in Proc. of the International Conference on Computational Science, Springer-Verlag, V.3149, 2005, pp.933-940.

[13] Erciyes, K., Dagdeviren, O., Cokuslu, D., and Ozsoyeller, D. A Survey of Graph Theoretic Clustering Algorithms in Mobile Ad hoc Networks and Wireless Sensor Networks, Journal of Applied and Computational Mathematics, 2007, pp.162-180.

[14] Erciyes K. Distributed Mutual Exclusion Algorithms on a Ring of Clusters, in Proc. of the International Conference on Computational Science and Its Applications, V.3045, 2004, pp.518-527.

[15] Erciyes, K., Ozsoyeller, D. and Dagdeviren, O. Distributed Algorithms to Form Cluster based Spanning Trees in Wireless Sensor Networks, in Proc. of ICCS 08, Springer Verlag LNCS, V.5101, 2008, pp.519-528.

[16] Fernandess, Y., and Malkhi, D. K-Clustering in Wireless Ad Hoc Networks, in Proc. of the 2nd ACM Workshop on Principles of Mobile Computing, Toulouse, France, 2002, pp.31-37.

[17] Gallagher, R. G., Humblet, P. A., and Spira, P. M. A Distributed Algorithm for Minimum-Weight Spanning Trees, ACM Transactions on Programming Languages and Systems, V.5, 1983, pp.66-77.

[18] Garay, J.A., Kutten, S., and Peleg, D. A sub-linear time distributed algorithm for minimum-weight spanning trees, in Proc. of the 34th Annual Symposium on Foundations of Computer Science, 1993, pp.659-668.

[19] Gentile, C. Haerri, J., and Van Dyck, R. E. Kinetic Minimum-Power Routing and Clustering in Mobile Ad-hoc Networks, in Proc. of the IEEE Vehicular Technology Conf., 2002, pp.1328-1332.

[20] Haitao, L. and Gupta, R. Selective Backbone Construction for Topology Control in Ad hoc Networks, in Proc. of the International Conference on Mobile Ad-hoc and Sensor Systems, 2004, pp.41-50.

[21] Hong, X. Y., Xu, K. X., and Gerla, M. Scalable Routing Protocols for Mobile Ad Hoc Networks, IEEE Network, 2002, pp.11-21.

[22] Johnson, D., Maltz, D., and Broch, J. DSR: The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks. Ad.-Ws. Ad-hoc Networks., 2001, pp.139-172.

[23] Kim, C.S., Melikov, A., Ponomarenko, L., Baek, J.H. An Analytical Approach for Performance Analysis of Multiservice Cellular Wireless Networks with a Randomized Access StrategyAn Analytical Approach for Performance Analysis of Multiservice Cellular Wireless Networks with a Randomized Access Strategy, Journal of Applied and Computational Mathematics, V.10, N.3, 2011, pp.529-538.

[24] Lien, Y. N. A New Node-Join-Tree Distributed Algorithm for Minimum Weight Spanning Trees, in Proc. of the 8th International Conference on Distributed Computing Systems, 1988, pp.334-340.

[25] Min, M., Wang, F., Du, D.-Z., and Pardalos, P. M. A Reliable Virtual Backbone Scheme in Mobile Ad-Hoc Networks, in Proc. of the 1st IEEE International Conference on Mobile Ad hoc and Sensor Systems (MASS), 2004, pp.60-69.

[26] Park, V. and Corson, S., Temporally-Ordered Routing Algorithm (TORA), Version 1 Functional Specification, 1998.

[27] Perkins, C. E., Belding-Royer, E. M., and Das S. Ad Hoc On Demand Distance Vector (AODV) Routing. RFC 3561, 2003.

[28] Rajaraman, R. Topology control and routing in Ad Hoc networks: A survey, ACM SIGACT News, V.33, N.2, 2002, pp.66-73.

[29] Ryu, J. -H., Song, S., and Cho, D. -H. New Clustering Schemes for Energy Conservation in Two-Tiered Mobile Ad-Hoc Networks, in Proc. of the IEEE ICC01, V.3, 2001, pp.862-66.

[30] Rubin, I., Behzad A., Runhe, Z., and Huiyu, L., Caballero, E. TBONE: A mobile-backbone protocol for ad hoc wireless networks, in Proc. of the IEEE International Conference on Aerospace Conference, V.6, 2002, pp.2727-2740.

[31] Singh, G. and Vellanki, K. A Distributed Protocol for Constructing Multicast Trees, in Proc. of the International Conference on Principles of Distributed Systems, 1998.

[32] Srivastava, S., and Ghosh, R. K. Distributed Algorithms for Finding and Maintaining a k-tree Core in a Dynamic Network, Information Processing Letters, V.88, N.4, 2003, pp.187-194.

[33] Wang, L., and Olariu, S. Cluster Maintenance in Mobile Ad-hoc Networks, Cluster Computing, V.8, N.2-3, 2005, pp.111-118.

[34] West, D. Introduction to Graph Theory. Second edition, Prentice Hall, New Jersey, 2001.

[35] Wu, J., and Li, H. A Dominating-Set-Based Routing Scheme in Ad Hoc Wireless Networks. Telecommunication Systems, V.3, 1999, pp.63-84.

[36] Ya-feng, W., Yin-long, X., Guo-liang, C., and Kun, W. On the Construction of Virtual Multicast Backbone for Wireless Ad hoc Networks, in Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems, 2004, pp.25-27.