Adaptive Backbone Multicast Routing for Mobile Ad hoc Networks*

Chaiporn Jaikaeo
Department of Computer Engineering
Kasetsart University
Bangkok, Thailand
fengchj@ku.ac.th

Chien-Chung Shen
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, U.S.A.
cshen@cis.udel.edu

1 Introduction

Mobile ad hoc networks consist of mobile nodes that autonomously establish connectivity via multihop wireless communications. Without relying on any existing, pre-configured network infrastructure or centralized control, mobile ad hoc networks can be instantly deployed and hence are useful in many situations where impromptu communication facilities are required. Typical examples include battlefield communications and disaster relief missions, in which communication infrastructures are not available or have been torn down. For many applications, nodes need collaboration to achieve common goals and are expected to communicate as a group rather than as pairs of individuals (point-to-point). For instance, soldiers roaming in the battlefield may need to receive commands from a commander (point-to-multipoint), or a group of commanders may teleconference current mission scenarios with one another (multipoint-to-multipoint). Therefore, multipoint communications serve as one critical operation to support these applications.

Many multicast routing protocols have been proposed for mobile ad hoc networks. Of them, several protocols rely on constructing a tree spanning all group members [1, 2], which may not be robust enough when the network becomes more dynamic with less reliable wireless links. In contrast, several protocols have data packets transmitted into more than one link, and allow packets to be received over the links that are not branches of a multicast tree. These protocols fall into the category of *mesh-based* protocols in that group connectivity is formed as a mesh rather than a tree to increase robustness at the price of higher redundancy in data transmission. Flooding, where data packets are forwarded to and received from all links, is also considered a mesh protocol since the mesh is in fact the entire network topology. In highly dynamic, highly mobile ad hoc networks, flooding is a better alternative to multicast routing due to its minimal state and high reliability [3]. To the extreme, flooding provides the most robust, but the least efficient mechanism since a multicast packet will be forwarded to every node as long as the network is not partitioned, while a tree-based approach offers efficiency but is not robust enough to accommodate highly dynamic scenarios. Furthermore, routing based on the concept of connected dominating set [4, 5, 6, 7] can also increase the overall efficiency since the search space is reduced to only nodes in the dominating set during the route discovery and maintenance processes.

Furthermore, most proposed ad hoc multicast routing protocols may only be suitable for specific network conditions, and exhibit drawbacks in other conditions. For instance, protocols which are aggressive in data forwarding and yield high delivery ratio in highly dynamic environments may overkill

*Portion of this chapter was published in the paper titled Adaptive Backbone-Based Multicast for Ad hoc Networks in IEEE International Conference on Communications (ICC), New York City, April 28–May 2, 2002.

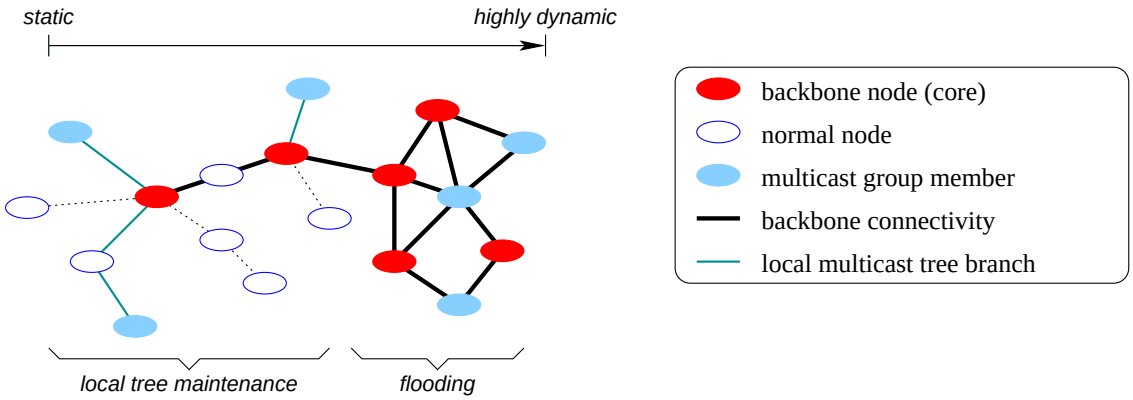


Figure 1: Adaptability of multicast routing inherited from the adaptive backbone construction mechanism: more backbone nodes appear in a more dynamic area than the rest of the network, causing a shift of the forwarding mechanism from tree-based forwarding to flooding in that area.

when the network is relatively static. As a result, this chapter describes a mobility-adaptive multicast routing protocol, called *Adaptive Dynamic Backbone Multicast* or ADBM for short, which offers efficient multicast operations in static or quasi-static environments, and becomes more aggressive and effective in highly dynamic environments, even within the same network. ADBM incorporates a backbone construction algorithm that autonomously extracts a subset of nodes to serve as backbone nodes and provide mobility-adaptive connectivity for multicast operations. Similar to multicast protocols based on rendezvous points [8, 9, 10], flooding of control messages can also be avoided by having each of these backbone nodes function as a *local core* node in its area to limit the amount of control traffic flooding in the process of group joining. Figure 1 illustrates the idea of ADBM, where a single network consists of two portions that are different in terms of mobility or dynamicity. The left portion of the network is relatively static, while nodes on the right side move with higher speed. For instance, a battlefield scenario can be envisioned where soldiers on the right side of the network are engaging the enemy, so that they are moving very quickly, while those on the left side are standing by and not moving too much. Within areas with low mobility, the multicast operation should behave using a tree-based protocol to take advantage of the efficiency of the multicast tree structure. In contrast, a multicast tree is difficult to be efficiently maintained over fast-moving nodes, and hence partial flooding would be more effective.

The remainder of the chapter is organized as follows. Section 2 describes the operations of ADBM, including the adaptive dynamic backbone construction algorithm, termed ADB. Performance study is briefly discussed in Section 3. Related works are presented in Section 4. Section 5 concludes the chapter with exercises.

2 Adaptive Dynamic Backbone Multicast (ADBM)

In this section, we describe how ADBM performs multicast, and how multicast connectivity adapts to various mobility conditions. The key mechanism facilitating ADBM's adaptability is the Adaptive Dynamic Backbone construction algorithm which autonomously constructs and maintains a virtual backbone within a network.

2.1 Adaptive Dynamic Backbone (ADB) Construction Algorithm

Within a network, a virtual backbone infrastructure allows a smaller subset of nodes to participate in forwarding control/data packets and help reduce the cost of flooding. Typical virtual backbone construction algorithms are based on the notion of connected, 1-hop dominating set (CDS), where a node is either a backbone node or an immediate neighbor of a backbone node. However, when considering a wide range of mobility conditions, even within the same network, we have the following observations:

1. Due to higher stability in more static environments, backbone nodes could be allowed to extend their coverage to more than one hop away, resulting in fewer number of nodes participating in backbone-level communications (e.g., flooding control messages over the backbone).

2. In highly dynamic environments, it may not be feasible to maintain connectivity among the backbone nodes when two backbone nodes are connected through several intermediate nodes. In addition, a larger population of backbone nodes is needed to increase reachability to other ordinary nodes.

ADB is designed to specifically address these observations. The operational framework of ADB is derived from the Virtual Dynamic Backbone Protocol (VDBP) [7], which selects a set of backbone nodes based on the 1-hop dominating set property, and chooses intermediate nodes to connect these backbone nodes together to form a connected backbone. However, ADB is not restricted to 1-hop dominating set, but allows a node to be associated with a backbone node that is more than one hop away, depending on stability of the path between that node and its corresponding backbone node. In other words, ADB creates a *forest* of varying-depth trees, each of which is rooted at a backbone node. In high mobility areas, small local groups of only one or two hops in radius will be formed, which reflects the fact that local topology information is changing frequently and should not be collected in a high amount as it will be outdated soon. On the contrary, relatively static areas will result in larger local groups, since more information can be collected and remains steady. The resulting backbone will provide an infrastructure that is adaptive to mobility, which is used to support multicast service for mobile ad hoc networks. Although ADB allows more than one hope coverage, it could behave like other CDS-based protocols, such as VDBP, by setting ADB's parameters to certain values.

ADB consists of three major components: (1) a neighbor discovery process, (2) a core selection process, and (3) a core connection process. The neighbor discovery process is responsible for keeping track of immediate neighboring nodes via HELLO packets. The core selection process then uses this information to determine a set of nodes who will become backbone nodes (or cores[1]). Since cores may not be reachable by each other in one hop, other nodes will be requested to act as intermediate nodes to connect these cores and together form a virtual backbone. This is done by the core connection process. At each node i, the three components are executed concurrently and collaboratively by maintaining and sharing the following variables and data structures:

- $parent_i$: keeps the ID of the upstream node toward the core from i. If i is a core by itself, $parent_i$ is set to i. This variable is initially set to i and is modified by the core selection process. (In other words, every node starts as a core node.)

- NIT_i (Neighboring Information Table at node i): maintains information of all the i's immediate neighbors. Fields of each table entry are described in Table 1. The entry corresponding to the neighbor j is denoted by $NIT_i(j)$. This table is maintained by the neighbor discovery process.

[1]The terms *core* and *backbone node* will be used interchangeably for the rest of this chapter.

Table 1: Fields used by the Neighbor Information Table (*NIT*)

Field	Description
id	Neighbor's node ID
coreId	ID of the core to which this neighbor is associated
hops	Number of hops from this neighbor to its current core
degree	Degree of connectivity
nodeStability	Estimated stability of this neighbor's surrounding area
pathStability	Estimated stability of the path from this neighbor to its core

Table 2: Fields used by the Core Information Table (*CIT*)

Field	Description
coreId	Reachable core's ID
nextHop	Next hop by which the core *coreId* is reachable
hops	Distance in terms of number of hops to that core

- $nlff_i$: stands for *normalized link failure frequency*. It reflects the dynamics of the area surrounding i in terms of the number of link failures per neighbor per second. The neighbor discovery process keeps track of link failures in every *NLFF_TIME_WINDOW* time period and update this variable as follows:

$$current\ nlff_i = \frac{f}{NLFF\_TIME\_WINDOW \times |NIT_i|}, \tag{1}$$

$$nlff_i = \alpha \cdot (current\ nlff_i) + (1 - \alpha) nlff_i, \tag{2}$$

  where f is the number of link failures detected during the last *NLFF_TIME_WINDOW* time period, and α is the smoothing factor ranging between 0 and 1. Initially $nlff_i$ is set to zero.

- CIT_i (Core Information Table at node i): keeps track of a list of nearby cores that are reachable by i, as well as the next hops on shortest paths to reach those cores. Each entry consists of the fields shown in Table 2. This table is maintained by the core connection process.

- CRT_i (Core Request Table at node i): keeps track of a list of nearby cores which i has been requested to establish connection with. The fields used in this table are shown in Table 3. This table is also maintained by the core connection process.

Neighbor Discovery Process

The neighbor discovery process is responsible for keeping track of 1-hop neighboring nodes. To do so, every node broadcasts a HELLO packet every *HELLO_INTERVAL* time period. In addition, HELLO packets also carry extra information to be used by other components of ADB, especially the core selection process. A HELLO packet h_i sent out by node i contains the following fields:

- $h_i.nodeId$: the unique ID of the sending node, i.e., i.

Table 3: Fields used by Core Requested Table (*CRT*)

Field	**Description**
coreId	Requested core's ID
requester	Neighbor who has requested a connection to *coreId*

- $h_i.coreId$: the ID of the core with which i is currently associated.

$$h_i.coreId = \begin{cases} i & \text{if } i \text{ is a core;} \\ NIT_i(parent_i).coreId & \text{otherwise.} \end{cases}$$

- $h_i.hops$: the number of hops away from the core.

$$h_i.hops = \begin{cases} 0 & \text{if } i \text{ is a core;} \\ NIT_i(parent_i).hops + 1 & \text{otherwise.} \end{cases}$$

- $h_i.degree$: the degree of connectivity (the number of neighbors), set to $|NIT_i|$.
- $h_i.nodeStability$: the estimated stability of the area surrounding i. This value is calculated from $nlff_i$ and gives an approximation of the probability that a certain wireless link between i and its neighbor will not break within the next second. The calculation of this value is explained below.
- $h_i.pathStability$: the estimated stability of the path from i to its current core in terms of the probability that this path will still exist within the next second.

$$h_i.pathStability = \begin{cases} 1 & \text{if } i \text{ is a core;} \\ NIT_i(parent_i).pathStability \times nodeStability_i & \text{otherwise.} \end{cases}$$

To ensure that each node and its neighbors will agree on the same information during the core selection process, all the above values are saved into temporary variables whenever a HELLO packet is sent out.

Upon receiving a HELLO packet, the receiving node updates the corresponding entry in its own *NIT*. Due to the fact that nodes may move, each entry is also associated with a timestamp recording the time at which it was last updated. When an entry is older than ($ALLOWED\_LOSSES \times HELLO\_INTERVAL$), it is removed from the table, which also results in a link failure event. Every $NLFF\_TIME\_WINDOW$ time period, the number of link failures is used to calculate *nlff* as shown in (1) and (2) above.

By maintaining *nlff*, each node i is able to estimate the stability of its surrounding area by calculating $nodeStability_i$ which expresses the probability that a certain wireless link between i and its neighbor will not break within the next second. Although GPS could be used to obtain more accurate estimation, using GPS may not be suitable in many situations, especially for smaller devices. Without knowing nodes' geographical locations (via GPS or any other means) or mobility speeds, $nodeStability_i$, which is to be incorporated into each outgoing HELLO packet, is calculated as follows.

For simplicity, we model the time interval node i will wait before detecting the next link failure by the exponential distribution with the average rate of λ_i failures per second. Therefore, the probability that no link failure will be detected within the next t seconds is $e^{-\lambda_i t}$. Since $nodeStability_i$ gives an approximation of the probability that a particular link of i will not break within the next second, if i currently has N_i neighbors, we have

$$(nodeStability_i)^{N_i} = e^{-\lambda_i}.$$

Therefore,

$$nodeStability_i = e^{-\frac{\lambda_i}{N_i}}. \quad (3)$$

Here, λ_i can be estimated by the actual number of link failures detected per second. Since N_i is actually $|NIT_i|$, it follows from (1), (2) and (3) that

$$nodeStability_i = e^{-nlff_i}. \quad (4)$$

Notice that the exponential distribution used here to model link failures may not perfectly predict the actual link failure probability. However, as stated earlier, the desired behavior of ADB is to form smaller clusters in more dynamic areas, and larger clusters in more static areas. Although there exist more accurate mechanisms (e.g., [11]) to predict link stability, this model serves as an easy-to-calculate indicator (due to the memoryless property of the exponential distribution) effective enough[2] for ADB to achieve its objective under different mobility conditions. This has been validated by simulation, as shown in Figure 2 for instance.

The current implementation of the neighbor discovery process assumes that all links are bidirectional and there is a separate mechanism in the underlying communication layer that discards all packets coming from unidirectional links. For instance, nodes might also exchange their lists of neighbors periodically. Each node then accepts only HELLO packets coming from neighbors whose neighbor lists contain its own ID.

Core Selection Process

The core selection process is responsible for extracting a subset of nodes to serve as core nodes and associating other ordinary nodes with these cores. In addition, this process ensures that each ordinary node is within the specified number of hops away from its core. Unlike most of existing cluster head election algorithms, a hop limit is not the only parameter used to determine whether a node is allowed to associate with a certain core. The core selection process also ensures that stability of the path between each ordinary node and its own core satisfies a certain constraint.

The core selection process at each node begins after the node has started for a certain waiting period, usually long enough to allow the node to have heard HELLO packets from all of its neighbors. This process decides whether a node should still be serving as a core, or become a child of an existing core by checking for its own local optimality. First, each node i computes its own *height* from its current status, which is $(nodeStability_i, degree_i, i)$. A height will also be calculated by the node i for each entry in NIT_i. As mentioned in the previous section, when calculating its own height, each node acquires its status that was saved in the temporary variables at the moment the last HELLO packet was sent out, rather than using its current status. Doing so will ensure that all surrounding nodes agree on the same height information, and avoid potential loop formation when the parent-child relationship is established among nodes. If a node has the highest height among its neighbors, it is considered a local optimal node and should continue serving as a core. If not, the node picks the local optimal node as a core and updates its *parent* variable. All subsequent HELLO packets will be changed accordingly.

While keeping the number of cores as low as possible, the core selection process has to ensure that the current backbone configuration satisfies two constraints: the hop count limit and the path stability. These two constraints are used to ensure that the hop count and the path stability from every node to its core must not exceed the parameters *HOP_THRESHOLD* and *STABILITY_THRESHOLD*, respectively. Therefore, once the local optimality check has been made, each node must keep listening to the incoming HELLO packets. For convenience, we define a predicate *MeetConstraints$_i$ (j)* to indicate

[2]Our goal is not to have 'absolute' accurate prediction which may require more complicated computation and additional information to be collected. Instead, we would like to have a simple indicator that can differentiate node and path stability in a 'relative' sense.

whether node i would meet the backbone constraints if it chose node j as its parent. Formally, this predicate is defined as follows:

$$\begin{aligned} &\mathit{MeetConstraints}_i(j) \iff (i = j) \vee \\ &\quad [\mathit{NIT}_i(j).\mathit{hops} < \mathit{HOP\_THRESHOLD} \wedge \\ &\quad \mathit{NIT}_i(j).\mathit{pathStability} > \mathit{STABILITY\_THRESHOLD}\,], \end{aligned}$$

where $\wedge$ and $\vee$ denote the conventional boolean operations "**and**" and "**or**," respectively. Upon receiving a HELLO packet, h_j, from node j, node i modifies its local variables based on the following conditions:

- **Condition 1:** $(j = \mathit{parent}_i) \wedge (\mathit{MeetConstraints}_i(j) = \mathit{FALSE})$.

  This condition implies that the association of node i with its current parent j has now violated the backbone constraints. Therefore, node i has to find a new parent k, where $k \neq j$ and $\mathit{MeetConstraints}_i(k)$ is true, with a minimum number of hops to the core. If no satisfactory parent can be found, node i becomes a core by setting parent_i to i.

- **Condition 2:** $(j \neq \mathit{parent}_i) \wedge (i \neq \mathit{parent}_i) \wedge (h_j.\mathit{hops} < \mathit{NIT}_i(\mathit{parent}_i).\mathit{hops}) \wedge (\mathit{MeetConstraints}_i(j) = \mathit{TRUE})$.

  Here, node i, which is currently not a core, has just heard a HELLO packet from node j who is not its parent but has a shorter distance to the core. Node i then sets parent_i to j.

- **Condition 3:** $(i = \mathit{parent}_i) \wedge (h_j.\mathit{coreId} \neq i) \wedge (\mathit{MeetConstraints}_i(j) = \mathit{TRUE})$.

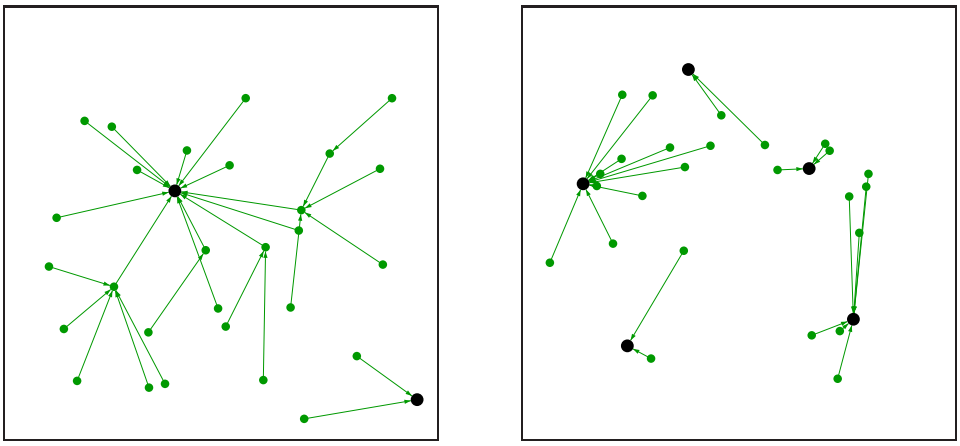
  This condition tells that node i, which is currently a core, has heard a HELLO packet from node j that belongs to another core, and node i could associate itself to j without violating the constraints. If this condition holds, node i will set parent_i to j; otherwise it remains a core.

We can see that while Condition 1 tries to ensure that all nodes who are not cores meet the constraints, it may result in more nodes becoming cores. Meanwhile, Condition 3 attempts to remove these cores by having each current core check if it can find a parent that would not violate the constraints. Therefore, the number of cores in the network will not keep increasing.

In the case of mobility, if a node detects that its parent has moved out of range (from the neighbor discovery process), it will do the same thing as if its parent violates the constraints (i.e., Condition 1). With different mobility conditions, the core selection process will result in different numbers of cores and depths of trees due to the path stability constraint. Figure 2(a) depicts a snapshot of the core selection process on a network of 30 stationary nodes, where there are only two cores and a larger tree is formed. In contrast, node mobility causes formation of smaller trees due to the path stability constraint, which results in more cores, as shown in Figure 2(b).

Core Connection Process

Nodes which are selected to become cores by the core selection process cannot form a connected backbone by themselves, since they may not be reachable by one another within a single hop. The core connection process is responsible for connecting these cores together by designating some nodes to take the role of intermediate nodes. Consequently, the cores and intermediate nodes jointly comprise a virtual backbone, as illustrated in Figure 3. To do so, each core relies on nodes that are located along the border of its coverage, or *border nodes*, to collect information about surrounding nearby cores. A node is said to be a border node if and only if it is able to hear HELLO packets from nodes that are associated with different cores. Each border node then builds a list of cores reachable by



(a) (b)

Figure 2: Different backbone structures under different mobility speeds (the larger black nodes represent the cores): (a) mobility speed is 0 m/s, and (b) mobility speed is 20 m/s.

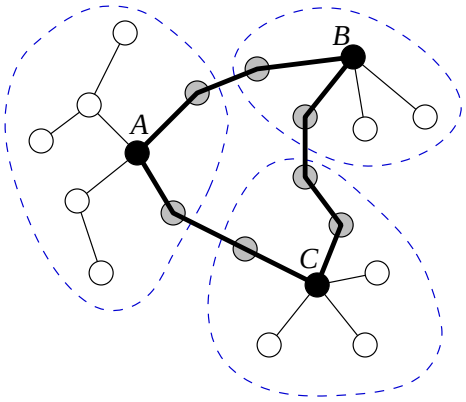


Figure 3: Example illustrating the core connection process: Cores A, B, and C (shown in black) are connected by eight intermediate nodes (shown in gray) and form a virtual backbone (represented by thick lines). Solid lines denote parent-child relationship; and dashed circles denote core coverage boundaries.

itself, including the next hop and the number of hops to each core, and reports this information to its own core so that a core connection request can be performed. To reduce overhead, a border node only needs to collect and report information regarding nearby cores with IDs lower than that of its own core. Hence, a core connection request between a pair of cores is always initiated by the core with the higher ID.

In ADB, the core connection process relies on HELLO packets to carry two extra pieces of information: COREINFO and COREREQUEST tables. Algorithm 1 (presented in pseudo code) illustrates how this information is piggybacked onto outgoing HELLO packets, and Algorithm 2 describes how this information is processed at receiving nodes. The following is the detailed explanation. Let $coreId_i$ be $NIT_i(parent_i).coreId$ if i is not a core, or i otherwise. When node i hears a HELLO packet, h_j, from another node j and $h_j.coreId_j < coreId_i$, node i will insert into its CIT_i (Core Information Table) an entry $\langle h_j.coreId_j, j, h_j.hops{+}1\rangle$, which represent *coreId*, *nextHop*, and *hops* fields, respectively, as described in Table 2. This step corresponds to lines 16–17 of Algorithm 1. To propagate this information to the core, node i piggybacks onto its outgoing HELLO packet a COREINFO table containing a list of entries $\langle coreId,hops\rangle$. Each entry indicates the shortest distance in terms of the number of hops from node i to each unique core in CIT_i, as shown in lines 21–24 of Algorithm 2. Note that CIT_i may contain multiple entries for the same core, but node i will choose only one with the least number of hops to that core. When any node j hears the HELLO packet containing the COREINFO from node i, node j looks for entries with lower core IDs than its own and inserts/updates the corresponding entries in CIT_j using i as the *nextHop* field as long as the following two conditions hold: (1) nodes i and j belong to the same core, and (2) the distance from node j to the core is less than that of node i. Lines 18–21 of Algorithm 1 denotes these steps. Nodes who are not cores and having nonempty *CIT* tables must always generate and piggyback COREINFO tables onto outgoing HELLO packets so that each core can collect all information it needs, and establish connectivity to nearby cores (again, in lines 21–24 of Algorithm 2). Due to the dynamics of the core selection process and the network topology itself, the *CIT* table is kept updated by having each node remove all the entries with *nextHop* field set to k when it receives a link failure notification regarding to the node k. In addition, it also removes all the entries in *CIT* that match the pattern $\langle c, k, \cdot\rangle$ when it hears a COREINFO from k without any entry regarding the core c.

Once a core has had information about other surrounding cores stored in its *CIT* table, it is able to initiate the connection request to those cores by piggybacking a COREREQUEST table onto each of its outgoing HELLO packets. A COREREQUEST table contains a list of entries $\langle coreId,nextHop\rangle$, each of which indicates which node is being requested to connect to each unique core from the *CIT* table, as shown in lines 8–15 of Algorithm 2. Similar to the construction of a COREINFO table, only one next hop with the shortest distance in terms of the number of hops is chosen for each core. Entries from *CIT* with one hop count are excluded from the COREREQUEST table since they represent other cores with direct contact, which do not require any intermediate nodes to establish connectivity with. When node i receives a HELLO packet containing a COREREQUEST table from node j, node i checks for entries where its ID appears on the *nextHop* field. For each entry $\langle c, i\rangle$ found in the COREREQUEST, node i inserts into CRT_i (Core Request Table) an entry with *coreId* and *requester* set to c and j, respectively (Algorithm 1, lines 23–25). Each node i who is not a core and has at least one entry in its CRT_i is required to construct a COREREQUEST table with entries corresponding to all the cores in CRT_i (Algorithm 2, lines 17–20). For each entry $\langle c, n_c\rangle$ in the COREREQUEST table that node i is to generate, if node c is the current core of node i, node i will use $parent_i$ as the value of n_c; otherwise node i will consult its CIT_i for the best next hop to node c.

Similar to the maintenance of *CIT*, when node i detects that the link to node k has failed, node i removes from CRT_i all the entries whose *requester* fields are equal to k. If CRT_i contains an entry $\langle c, k\rangle$ but node i hears a COREREQUEST table from node k without an entry $\langle c, i\rangle$, the entry $\langle c, k\rangle$ is removed from CRT_i.

Algorithm 1 Node i processing HELLO packets

1: **Input:**
2: $h \leftarrow$ incoming HELLO
3: $src \leftarrow$ the sender of h
4: **Begin:**
5: *coreInfo* $\leftarrow$ COREINFO piggybacked by h
6: *coreReq* $\leftarrow$ COREREQUEST piggybacked by h
7: **if** $\text{parent}_i = i$ **then** /* i is a core */
8: $\quad coreId_i \leftarrow i$
9: $\quad hops_i \leftarrow 0$
10: **else** /* i is not a core */
11: $\quad coreId_i \leftarrow NIT_i(parent_i).coreId$
12: $\quad hops_i \leftarrow NIT_i(parent_i).hops + 1$
13: **end if**
14: remove all entries $\langle cid,\ nextHop,\ hops \rangle$ from CIT_i **where** $nextHop = src$
15: remove all entries $\langle cid,\ requester \rangle$ from CRT_i **where** $requester = src$
16: **if** $h.coreId < coreId_i$ **then**
17: $\quad$ insert $\langle h.coreId,\ src,\ h.hops{+}1 \rangle$ into CIT_i
18: **else if** $h.coreId = coreId_i$ & $h.hops > hops_i$ **then**
19: $\quad$ **for each** $\langle cid, hops \rangle$ in *coreInfo*, **where** $cid < coreId_i$ **do**
20: $\quad\quad$ insert $\langle cid,\ src,\ hops{+}1 \rangle$ into CIT_i
21: $\quad$ **end for**
22: **end if**
23: **for each** $\langle cid, nextHop \rangle$ in *coreReq* **where** $nextHop = i$ **do**
24: $\quad$ insert $\langle cid, src \rangle$ into CRT_i
25: **end for**

Algorithm 2 Node i piggybacking CoreInfo and CoreRequest

Input:
$h \leftarrow$ outgoing Hello
Output:
outgoing Hello with piggybacked info

Begin:
$coreInfo \leftarrow \{\}$
$coreReq \leftarrow \{\}$
if $parent_i = i$ **then** /* i is a core */
  for each $\langle cid,\ nextHop,\ hops \rangle$ in CIT_i **do**
    $d \leftarrow$ distance to the core cid from i
    $best \leftarrow$ best next hop to the core cid from i
    if $d > 1$ **then**
      $coreReq \leftarrow coreReq \cup \{\langle cid, best \rangle\}$
    end if
  end for
else /* i is not a core */
  for each $\langle cid,\ requester \rangle$ in CRT_i **do**
    $best \leftarrow$ best next hop to the core cid from i
    $coreReq \leftarrow coreReq \cup \{\langle cid, best \rangle\}$
  end for
  for each $\langle cid,\ nextHop,\ hops \rangle$ in CIT_i **do**
    $d \leftarrow$ distance to the core cid from i
    $coreInfo \leftarrow coreInfo \cup \{\langle cid, d \rangle\}$
  end for
end if
piggyback $coreInfo$ and $coreReq$ onto h
return h

Table 4: Fields used by the Join Table (JT)

Field	Description
groupId	Multicast group ID
requester	Node who has requested to join this group

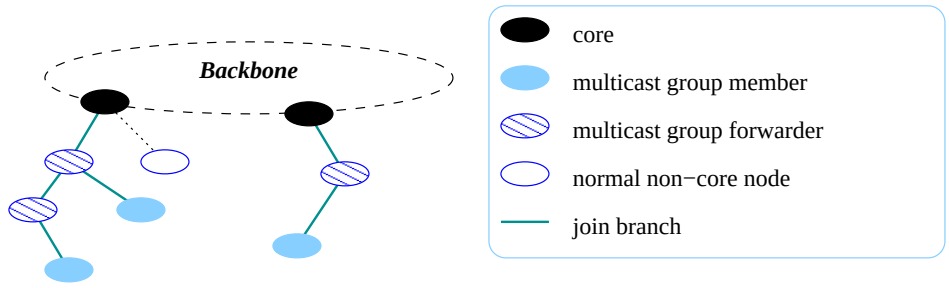


Figure 4: An example showing nodes forming of multicast trees, which are connected by the backbone, during group joining process

Node i determines if it is currently on the backbone by checking if it is either a core ($parent_i = i$) or an intermediate node ($|CRT_i| > 0$). These nodes on the backbone play an important role for facilitating multicast operations, which will be described in the next section.

2.2 Group Joining

The group joining mechanism in ADBM is based on the concept of rendezvous points, where control messages are directed toward the core of the group instead of being broadcast to other irrelevant nodes or the entire network. Here, the rendezvous point for every node is not a single node, but it consists of the cores that have already formed a connected backbone as a result from ADB. An node who is willing to join a multicast group sends a request toward its core via their parents. Intermediate nodes who receive and forward these requests then become forwarding nodes for the group, which, together with the connected backbone, eventually form multicast connectivity among all group members. We explain the group joining operation in more details as follows.

In addition to the data structures for ADB described in Section 2.1, each node maintains a *join table* which has the same structure as a core request table (Table 3) with a group ID field instead of a core ID, as shown in Table 4. For node i, its join table is denoted by JT_i. Node i who is willing to join a multicast group informs its parent by piggybacking a JoinRequest, along with its parent ID, $parent_i$, onto its outgoing Hello packets. The JoinRequest contains a list of all multicast group IDs that node i has joined and is willing to join. The JoinRequest also includes group IDs of which i is currently a forwarding node. When a node j receives a Hello with a JoinRequest, it checks if its ID is specified as the intended receiver of the JoinRequest. If so, node j updates its JT_j with all the entries found in the JoinRequest received from node i, using i as the *requester* field. It is required that nodes who are members of some groups, and any nodes whose join tables are not empty, with the exception for cores, periodically broadcast JoinRequest with their Hello messages. A node who has an entry for the group ID g in its join table then marks itself as a forwarding node for group g, and nodes on the backbone are considered forwarding nodes for any group. As a result, these forwarding

nodes form a multicast tree within each core's coverage, where multiple trees are connected by the backbone, hence establishing connectivity among the group members, as illustrated in Figure 4.

To deal with network dynamics, the join table is maintained in the same way as ADB does to *CIT* and *CRT* in that entries whose *requester* fields are i are removed from the table whenever a link to node i has broken. The entry with *groupId* and *requester* set to g and i, respectively, is also removed when a JoinRequest received from node i contains no entry for the group g.

2.3 Multicast Packet Forwarding

Although multicast connectivity establishment within a core's coverage resembles a tree formation, ADBM protocol allows forwarding nodes and members to accept data packets that arrive from any node. Hence it is considered a mesh-based protocol. Since there can exist more than one path between a sender and a receiver, each data packet is assigned a unique sequence number when it is leaving its source. The sequence numbers are checked by each forwarding node and member node to ensure that no duplicate data packets are rebroadcast or delivered to the application. When a node i receives a non-duplicate data packet for the group g, it checks whether it is currently a forwarding node of the group, i.e., whether one or more of the following conditions hold:

- node i is a core, i.e., $parent_i = i$.
- node i is an intermediate node on the backbone, i.e., $|CRT_i| > 0$.
- node i has an entry for g in its join table.

If so, node i rebroadcasts the packet. Otherwise, the packet is silently discarded. If node i is a member of the group g and none of the above conditions hold, node i will deliver the packet to the application without rebroadcasting.

3 Performance Discussion

When studying performance of most routing protocols, either unicast or multicast, we are usually interested in two different aspects: *effectiveness* and *efficiency*. The former captures how effective the protocol is in delivering packets to the destination(s). The latter typically focuses on how much more overhead is spent to get the job done. Base on these two aspects, we haved measured the performance of ADBM using the following metrics.

- **Packet delivery ratio**: The ratio of the number of non-duplicate data packets successfully delivered to the receivers versus the number of packets supposed to be received. This metric reflects the effectiveness of a protocol.
- **Number of total packets transmitted per data packet received**: The ratio of the number of data and control packets transmitted versus the number of data packets successfully delivered to the application. Hello packets are also considered as packets transmitted. This measure shows efficiency of a protocol in terms of channel access. The lower the number, the more efficient the protocol.
- **Number of total bytes transmitted per data byte received**: This metric is similar to the second metric except that the number of bytes is considered instead. Here, bytes transmitted include everything that is sent to the MAC layer (i.e., IP and UDP headers, as well as Hello packets), where data bytes received involve only the data payloads. This metric presents efficiency of a protocol in terms of bandwidth utilization. Similar to the second metric, the lower the number, the more efficient the protocol.

Now let us briefly discuss ADBM's performance. As the protocol was designed to be adaptive with wide range of node mobility, we have studied its behavior under various mobility speeeds. Overall, the study showed that ADBM performs efficiently (i.e., most packets are successfully delivered) under low mobility and still effectively under extremely high mobility with various network and application settings. However, choosing appropriate values for *HOP_THRESHOLD* and *STABILITY_THRESHOLD* is the key to achive good tradeoff between effectiveness and efficiency of the protocol. For *HOP_THRESHOLD*, too low values will cause many nodes to become cores even when the nodes are stationary, which is not good in terms of efficiency. On the contrary, higher hop thresholds should give better results in general as less nodes will be on the backbone under low mobility; and with high mobility, more nodes will be forced to join the backbone, which sustains the protocol's effectiveness, given that *STABILITY_THRESHOLD* is set properly.

STABILITY_THRESHOLD is another important parameter we need to adjust. Setting this threshold too low makes the nodes very sensitive to packet losses (which are caused not only by mobility, but packet collisions as well), and the network could end up with many more forwarding nodes than necessary. In contrast, high *STABILITY_THRESHOLD* values will potentially reduce packet delivery ratio as mobility increases since large coverage of cores prevent core connectivity (as well as multicast connectivity) from reacting to mobility quickly enough.

4 Related Work

This section surveys related multicast as well as virtual backbone protocols for ad hoc networks. There have been a number of techniques proposed for multicast support over ad hoc networks, ranging from a simple flooding scheme to tree-based or mesh-based approaches. AMRoute (Ad Hoc Multicast Routing Protocol) [1] follows the tree-based approach by relying on an underlying ad hoc unicast routing protocol to provide unicast tunnels for connecting the multicast group members together. AMRIS (Ad Hoc Multicast Routing with Increasing Sequence Numbers) [2] and MAODV (Multicast Ad Hoc On-Demand Distance Vector Protocol) [12] are also tree-based but they are independent from unicast routing protocols while maintaining multicast trees. ODMRP (On Demand Multicast Routing Protocol) [13] and CAMP (Core-Assisted Multicast Protocol) [10] allow data packets to be forwarded to more than one path, resulting in mesh structures to cope with link failures and increase robustness in dynamic environments. However, one major disadvantage of ODMRP and other on-demand approaches is their frequent flooding of control messages. For example, in ODMRP senders periodically flood the entire network with query messages to refresh group membership. Dynamic Core based Multicast routing Protocol (DCMP) [14] mitigates this problem by having only a small subset of senders act as core for the other senders to limit the amount of flooded messages. Although CAMP also suggests the use of cores to direct the flow of join request messages, a mapping service from multicast groups to core addresses is assumed, and a core may not be available at all times. Furthermore, CAMP still relies on an underlying unicast routing protocol which provides correct distances to known destinations within a finite time. Instead of specifying an explicit core address and relying on a unicast protocol, ADBM employs the parent-child relationship to direct a join request from a node toward its core, which prevents the message from being globally flooded.

For scalability, a hierarchical approach or a hybrid approach is often employed. MZR (Multicast routing protocol based on Zone Routing) [15] adopts a hybrid approach using the same mechanism provided by the Zone Routing Protocol (ZRP) [16]. A *zone* is defined for each node with the radius in terms of the number of hops. A proactive protocol is used inside each zone, while reactive route queries are done at the zone border nodes on demand, resulting in a much smaller number of nodes participating in the global flooding search. CGM (Clustered Group Multicast) [17] and MCEDAR (Multicast Core-Extraction Distributed Ad hoc Routing) [6] employ a similar concept by having only a

subset of network involved in multicast data forwarding. This is done by extracting an approximated 1-hop dominating set from the network to be used for control and/or data messages. Also based on a dominating set concept, a number of virtual backbone protocols, such as B-protocol [18] and VDBP [7], provide a generic support for creating and maintaining a virtual backbone over an ad hoc network. They can therefore serve as a mechanism to reduce the communication overhead. Amis et al. [19] proposed a generalized clustering algorithm based on a d-hop dominating set, which allows a node to be at most d hops away, rather than only one hop, from its backbone node. However, these protocols have a similar restriction in terms of local group sizes. Clustering algorithms based on a 1-hop dominating set always require each ordinary node to be exactly one hop away from the nearest backbone node. In relatively static environments, a node should be allowed to connect to a backbone node that is more than one hop away to lower the number of backbone nodes and reduce overhead of multicast data forwarding. On the other hand, a d-hop clustering algorithm has a fixed value for the parameter d, which always attempts to associate nodes to backbone nodes as long as they are within d hops away, regardless of their surroundings' mobility condition. Since each pair of nearby backbone nodes must be connected through several intermediate nodes (up to $2d + 1$ hops), it may be infeasible to maintain backbone connectivity in highly dynamic environments. Thus, these existing protocols are not adaptive in a situation where a network has the level of mobility changing from time to time, or where there are dynamic and static portions coexisting in the same network. In contrast, ADBM uses two parameters, *HOP_THRESHOLD* and *STABILITY_THRESHOLD*, to determine whether a node should be attached to an existing backbone node. In static environments, where nodes do not experience many link failures, the stability criterion is always satisfied. The *HOP_THRESHOLD* parameter is, therefore, the only parameter controlling the backbone formation, making ADBM's backbone construction algorithm (i.e., ADB) similar to a typical d-hop clustering algorithm. On the contrary, *HOP_THRESHOLD* rarely affects the backbone formation in highly dynamic environments due to decreasing of node and path stability that makes nodes likely to violate the stability criterion before the hop-limit criterion is violated. Hence, nodes in high mobility areas are likely to be controlled by the *STABILITY_THRESHOLD* parameter when deciding which backbone nodes they should attach to, or whether they should become backbone nodes themselves. As a result, ADBM achieves adaptability by allowing nodes in static areas to be up to *HOP_THRESHOLD* away from their backbone nodes (to reduce overhead), and implicitly lowering the hop limit for nodes in dynamic areas (to increase robustness of virtual backbones).

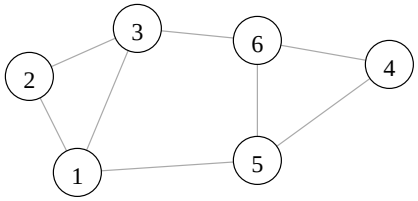
McDonald and Znati introduced an adaptive clustering framework, called (α, t) *cluster* [20], to support a mobility-adaptive hybrid routing architecture. This framework partitions the network into clusters of nodes and ensures that every node in each cluster has a path to every other node in the same cluster for the next time period t with probability α. Hence, with different mobility conditions, cluster sizes are adjusted accordingly. Although ADB and (α, t) cluster share similar objectives in providing mobility-adaptive infrastructures for hybrid routing, there are several differences between the two protocols. First, the design of the (α, t) cluster framework mainly aims to support unicast routing, while ADB is intended for multicast routing. Second, in the (α, t) cluster framework, a node has to ensure that its associativity with every other node in the same cluster meets certain criteria. ADB, on the other hand, requires a node to maintain stability constraints with only one node in the cluster, i.e., the backbone node. And finally, to maintain cluster membership, the (α, t) cluster framework relies on routing information provided by an underlying proactive routing protocol, while ADB is independent of any routing protocol to operate.

5 Conclusion

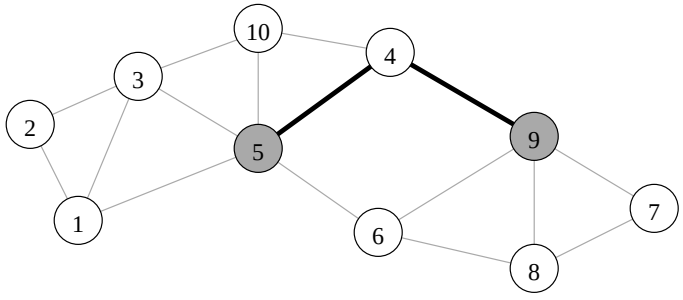
We have introduced a mobility-adaptive multicast protocol for mobile ad hoc networks, called Adaptive Dynamic Backbone Multicast, or ADBM. The protocol, based on a two-tier architecture, incorporates the Adaptive Dynamic Backbone (ADB) algorithm to construct a virtual backbone infrastructure that facilitates multicast operations. ADB extracts a subset of nodes, called backbone nodes or cores, out of the network. These cores then establish connectivity among one another to form a virtual infrastructure. Each core is allowed to have larger coverage in relatively static areas, while the coverage is reduced in more dynamic areas so that connectivity among cores can be efficiently established and quickly reestablished under high mobility. ADBM employs the provided backbone infrastructure to achieve adaptive multicast routing which integrates a tree-based scheme and a flooding scheme together to provide both efficiency in static areas and robustness in dynamic areas, even within the same network. Since every node is either a core or has an association to a core via its parent node, cores are also used to avoid global flooding of control messages triggered by a node's joining a group or maintaining group membership, as found in other on-demand multicast protocols.

Exercises

1. Consider a network of 6 stationary nodes running ADB, as shown below. Gray lines indicate that nodes are neighbors. Assuming that all nodes have already heard of their neighbors' *HELLO* packets at least once and never encountered a lost *HELLO*, and the core selection process has *not yet* been triggered. What is the content of a *HELLO* packet sent out by node 6?



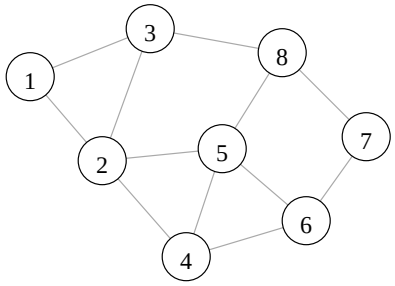
2. The diagram below illustrates a network of 10 nodes with a *stable* backbone formation. Gray lines indicate that nodes are in each other's vicinity, black lines indicate the current backbone formation, and gray nodes are current serving as cores. Given that every node has the normalized link failure frequency (*nlff*) of 0.2 link failure per neighbor per second, show the Neighbor Information Tables (NIT) of nodes with IDs 1 and 7.



3. The following table summarizes current node information right before the core selection process is triggered. Which of the five nodes will continue serving as cores *immediately after the core selection process was triggered*?

Node	Neighbors	*nlff*
1	{2}	0
2	{1,3,5}	0
3	{2,4}	0.1
4	{3,5}	0
5	{2,4}	0.1

4. Suppose ADB is running on a network represented by the graph below:



   Let every node have *nlff* of 0, and ADB's parameters in the following table be used:

Parameter	Value
HOP_THRESHOLD	3
STABILITY_THRESHOLD	0.9

   Compute the final backbone formation. (In case there are more than one possible formations, show only one.)

5. Repeat the previous exercise, but this time suppose every node now has *nlff* of 0.1 link failure per neighbor per second instead of 0.

6. Consider a network of 9 nodes with partial ADB-related information shown in the table below. If nodes 1 and 9 join the same multicast group, which of the nine nodes will be considered forwarding nodes by ADBM?

Node	Neighbors	Parent computed by ADB
1	{4}	4
2	{3}	3
3	{2,4,5}	3
4	{1,3,5}	3
5	{3,4,6}	3
6	{5,7}	7
7	{6,8}	7
8	{7,9}	7
9	{8}	8

7. Given a network of C cores running ADBM with a multicast group of M members. Suppose, on average, the coverage of each core is H hops and is connected to D other cores, and the average hop count from an individual node to its core is $\overline{H}$. How many times a multicast packet is transmitted when it is sent from a source node to all other members, assuming there is no packet loss in the delivery?

References

[1] Mingyan Liu, Rajesh R. Talpade, and Anthony McAuley, "AMRoute: Adhoc Multicast Routing Protocol," Tech. Rep. 99, The Institute for Systems Research, University of Maryland, 1999.

[2] C.W. Wu and Y.C. Tay, "AMRIS: A Multicast Protocol for Ad hoc Wireless Networks," in *IEEE Military Communications Conference (MILCOM)*, Atlantic City, NJ, November 1999, pp. 25–29.

[3] C. Ho, K. Obraczka, G. Tsudik, and K. Viswanath, "Flooding for Reliable Multicast in Multihop Ad Hoc Networks," in *The 3rd Intl. workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M'99)*, August 1999.

[4] Raghupathy Sivakumar, Bevan Das, and Vaduvur Bharghavan, "Spine Routing in Ad hoc Networks," *ACM/Baltzer Publications Cluster Computing Journal*, vol. Special Issue on Mobile Computing, 1998.

[5] P. Sinha, R. Sivakumar, and V. Bharghavan, "CEDAR: A Core-Extraction Distributed ad hoc Routing Algorithm," in *IEEE INFOCOM'99*, March 1999.

[6] Prasun Sinha, Raghupathy Sivakumar, and Vaduvur Bharghavan, "MCEDAR: Multicast Core-Extraction Distributed Ad hoc Routing," in *IEEE Wireless Communications and Networking Conference (WCNC) '99*, New Orleans, LA, September 1999, pp. 1313–1317.

[7] Ulas C. Kozat, George Kondylis, Bo Ryu, and Mahesh K. Marina, "Virtual Dynamic Backbone for Mobile Ad Hoc Networks," in *IEEE International Conference on Communications (ICC)*, Helsinki, Finland, June 2001.

[8] Tony Ballardie, Paul Francis, and Jon Crowcroft, "Core-based trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing," in *Communications, architectures, protocols, and applications*, San Francisco, CA, USA, 13-17 September 1993.

[9] R. Royer and C. Perkins, "Multicast using ad-hoc on demand distance vector routing," in *Mobicom'99*, Seattle, WA, August 1999, pp. 207–218.

[10] J. Garcia-Luna-Aceves and E. Madruga, "The Core-Assisted Mesh Protocol," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 8, August 1999.

[11] Michael Gerharz, Christian de Waal, Matthias Frank, and Peter Martini, "Link Stability in Mobile Wireless Ad Hoc Networks," in *Proceedings of the IEEE Conference on Local Computer Networks (LCN) 2002*, November 2002.

[12] Elizabeth M. Royer and Charles E. Perkins, "Multicast Operation of the Ad-Hoc On-Demand Distance Vector Routing Protocol," in *Mobile Computing and Networking*, 1999, pp. 207–218.

[13] S. Bae, S. Lee, W. Su, and M. Gerla, "The Design, Implementation, and Performance Evaluation of the On-Demand Multicast Routing Protocol in Multihop Wireless Networks," *IEEE Network, Special Issue on Multicasting Empowering the Next Generation Internet*, vol. 14, no. 1, pp. 70–77, January/February 2000.

[14] Subir Kumar Das, B. S. Manoj, and C. Siva Ram Murthy, "A Dynamic Core Based Multicast Routing Protocol for Ad hoc Wireless Networks," in *The ACM Symposium on Mobile Adhoc Networking and Computing (MOBIHOC 2002)*, Lausanne, Switzerland, June 9–11 2002.

[15] Vijay Devarapalli and Deepinder Sidhu, "MZR: A Multicast Protocol for Mobile Ad Hoc Networks," in *IEEE International Conference on Communications (ICC)*, Helsinki, Finland, June 2001.

[16] Z. J. Haas and M. R. Pearlman, "The zone routing protocol (ZRP) for ad hoc networks," 1997, IETF Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-manet-zone-zrp00.txt.

[17] Chunhun Richard Lin and Shiang-Wei Chao, "A Multicast Routing Protocol for Multihop Wireless Networks," in *IEEE Global Telecommunications Conference (GLOBECOM)*, Rio de Janeiro, Brazil, December 1999, pp. 235–239.

[18] Stefano Basagni, Damla Turgut, and Sajal K. Das, "Mobility-Adaptive Protocols for Managing Large Ad Hoc Networks," in *IEEE International Conference on Communications (ICC)*, Helsinki, Finland, June 2001.

[19] Alan D. Amis, Ravi Prakash, Thai H.P. Vuong, and Dung T. Huynh, "Max-Min D-Cluster Formation in Wireless Ad Hoc Networks," in *IEEE INFOCOM 2000*, Tel-Aviv, Israel, March 2000, pp. 32–41.

[20] A. Bruce McDonald and Taieb F. Znati, "A Mobility-Based Framework for Adaptive Clustering in Wireless Ad hoc Networks," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 8, August 1999.