An Implementation Study of Multicast Extensions of AODV

Dinesh Dharmaraju Manish Karir John S. Baras
Center for Satellite and Hybrid Communication Networks
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742, USA
{dineshd, karir, baras}@isr.umd.edu

Subir Das
Telcordia Technologies,
Morristown, NJ 07960, USA
subir@research.telcordia.com

Keywords: multicast routing, MAODV, ad hoc routing, implementation study, AODV, protocols, wireless networks

Abstract
In this paper we describe our experience of implementing multicast extensions of AODV (MAODV). Using insight gained from our implementation we propose some enhancements to MAODV that enhance its stability and its performance. We propose the use of a tree optimization scheme based on Group Hello messages, a scheme for merging group partitions, as well as a method to add reliability to the multicast tree formation and maintenance, via the use of an acknowledgement for MACT messages. Our implementation is based on the kernel AODV implementation from NIST. We describe details regarding our implementation that can provide valuable insight to future implementers of ad hoc routing protocols, and in particular, people attempting to implement multicast routing protocols. We also provide a brief description of some basic validation experiments that we used to verify the functionality and the various features of our implementation.

1. INTRODUCTION

Wireless networks are being deployed at a very rapid pace. In contrast with the infrastructure-based networks that are being deployed, there is an entire category of infrastructureless networks, which are being designed to operate in situations where it might not be possible to have an infrastructure. By their very nature, wireless networks are essentially broadcast in nature. All nodes within the transmission radius of a node can overhear every transmission by this node. Therefore, the most efficient use of wireless bandwidth is via the use of broadcast from a single node. While this is the most efficient use of bandwidth, it is not the most useful as often communication only needs to be between two specific nodes. Unicast transmissions on the other hand are the least efficient use of the wireless channel. An interesting and useful compromise between these two extremes is the use of multicast. Multiple receivers who subscribe to be part of the receiver group can receive using multicast a single transmission. The problem of multicast routing is born out of the need of achieving multicast capability in a scenario where all the nodes interested in participating in the multicast group are not within the transmission range of the sender. There needs to be some mechanism to forward multicast traffic through the entire multi-hop network, based on group member information. To solve this problem in wireless networks, several multicast protocols have been proposed such as MAODV [1], AMROUTE [2], LAM [3], and ODMRP [4].

Most protocols proposed for multicasting in wireless ad hoc networks depend on some underlying unicast protocol. Therefore, when designing a multicast routing protocol one might be able to rely on some structure already provided by the unicast routing protocol. One of the strongest candidates for standardization, by the MANET working group at the IETF, as the unicast protocol of choice in wireless ad hoc networks is AODV. In this paper, we describe our experience in implementing and validating the multicast extensions of AODV (MAODV). We also propose some modification to the protocol that enhances its stability and reliability.

The rest of this paper is organized as follows: Section 2 describes some selected related work in the areas of not only multicast routing protocols but also unicast routing protocols in wireless ad hoc networks, in particular similar work in implementation of routing protocols in wireless ad hoc networks. Section 3 provides a brief overview of the MAODV protocol. Section 4 describes our implementation of the multicast extensions of AODV in Linux, as well our proposed modifications to the protocol. Section 5 describes some of the basic validation experiments we have used to attempt to verify correct operation of the MAODV protocol. In section 6 we describe some measurements from our implementation as a measure of its efficiency. Finally, section 7 provides our conclusions and some directions for future work.

2. RELATED WORK

Though several multicast routing protocols have been proposed for ad hoc networks, ODMRP [4] is different from others, in that it is mesh based and is independent of the underlying unicast routing protocol. Most multicast

The material described in this paper is based upon work performed in collaboration with Telcordia Technologies and sponsored by the U.S. Army Research Laboratory under award number DAAD17-00-C-0115. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

routing protocols however, rely on some underlying unicast routing mechanism. These protocols take advantage of the underlying unicast routing protocol and save on control overhead. LAM [3] is a group shared tree protocol that does not require timer-based messaging. LAM is tightly coupled with the underlying unicast routing protocol TORA [5], and relies on TORA's unicast route finding capability. AMROUTE [2] is a shared-tree protocol, which allows dynamic core migration based on group membership and network configuration.

By its very nature MAODV [1][6] is integrated with AODV [7], which is a strong candidate for standardization in the IETF. This makes it a good choice to investigate issues in implementing multicast routing protocols in ad hoc networks. There have been several simulation studies of MAODV [1] [8] however; implementation of a routing protocol in a real ad hoc network can often provide details that network simulations can skip.

Unicast versions of AODV have been widely studied and implemented. A number of public domain implementations for AODV are available [9][10][11][12]. MADHOC [9] is an implementation of AODV, which is entirely based in user space. Kernel AODV [10] on the other hand is implemented entirely in kernel space. The implementations outlined in [11] and [12] are based partly in user space and partly in kernel space. The advantage of the kernel space implementations is that there is easy access of all the packets and their headers. This makes it easier to utilize header fields and options when making protocol decisions. Kernel space implementations also often tend to be more efficient as they do not have to transfer data back and forth between user space and kernel space. However, kernel implementations of protocols are more difficult to implement, and are less portable, making them more difficult to maintain. The implementation in [13] is largely a user space implementation of AODV and MAODV but provides only few implementation details. In this paper, we present the kernel space implementation of MAODV and its validation. We also propose several mechanisms to enhance the performance of the base protocol, which was presented in [13].

3. OVERVIEW OF MULTICAST AODV

The operation of MAODV is analogous to the operation of AODV. Multicast routes are discovered on demand. The multicast route request is broadcast similar to the unicast route request, and the route reply propagates back from the nodes that are members of the multicast group. The MAODV Internet draft [6] and [1] describe in detail the operation of MAODV. For completeness, we summarize some of the basics features and operation of AODV and MAODV below. A more detailed description can be found in [1] [6] [7].

Every node running MAODV maintains two routing tables. The first one is used for unicast operation of AODV and is simply referred to as *Route Table*. The fields of the routing table are as follows: Destination IP Address, Destination Sequence Number, Hop Count to Destination, Last Hop Count, Next Hop, Next Hop Interface, List of Precursors, Lifetime, and Routing Flags. [6] [7] give a detailed description of these fields and how they are used for unicast route discovery. The second routing table is the *Multicast Routing Table*. This contains the following fields:

- Multicast Group IP Address
- Multicast Group Leader IP Address
- Multicast Group Sequence Number
- Next Hops
- Hop Count to next Multicast Group Member
- Hop Count to Multicast Group Leader

Every node that is a router for a multicast group maintains a *Multicast Route Table* entry for that group. In section 4.4, we describe the additional fields that we have added to the *Multicast Route Table* to facilitate implementation and correct operation of the protocol.

For every multicast group, a bi-directional shared tree is formed, consisting of the members of the multicast group and the intermediary nodes. Each multicast group has a group leader associated with it. The primary function of the group leader is to maintain and disseminate the multicast group sequence number. This sequence number is used to indicate the freshness of routing information for the multicast group.

When a node wishes to join a multicast group, it broadcasts a RREQ message with the *join* flag set. When intermediary nodes receive the RREQ, they create an entry in their Multicast Route Table, for that group, with the *Next Hop* field set to the IP address of the node from which it received the RREQ. It also sets the *downstream* flag for this Next Hop entry. If this node is a member of the multicast tree, for this group, it will unicast a RREP back to the originator of the RREQ. Otherwise, it rebroadcasts the RREQ. A node that receives a RREP for a multicast group, forwards it, and it also adds an entry in its Multicast Route Table for that groups Next Hop list, with an IP address of the node from which it received the RREP, and also marks this entry with the *upstream* flag.

A node wishing to join the multicast group waits for a time period of RREP_WAIT_TIME for RREPs from different nodes. After this interval, it selects the best route (in terms of freshness and distance from the multicast tree). It then sends a unicast MACT message along the selected best route. When a node receives a MACT for a particular group, it checks to see if it is a member of the multicast tree. If the node is already a part of the multicast tree, the *Next Hop* from which the MACT was received is activated, thus grafting the link on to the multicast tree. If it is not part of the multicast tree, it forwards the MACT to its *upstream*

Next Hop, activating the *upstream Next Hop*, and making it a part of the multicast tree. Processing continues in this manner until a node that was already a part of the multicast tree is reached and the addition of the tree branch is complete. If the node originating the RREQ does not receive an RREP before timing out, it broadcasts another RREQ with *Broadcast ID* increased by one. The node generates up to RREQ_RETRIES number of RREQs. If the node doesn't receive an RREP by this time, it assumes that the multicast group does not already exist, becomes the group leader for that group, and starts broadcasting periodic *Group Hello* messages for the multicast group.

When a link break is detected as described in [1][6], the node downstream of the link breakage tries to repair the link by sending RREQs with *Multicast Group Rebuild Extension*. A node, which has a lesser hop count to the group leader and hence is closer to the group leader, responds to the RREQ with an RREP. The node waits for RREP_WAIT_TIME similar to what is described in the previous paragraph and activates the appropriate route. If the node does not receive any RREPs, it becomes a group leader, if it is a member of the multicast group. If the node is not a member of the multicast group, it sends a MACT with *G (group)* flag set to a downstream node. The downstream node, on the reception of a MACT with *G* flag set becomes a group leader if it is a member of the multicast group. If it is not a member of the multicast group, it sends the MACT with *G* flag set to its downstream node.

When a node leaves a multicast group, if it is not a leaf node, it continues to be on the multicast tree. If it is a leaf node, it sends a MACT with a *P (prune)* flag set to its multicast Next *Hop* and removes all the information for the multicast group from its *Multicast Route Table*. The next hop node on the reception of the MACT with a *P* flag set deletes the *Next Hop* entry from its own *Multicast Route Table*. If this node is itself not a member of the multicast group, and if the pruning of the other node has made it a leaf node, it can similarly prune itself from the tree. The tree branch pruning terminates when either a group member or a non-leaf node is reached.

When a multicast tree becomes disconnected due to a network partition, and if the partitions later reconnect, a node will receive multiple *Group Hellos* for the multicast group. In this scenario, the group leader with the lower IP address initiates the partition merge procedure. In section 4.4, we describe our additions to the partition merge procedure.

4. IMPLEMENTATION

4.1. TESTBED DESCRIPTION

There exist several different implementations of AODV. Therefore, we had to choose which implementation we should use as a base for adding the multicast extensions. We choose the AODV implementation from NIST, as it provided the necessary performance, flexibility, and stability. This implementation of AODV is written entirely as a kernel module. Our implementation of MAODV is based on NIST's base kernel AODV version 1.5[10]. As we used the NIST code as a starting point, our MAODV implementation is also a dynamically loadable kernel module that runs on Linux kernel version 2.4. The development was done on Red Hat Linux version 7.3. We used Linux kernel 2.4.18 for our implementation.

The development was done on a simple ad hoc network testbed consisting of Dell C640 laptops and IBM ThinkPad 600E laptops equipped with Orinoco Gold 802.11b wireless LAN cards. The wireless LAN cards operate in 2.4GHz spectrum, and in ad hoc mode provide a maximum capacity of 2Mb/s.

4.2. SOFTWARE ARCHITECTURE

Our implementation uses `netfilter` hooks to handle packets inside the kernel similar to the AODV implementations [10][11][12]. The MAODV module opens a UDP socket with port number 654. AODV and MAODV control messages are sent to and received from this socket. The module also initializes an IGMP socket for manipulating the multicast forwarding table that is provided by the Linux kernel. MAODV adds multicast routes to the kernel's multicast forwarding cache through the use of MRT_ADD_MFC socket option on the IGMP socket and deletes multicast routes through the use of MRT_DEL_MFC socket option on the IGMP socket.

When a node joins a multicast group, it starts sending IGMP_HOST_MEMBERSHIP_REPORT messages to the IGMP_ALL_ROUTERs group (224.0.0.2). We monitor these messages from a NF_IP_PRE_ROUTING `netfilter` hook. A `group_membership_table`, keeps track of all the multicast groups the node is a member of. The IGMP_HOST_MEMBERSHIP_REPORT is then dropped and an RREQ with *join* flag set is broadcasted. When the node leaves the multicast group, it sends the IGMP_HOST_LEAVE_MESSAGE to the IGMP_ALL_ROUTERs group (224.0.0.2). This message is also observed in the pre-routing `netfilter` hook. This message is then dropped and necessary action is taken as specified by the MAODV protocol [6]. During our implementation we also corrected a software bug in the IGMP code for Linux version 2.4, which was corrupting the IGMP_HOST_MEMEMBERSHIP_REPORT packets.

The kernel multicast route table entries in Linux are based on the *origin*, *destination* pair of the multicast packet. To keep track of the sources using the node as a multicast router, we added a `source_list` to the MAODV *Multicast Route Table*. A node adds itself as a default entry in the `source_list`. This is to make sure that the multicast packets originating from it are always forwarded. If the node is a valid router for a multicast group, in the pre-routing `netfilter` hook, it checks to see if the originator ID of the multicast packet that it has received for that group is in the `source_list`. If it is not already in the `source_list`, it adds the originator ID to the list.

When a node wishes to send packets to a multicast group, but does not intend to join the group, it sends RREQ packets to the multicast tree without the *join* flag set [6][13]. This is detected by monitoring packets in a NF_ IP_LOCAL_OUT output `netfilter` hook, and checking if a multicast data packet destined to a multicast group for which there is no *Multicast Route Table* entry is generated by the node. Any node (not necessarily a member of the multicast tree) that has a valid multicast route to the destination multicast tree responds with an RREP. The node waits for a time period of RREP_WAIT_TIME and sends a MACT to activate the route. We added a new flag `is_tree_member` to the MAODV *Multicast Route Table* to distinguish nodes that have a valid multicast route to the multicast tree, but are not part of the multicast tree (because they do not connect multicast group members). If a node that has its `is_tree_member` flag cleared doesn't receive a multicast packet destined to the multicast group for ACTIVE_ROUTE_TIMEOUT [6] amount of time, the *Multicast Route Table* entry for the group is deleted. Therefore, we note that, although, the routes to the multicast tree are soft state as in unicast AODV, the *Multicast Route Table* entries are hard state on nodes that are members of the multicast tree for the multicast group under consideration. . They are not removed unless a node is pruning itself from the multicast tree.

When a node that is a part of the multicast tree loses connectivity to its upstream *Next Hop* because of link breakage, it tries to rebuild the tree as mentioned in section 3. The node sets a flag called `rebuild_in_progress that` we have added to the *Multicast Route Table*. This flag makes sure that the node doesn't reply to RREQs from another node that is trying to join the multicast group. This is because the current node is in a transient state trying to rebuild the tree.

4.3. DATA PACKET CACHE

Part of our implementation also includes a *Data Packet Cache*. It is necessary for any multicast routing protocol implementation in a wireless network to implement a similar mechanism to discard duplicate packets. Consider the case illustrated in Figure 1. Nodes A, B and C are within, each other's communication range. The multicast tree is formed as shown by the solid lines. When node A forwards a multicast data packet, nodes B and C are able to listen since all the nodes are in communication range of each other. Since B and C are on the multicast tree, they too forward the multicast data packet. Node A will receive these transmissions from B and C and forward them again as they are data packets for the multicast group. This leads to a packet storm, where the nodes continuously forward duplicate packets till the TTL expires. The correct operation of multicast forwarding requires that Node A should not relay these duplicate transmissions. Therefore, there needs to be a mechanism to discard duplicate packets. This is typical of multicasting in wireless networks. Similar to the approaches in [6] and [8] we tackle this problem by using a *Data Packet Cache*. Our implementation caches the *source IP address* and *ID* fields of the IP headers of the multicast data packets it sees. We have implemented the *Data Packet Cache* as a hash table using the source IP address and the ID field to generate the hash value. Proper design of data packet buffer is crucial because of its necessity in eliminating duplicate packets. In the case illustrated in Figure 1, node A must cache information about a packet until it receives the packet back after nodes B and C forward the packet. The size of the data packet buffer should also not be too large as the entire cache is searched for each input data packet. If the data packet buffer is too large, it could potentially become a bottleneck on the data-forwarding path.

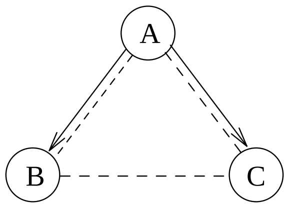

Figure 1: Data Packet Cache Scenario

4.4. MODIFICATIONS TO MAODV

During the implementation of MAODV, we realized that certain modifications to the protocol would enhance the performance of the protocol. In this section, we describe these changes.

Tree Optimization using Group Hello Messages

The group leader for a multicast group generates the *Group Hello* messages periodically once every GROUP_HELLO_INTERVAL milliseconds. These messages are propagated through out the connected portion of the network. These messages could be used to proactively maintain routes to the group leader. We added an additional field called *Group Leader Sequence Number* to the Group *Hello* message to enable this. A node will update its route to the group leader if the received *Group Hello indicates* a higher *Group Leader Sequence Number* than its record of group leader's sequence number indicating a fresher route; or if the *Group Leader Sequence Number* is equal to its record of the group leader sequence number and if the *Group Hello* is received with a lesser *Hop Count* field indicating a shorter path to the group leader. On updating the route, the node makes the node from which it received the *Group Hello* as its next hop in the route table entry to the group leader.

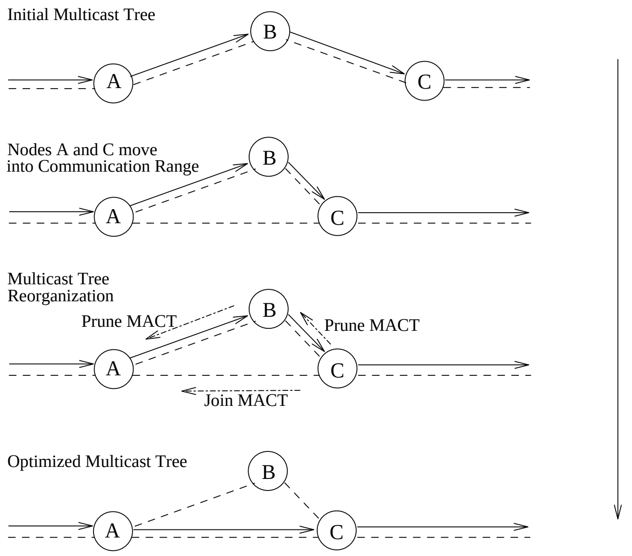

Figure 2: Multicast Tree Optimization

Consider the case illustrated in the first portion of Figure 2. The dashed line between any two nodes indicates that they are within communication range of each other, and a solid line arrow indicates that the node participate in a multicast tree, either as group members or as forwarding nodes. Nodes A and B are in communication range of each other. Nodes B and C are in communication range of each other. But, nodes A and C are out of communication range of each other. Nodes A, B and C are part of a multicast tree as indicated by the solid arrow lines. Nodes A and C are members of the multicast group and node B is a forwarding node. Node A is an upstream next hop of B, and C is a downstream next hop of B. Now, if node C moves into the communication range of A the multicast tree becomes inefficient. Packets for the multicast group from node A will continue to reach C via B, even though, A and C are in direct communication range of each other. We propose a modification to the MAODV base protocol that can accommodate this scenario, and result in a more efficient multicast tree. Our solution is as follows: When nodes A and C move in the communication range of each other, C will be able to hear *Group Hello* messages being broadcast by A. The *Group Hello* messages would have the *O* flag set to 0 indicating that the *Group Hello* has traveled along the tree. The *Hop Count* field in the *Group Hello* message from node A, indicates that if C joined the tree as a downstream neighbor of A, it would be closer to the group leader and the tree would be optimized. Therefore, node C sends a MACT with *J* flag set (Join MACT) to node A grafting the link A-C onto the multicast tree. At the same time, node C will send a MACT with the P flag (Prune MACT) set to node B. Since node B is only a multicast forwarder, he would prune himself from the multicast tree as he would no longer have any downstream nodes for the multicast tree. In this way links A-B and B-C will be pruned from the multicast tree. Figure 2 shows this optimization. Using this method, even though the multicast tree may be formed non-optimally during its construction, it will eventually become optimized because of the procedure mentioned above.

Partition Merge

We also wish to propose a second enhancement to the MAODV protocol, which can significantly improve the stability of the protocol during partition merge. We propose that an additional `merge_in_progress` flag be added to the *Multicast Route Table*. In the event that a multicast tree gets partitioned due to network mobility, we can have multiple instances of the multicast tree, each with its own group leader. This can create problems if the network partitions subsequently reconnect. For simplicity we consider the example, where the network gets partitioned into two. In this scenario, there would be two group leaders (say GL_1 and GL_2 : $GL_1 < GL_2$) sending *Group Hello* messages in the network. Let G_1 be the portion of the tree whose group leader is GL_1 and G_2 be the portion of the tree whose group leader is GL_2. When a member of G1 hears a *Group Hello* from GL_2, it sends a unicast RREQ with the *R* flag set to GL_1, indicating that two portions of the multicast tree have reconnected. The node also sets a soft state `merge_in_progress` flag to 1 indicating that the *partition merge* procedure has started. This node will then drop all subsequent *Group Hello* messages that reach it. The node also updates its unicast route to GL_2. We have added the fields GL_1, GL_2 and GL *sequence* to a RREQ, which has the R flag set. The GL sequence is set to the sequence number of GL_2. Also, the *Hop Count* field in the RREQ indicates hop count to GL_2. All nodes that receive this RREQ update their route to GL_2 by looking at the GL *sequence* and *Hop Count* fields in the RREQ. They also set the `merge_in_progress` flag in their *Multicast Route Tables*. If the `merge_in_progress` flag is set, it means that the partition merge procedure is in execution and they need not react if they receive any more Group *Hellos* from GL_2. When GL_1 receives the RREQ with the *R* and *J* flags set, it unicasts an RREQ with *R* and *J* flags set to GL_2. It also sets its `merge_in_progress flag` to 1. While this flag is set, it disregards any further RREQs with *R* flag set that any other nodes may send. The GL sequence field of the RREQ is set to the current destination sequence number of GL_1. On reception of the RREQ, GL_2 unicasts a RREP back to GL_1, merging the two trees. GL_2 also sends a *Group Hello* with U flag set so that all the nodes in G_2 update their group leader information. Because of the unreliable nature of the wireless channel, any of the merge RREQs or RREPs can get lost. If this happens, the partition merge procedure may never complete. Therefore, we set the `merge_in_progress` flag to expire in PARTITION_MERGE_TIMER milliseconds. After this flag expires, a node that is a member of G_1 reinitiates the merge procedure on the reception of a GRPH from GL_2. If the value for the PARTITION_MERGE_TIMER is too small, members of G1 will keep sending RREQs with R flag set waiting long enough for the tree merge to complete. However, the PARTITION_MERGE_TIMER should not be too long either, otherwise even after partition merge; packets will continue not to be delivered to some group members. After the two portions of the multicast tree have merged, the tree will most likely be non-optimal. However, as a result of the *tree optimization* procedure using *Group Hellos* mentioned in the earlier part of the section, the tree will gradually graft and prune different links to become more efficient.

Reliability of MACT Messages

During the experiments we ran, we observed that control packets might get lost, as the wireless channel is unreliable. If only, RREQ or RREP packets are lost, the multicast tree may take a longer time to form or it may form non-optimally. This is not crucial, as eventually the multicast tree will optimize itself. However, if MACT messages are lost, the tree may never form or may cause the nodes to have incorrect multicast *Next Hop* information. For example, if a MACT with a *join* flag is lost, the branch that the MACT is trying to graft will never be added onto the multicast tree. If a MACT with a *prune* flag is lost, a node that is on the multicast tree may never realize that its *Next Hop,* which was a leaf node, has pruned away and might continue to forward multicast data packets, causing a waste of bandwidth. If a MACT with *Group Leader* flag were lost, a node, which is supposed to become a group leader, would never realize that it is supposed to become a group leader. Therefore, MACT messages with *join*, prune and *Group Leader* flags need to be reliably delivered. We have added an additional control message called MACT_ACK. After a node sends a MACT message to its neighbor, it waits for a MACT_ACK message from its neighbor. If it does not receive a MACT ACK within a time of MACT_ACK_TIMER, it resends the MACT message. The node can send a maximum of MACT_RETRIES number of times. The lack of reliability in these crucial tree maintenance messages has been a critical oversight in the initial design of the protocol. Adding reliability to these messages ensures that the multicast tree is formed reliably.

5. VALIDATION EXPERIMENTS

An integral part of implementing a routing protocol is verifying its correct operation in various scenarios. We performed validation experiments to test different features of MAODV in different scenarios. We used the `iptables` functionality in Linux to emulate different topology scenarios. Our implementation was tested in increasingly complex scenarios, and this provides us some degree of reassurance and confidence in its correct operation. We used a combination of widely used multicast application such as `vic` as well as simple custom written test programs to verify the operation of MAODV. Access to the Multicast Routing Table is provided to user space via the `proc` file system in Linux.

5.1. TWO AND THREE NODE SCENARIOS

Even a simple two-node scenario can verify a significant amount of MAODV operation such as correct operation of Group Leader, RREQ, RREP, MACT, interaction with the Linux kernel multicast forwarding table, the proc file system, as well as multicast data forwarding. However a three-node test topology allows us to test the operation of the protocol in several key scenarios. Using `iptables` with only three nodes we can emulate different connectivity scenarios as well as basic mobility of the nodes. These different scenarios are illustrated in Figure 2. The dashed lines indicate connectivity between nodes, and the solid arrow lines indicate the multicast tree.

Three nodes A, B and C connected as shown in the first portion of Figure 2. Nodes A and B can see each other. Similarly, nodes B and C can see each other. But nodes A and C cannot see each other. In this scenario, an application on node A joins a multicast group. It sends RREQ_RETRIES number of RREQs for the multicast tree and not having received any RREPs, it becomes a group leader. Next, node C joins the multicast group. It sends out RREQs. The RREQ is forwarded to node A via node B. Node A, which is a member of the multicast tree, responds to the RREQ with an RREP. Node C on receiving the RREP waits for RREP_WAIT_TIME and then sends a MACT towards node B, and activates the *Next Hop* B. Node B sends a MACT_ACK to node C and activates its *Next Hop* entries. It then sends a MACT to node A. Node A responds with a MACT_ACK to node B and activates the *Next Hop B.* The multicast tree is formed as indicated by the solid lines in the first portion of Figure 2. This straight-line topology verifies the correct operation of MAODV, in a scenario where an intermediate node is not a member of the group, but simply a multicast-forwarding node.

Next, we remove the `iptables` packet filtering from both node A and C and allow node A to see packets originating from MAC address C and vice versa. This creates a topology where each node is in the communication range of the other two as indicated by the dashed lines in the second scenario in Figure 2. Node C is now able to hear the *Group Hello* messages being forwarded by node A. These have a lesser *Hop Count* than the previous Group Hello messages received from node B. Node C now sends a MACT with a *join* flag to node A and a MACT with *prune* flag set to node B as mentioned in the section 4.4. Node B and node A send a MACT_ACK back to node C. Also, as node B no longer has any downstream next hop nodes in the multicast tree it prunes itself from the multicast tree by sending a MACT with *prune* flag set to node A. Node A sends a MACT_ACK to node B, completing the prune. The multicast tree now looks as indicated by the solid lines in the last scenario in Figure 2.

Re-enabling the iptables packet filtering to emulate the scenario where nodes A and C cannot communicate directly, we can verify that the multicast tree once again forms correctly. This verifies that node C correctly generates a multicast RREQ with the *Group Rebuild* extensions [6] as it has lost contact with its upstream Next Hop node A.

5.2. PARTITION MERGE

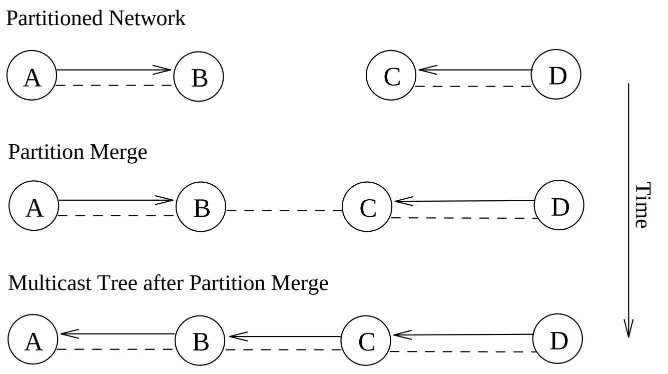

Figure 3: Partition Merge

It is important to verify the correct operation of the MAODV protocol when a partition merge occurs. We emulate the topology as indicated in the initial scenario in Figure 3 using `iptables` as described in previous sections. We create a partitioned network, such that nodes A and B are in one partition and nodes C and D are in a different partition. All nodes join the same multicast group, but due to the partition, two separate multicast trees are formed. Node A is the group leader for one tree and node D is the group leader for the other tree. The IP address of node A is numerically lesser than node D. Next, we use `iptables` to change the connectivity between nodes such that nodes B and C can now communicate with each other (two partitions coming together). Node B, can now hears *Group Hellos* being sent by node D. Node B sends an RREQ with *Partition Merge* extensions and the R flag set to node A, which is its current group leader for. Node A on receiving this then sends a unicast RREQ with *Partition Merge* extensions and R and J flags set to node D which is the group leader of the second partition. Node D now generates an RREP with R flag set. The two partitions of the multicast tree are now merged. This process is illustrated in Figure 3.

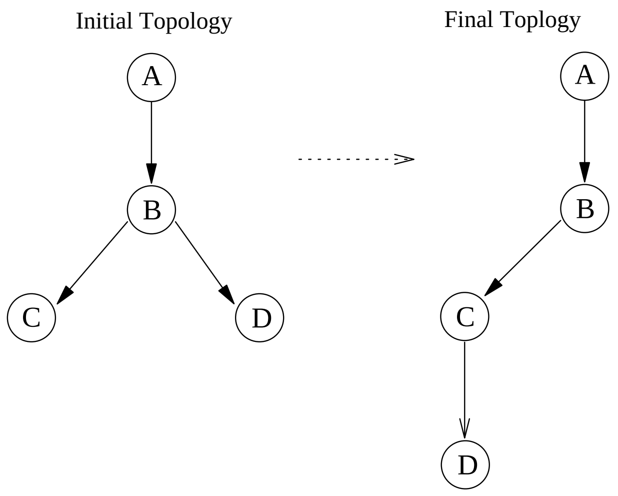

Figure 4: Control Overhead Experiment

6. EXPERIMENTAL RESULTS

In addition to verifying the basic operation of the MAODV protocol, we also attempted to gain some insight into its efficiency in terms of packet forwarding. We wanted to measure not only the control packet overhead, that MAODV generated, but also wanted to quantify at what rate could our implementation forward multicast traffic. We wanted to take these measurements for the topology outlined in Figure 4, with the nodes spatially separated by sufficient distance to generate the topology. However, we ran into the problem outlined in [14]. The problem is that *Hello* messages do not accurately indicate connectivity for forwarding data packets. *Hello* packets are broadcast packets, which are transmitted with maximum power. So it is possible for nodes to see *Hello* packets from other nodes and think that there exists connectivity, between them, however, when data packets are sent they do not reach the destination nodes. Therefore, we emulate the desired topology with `iptables` as mentioned in the previous sections. The problem with this approach is that the wireless channel bandwidth is now shared. This may have a significant impact in any throughput measurements.

We wanted to characterize the control packet overhead generated by MAODV. However, this is highly dependent on the mobility scenario, and very difficult to generalize. Therefore, we decided to measure this overhead for a small representative example. We start the experiment with the nodes emulating the initial topology shown in Figure 4. Node A is chosen as the multicast source and nodes C and D were chosen as multicast receivers. Node A sends packets of size 1000 bytes. Node B acts as a forwarding node that is not a member of the multicast group. During the course of the experiment, we change the topology of the nodes to emulate basic mobility. The new topology emulates the motion of node D from the vicinity of node B to the vicinity of node C. We measured the number of MAODV control packets received at every node and the number of multicast data packets that were generated by node A. Node A sends data for 370 sec. And nodes A, B and C remain on the multicast tree for 540 sec. The average data transmission rate was measured to be 242Kbps, and the control overhead was 2.75 pkts/sec.

7. CONCLUSIONS

In this paper, we presented our experiences in implementing multicast AODV in the kernel space. We tested and validated the protocol with the implementation. We have also presented some modifications that can enhance the performance of MAODV. We propose the use of a mechanism to optimize the multicast tree as often the tree may not be the most efficient. We propose one such mechanism based on Group Hello messages. We also propose a scheme to combine two disjoint instances of a multicast tree after a partition merge has occurred. In addition we believe that it is essential that MACT messages of MAODV have some mechanism to guard against corruption or loss. This can have a serious impact on the formation and the maintenance of the multicast tree.

We are currently investigating modifications to a users space implementation of AODV. We would like to compare the performance of both our kernel level implementation as well as a user space implementation that includes some of our proposed enhancements. We are also testing our implementation on strongARM processor based devices such as the IPAQ and the cerfCube. Our implementations will be made publicly available.

ACKNOWLEDGEMENTS

The authors would like to thank Karthikeyan Chandrashekar for helpful discussions on optimizing the multicast tree; Dionysus Blaze, Dan Rusk and Jonathan Polley for helping fix bugs.

REFERENCES

[1] E. Royer and C. Perkins, “Multicast Operation of the Ad-Hoc On-demand Distance Vector Routing Protocol,” *Proceedings of Mobicom’99, Seattle, WA*, pp. 207–218, August 1999.

[2] E. Bommaiah, M. Liu, McAuley A. and R. Talpade “AMROUTE: Ad Hoc Multicast Routing Protocol,” *IETF Internet Draft, draft-talpade-manet-amroute-00.txt (Work in Progress)*, Aug 1998.

[3] L. Ji and M. Corson, “A Lightweight Adaptive Multicast Algorithm,” *Proceedings of IEEE GLOBECOM*, pp. 1036–1042, Dec 1998.

[4] M. Gerla, C.C. Chiang, S.J. Lee, “On-demand Multicast Routing Protocol,” *Proceedings of IEEE WCNC ’99, New Orleans, LA*, pp. 1298–1302, Sep 1999.

[5] V. Park and M. Corson, “A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks,” *Proceedings of IEEE Conference on Computer Communications ’97, Kobe, Japan.*, pp. 1405–1413, 1997.

[6] C. Perkins and E. Royer-Belding, “Multicast Ad Hoc On-demand Distance Vector (MAODV) Routing,” *IETF Internet Draft, draft-ietf-manet-maodv-00.txt (Work in Progress)*, Jul 2000.

[7] E. Royer and C. Perkins, “Ad Hoc On-demand Distance Vector Routing Protocol,” *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, New Orleans, LA*, pp. 90–100, Feb 1999.

[8] T. Kunz and E. Cheng, “Multicasting in Ad-Hoc Networks: Comparing MAODV and ODMRP,” *Proceedings of the Workshop on Adhoc Communications, Bonn, Germany*, Sep 2001.

[9] F. Lilieblad, O. Mattsson, P. Nylund, D. Ouchterlony, and A. Roxenhag, “Personal Communication” *http://fl.ssvl.kth.se/~g4/madhoc/docs/techdoc.ps.*

[10] K.B. Luke., “NIST Kernel AODV Implementation.” *http://w3.antd.nist.gov/wctg/aodv_kernel/index.html*

[11] E. Nordstrom “AODV-UU Implementation.” *http://user.it.uu.se/~henrikl/aodv/*

[12] E. Royer and C.Perkins, “AODV-UCSB Implementation.” *http://moment.cs.ucsb.edu/AODV/aodv.html*

[13] E. Royer and C. Perkins, “An Implementation study of the AODV Routing Protocol,” *Proceedings of the IEEE Wireless Communications and Networking Conference, Chicago, IL*, Sep 2000.

[14] H. Lundgren. and E. Nordstrom, “Coping with Communication Gray Zones in IEEE 802.11b based Ad Hoc Networks WOWMOM 2002,” *International Workshop on Wireless Mobile Multimedia*, Sep 2002