An Implementation Study of the AODV Routing Protocol

Elizabeth M. Royer
Dept. of Electrical & Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106
eroyer@alpha.ece.ucsb.edu

Charles E. Perkins
Communications System Laboratory
Nokia Research Center
Mountain View, CA 94043
cperkins@iprg.nokia.com

Abstract* – The Ad hoc On-Demand Distance Vector (AODV) routing protocol is designed for use in ad hoc mobile networks. Because of the difficulty of testing an ad hoc routing protocol in a real-world environment, a simulation was first created so that the protocol design could be tested in a variety of scenarios. Once simulation of the protocol was nearly complete, the simulation was used as the basis for an implementation in the Linux operating system. In the course of converting the simulation into an implementation, certain modifications were needed in AODV and the Linux kernel due to both simplifications made in the simulation of AODV and to incompatibilities of the Linux kernel and the IP-layer to routing in a mobile environment. This paper details many of the changes that were necessary during the development of the implementation.

I. INTRODUCTION AND MOTIVATION

Mobile wireless devices are rapidly gaining popularity due to recent improvements in the portability and power of these products. There is a growing need for communication protocols which allow users of these devices to communicate over wireless links. To allow such on-the-fly formation of networks, the Ad hoc On-Demand Distance Vector (AODV) routing protocol has been developed [5], [6], [7]. AODV has been designed for use in ad hoc mobile networks. It allows users to find and maintain routes to other users in the network whenever such routes are needed.

Testing mobile wireless protocols in a real-world environment presents numerous difficulties. These difficulties include creating repeatable scenarios with tens, hundreds, or even thousands of mobile nodes. Creating multiple scenarios with only small variances is also quite challenging. Because of these difficulties, simulations of AODV have been created to test the protocol in a variety of repeatable scenarios [5], [7]. However, while simulating a protocol can aid in the basic design and testing of the protocol, certain assumptions and simplifications can be made in a simulation that are not valid in a real-world scenario. Hence, it is important to implement the protocol, once the simulation is complete.

This paper describes recent work in the development of an implementation of the AODV protocol. While the implementation is still in the process of completion, certain changes have already been necessary to both the protocol and the kernel in order to allow AODV to operate correctly. Many of these

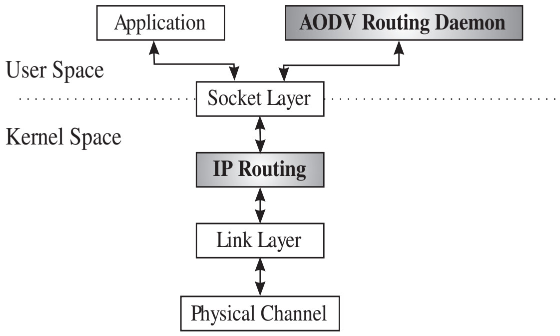
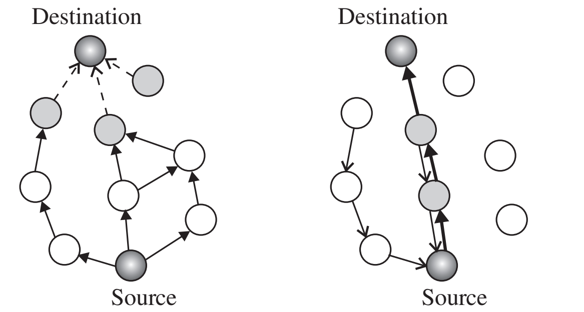


Fig. 1. Implementation Structure.

changes are notable because they have resulted in modifications which have been incorporated into the AODV Internet Draft [6]. Others are useful information for researchers endeavoring their own protocol implementations.

To minimize changes to the kernel, AODV was implemented as a daemon in user space. Other implementations [1], [3] have selected this approach as well. Figure 1 illustrates the logical structure of the implementation, highlighting where the modifications occurred. We chose to implement AODV in the Linux kernel because of the inherent mobility and open-source characteristics of Linux. The alternative to implementing a routing protocol in user space is to incorporate the protocol into the existing kernel, as in [4]. Incorporating the protocol into the kernel has the advantages that it will operate faster, and a mechanism is not required for transportation between kernel and user space. Kernel implementations naturally have easier access to all header data at the network (IP) layer and below, making it much easier to utilize header fields and options when making protocol decisions. For example, this would simplify the use of source routes in protocol operations. However, this method also has the negative effects of making the protocol less portable, making it more difficult to maintain, reducing the protocol functionality, and impairing memory management.

The remainder of this paper is organized as follows. Section II presents an overview of AODV's unicast and multicast operation. Section III-A details some of the modifications that



(a) RREQ Broadcast (b) RREP Propagation and Subsequent Route

Fig. 2. Route Discovery Cycle.

were necessary to the AODV protocol, while Section III-B describes modifications needed in the Linux kernel. Plans for future work in the AODV implementation are discussed in Section IV. Finally, Section V concludes the paper.

II. OVERVIEW OF AODV

The Ad hoc On-Demand Distance Vector (AODV) routing protocol provides unicast, broadcast, and multicast communication in ad hoc mobile networks [5], [6], [7]. AODV initiates route discovery whenever a route is needed by a source node, or whenever a node wishes to join a multicast group. Routes are maintained as long as they are needed by the source node or as long as the multicast group exists, and the routes are always loop-free through the use of sequence numbers. AODV nodes maintain a route table in which next hop routing information for destination nodes is stored.

A. Unicast

Route discovery in AODV follows a route request/route reply query cycle. A source node in need of a route broadcasts a *Route Request* (RREQ) packet (Fig. 2(a)) across the network. Any node with a current route to the destination, including the destination itself, can respond to the RREQ by unicasting a *Route Reply* (RREP) to the source node. Once the source node receives the RREP, it can begin sending data packets along this route to the destination. Fig. 2(b) illustrates the propagation of RREP messages back to the source node, and the subsequent route selected by the source node to the destination.

Because nodes are moving, link breaks are likely to occur. When a link break in an *active* route occurs, the node upstream of the break broadcasts a *Route Error* (RERR) message containing a list of all the destinations which are now unreachable due to the loss of the link. The RERR is propagated back to the source node. Once the source node receives this message, it may reinitiate route discovery if it still needs the route.

B. Multicast

As network nodes join a multicast group, a bi-directional shared tree composed of the multicast group members and intermediary nodes needed to connect the group members is created. Each multicast group has associated with it a group leader. The primary function of the group leader is to maintain and disseminate the multicast group sequence number. This sequence number is used in maintaining freshness of routing information for the multicast group.

A node wishing to join a multicast group broadcasts a RREQ message with the *join* flag set. Any node which is a member of the multicast tree can respond to this message by unicasting a RREP back to the originator of the RREQ. The node joining the group waits a discovery period, during which it collects RREPs. Any node forwarding a RREP must also keep track of its best route to the multicast tree. At the end of the discovery period, the requesting node selects the shortest route to the multicast tree, and unicasts the next hop along this route a *Multicast Activation* (MACT) message. When the next hop receives this message, it *activates* the link in its multicast route table. If this node is already a tree member, the new tree branch is finalized. Otherwise, the next hop in turn unicasts a MACT message to the neighbor it has selected as its next hop towards the multicast group tree. Processing continues in this manner until a node that was already a member of the multicast tree is reached and the addition of the tree branch is complete. Further details about AODV multicast can be found in [7].

III. MODIFICATIONS

AODV has been implemented as a routing daemon in user space. The daemon communicates with the Linux kernel through the use of sockets. At initialization, AODV opens a UDP socket to the kernel. This socket is used for both the transmission and reception of AODV control messages. AODV has been issued port number 654, and hence binds to this port when opening the socket. Other channels of communication include an IGMP socket for multicast, and a `netlink` socket for routing table updates. These sockets are described further in the following sections.

During the development of the AODV implementation, many situations arose which required modifications to either the AODV protocol or the Linux kernel. Some of the changes were necessary due to simplifications made when simulating AODV, while others were needed because of incompatibilities between AODV and the Linux kernel. The following sections detail many of the modifications that were needed in the course of the implementation development.

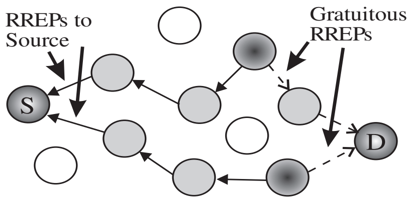


Fig. 3. Gratuitous RREPs.

A. Protocol Modifications

A.1 The Route Reply

One of the most basic changes made to AODV was the way in which RREP packets are forwarded. When a node receives a RREQ, it replies if it either is the destination, or if it has a current route to the destination. In the simulation, RREPs were originally unicast from the responding node to the source. As the RREP was propagated, intermediate nodes updated their route tables to include a route to the destination. In the implementation, however, this does not work, because if the RREP is unicast from the responding node to the source, the intermediate nodes use IP forwarding and do not process the packet. Hence the protocol needed to be changed so that RREPs are unicast on a hop-by-hop basis. Additionally, a Source IP Address field was added to the RREP so that the ultimate destination of the RREP would be retained.

Another modification was made due to a discovery resulting from an implementation by Dan Ouchterlony et. al. [3]. When an intermediate node sends a reply to the source, the destination does not learn of that route to the source node because it has not received the route request. If all the route requests are answered by intermediate nodes and consequently the destination never receives a copy of the RREQ, it will never learn of a route to the source node. This can be detrimental if the source node wishes to establish a TCP connection. In order that the destination learns of routes to the source, a modification to AODV was made so that intermediate nodes responding to RREQs by unicasting a RREP to the source node also send a *gratuitous RREP* to the destination informing it of the new route. The intermediate node places the destination IP address in the Source IP Address field of the RREP, and places the source IP address in the Destination IP Address field, since this is the node for which the route is offered. The intermediate node enters its distance from the source in the HopCount field, and then unicasts this RREP to the destination. When the destination receives the RREP, it will know of a route to the source. Figure 3 illustrates an example of this process.

In order to reduce the additional overhead incurred by the gratuitous RREP, we have added a *Grat RREP* flag to the RREQ. The source sets this flag if the session is going to be run over TCP, or if the destination should receive the gratuitous RREP for any other reason. Otherwise, it leaves the flag unset. Nodes receiving the RREQ and responding with a RREP only transmit the gratuitous RREP to the destination if this flag is set.

A.2 The DELETE_PERIOD

When a node reboots, it loses all of its routing information, including the value of its sequence number. In the formal verification of AODV done by Bhargavan et. al. [2], it was found that routing loops could result after a node reboots because neighboring nodes may still have the rebooted node as a next hop for one or more destinations. If the rebooted node attempts discovery for one of those destinations, the neighboring nodes may reply, and hence a route will be created. To prevent this situation, each node waits for DELETE_PERIOD after rebooting before it responds to any routing messages. If it receives a data packet during this time, it sends a RERR for that destination, since it no longer has the route. Additionally, it resets the waiting timer to the current time plus DELETE_PERIOD to ensure that all routes with it as a next hop expire.

A.3 Route Tables

The AODV routing daemon communicates changes to the IP route table through the use of a `netlink` socket. Whenever AODV has a route addition, modification, or deletion, it transmits a message to IP through this socket and the route is updated accordingly.

To prevent the premature deletion of routes in the kernel routing table, AODV's route table maintenance was altered to include a periodic refresh of the kernel route table entries. This was easily accomplished through the use of a periodic timer. When the timer expires, AODV sends a message to IP on the `netlink` socket telling it to update, or refresh, the route.

A.4 Multicast

When an application wishes to join a multicast group, it sends an IGMP membership report containing the address of group. Because of the goal to minimize changes to the kernel, we wanted to leave this functionality intact. Hence, AODV was modified so that it initializes an IGMP socket and then listens to the messages transmitted on this socket. An application wishing to join a multicast group transmits an IGMP_MEMBERSHIP_REPORT on the IGMP socket in order to request a nearby router to join the group. When AODV detects a membership report from its host listing a new multicast group, it initiates route discovery for that group. Similarly, an application requesting to leave a multicast group transmits an IGMP_LEAVE_GROUP message. When AODV detects this message transmitted by its host on the socket, it initiates the process whereby it leaves the multicast group.

The IGMP socket is also used for multicast route table updates. When a node either becomes a member of the multicast tree or learns of a route to that tree, AODV updates the kernel's

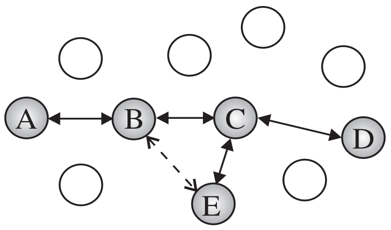


Fig. 4. Example Multicast Tree.

multicast route table through the use of the MRT_ADD_MFC socket option. AODV only issues this command once a MACT message activating the route has been received. Likewise, when a node leaves the multicast tree or when its route to the tree expires, a MRT_DEL_MFC command is issued on the IGMP socket to invalidate this multicast table entry.

Multicast route table entries in the kernel must be periodically refreshed; otherwise, they expire and are deleted. AODV maintains a periodic timer for refreshing multicast route table entries. This refresh is also accomplished through the use of the IGMP socket.

As nodes join a multicast group, a bi-directional shared tree is created which connects the multicast group members. When a node on the multicast tree receives a data packet destined for the multicast group, it processes the packet if it is a multicast group member. It then rebroadcasts the packet, regardless of whether it is a group member or just a router on the tree. The destination address of the data packet is the multicast group address. In the simulator version of AODV, when a node receives a multicast data packet, it only processes the packet if the packet arrives from one of its next hops. In the course of the implementation development, however, we realized a situation such as the one depicted in Figure 4 can arise. In the figure, the solid arrows indicate multicast tree links. Node E is connected to the multicast tree through node C. However, suppose node E is also within the transmission radius of node B, another node on the tree. When node B transmits multicast data packets, both nodes C and E receive them. We have modified the AODV protocol to enable multicast tree nodes to process multicast data packets if they are a member of the multicast group, regardless of from whom they receive the packet. Hence, node E processes data packets it receives from node B or C. In this way, the multicast tree is used for ensuring that multicast group members are connected, but this increase in redundancy leads to a greater number of multicast data packets successfully delivered to the multicast group members. Looping is prevented through the use of the IP *Ident* header field, as described in Section III-B.

AODV had originally been designed so that there was a difference in operation between a node wanting to join a multicast group, and a node needing to find a route to the multicast group for the transmission of data packets to that group. In the former scenario, a node joining the group waits the full route discovery period after broadcasting a RREQ to receive RREPs from tree members. At the end of this discovery period, the node unicasts a MACT message to its selected next hop, and the branch is added on to the tree. On the other hand, when a node only needs a route to the tree but does not actually want to join the group, it broadcasts a RREQ and waits for reception of a RREP. After it receives the first RREP, it begins using that route (as in unicast AODV). If it later receives a better route, it then updates its routing table entry.

While implementing the protocol, we realized that this approach will not work for the case when the node is just finding a route to the group. Because multicast traffic is by nature broadcast locally, neighboring nodes which forward a RREP to the source node will have no way of knowing whether they are selected as next hops for forwarding the multicast data to the multicast tree. Hence, when the source broadcasts the data, multiple neighboring nodes could potentially rebroadcast the packet, resulting in an inefficient use of the available bandwidth. We thus modified AODV so that when a node is discovering a route to the multicast group, it must wait the full discovery period, regardless of whether or not it wishes to join the group. At the end of this discovery period, the node unicasts a MACT message to its selected next hop. A *join* flag was added to the MACT message so that the node can indicate whether it is actually joining the multicast tree. Only nodes that receive this MACT message may rebroadcast multicast data packets. This enables unnecessary broadcasts to be suppressed, and bandwidth is more efficiently utilized.

A.5 Interfaces

Multi-homed devices were not originally taken into consideration in the design of the protocol. It was implicitly assumed that AODV would operate over single interface radios. However, because AODV should also operate smoothly over wired networks, and because it is likely that AODV will also be used with multi-homed radios, the consideration of interfaces needed to be incorporated into the protocol design.

When a node receives a RREQ, it needs to know upon which interface the packet arrived so that it will later know which interface to use to reach the source. Similarly, when a node receives a RREP, the interface upon which the RREP arrived is needed so that the destination can later be reached. The same holds true for RERR and MACT messages. If a node has multiple interfaces, it is not necessary that a RERR be broadcast out of each of its interfaces; the RERR should only be transmitted on those interfaces that have a neighbor which uses the route. To enable AODV to maintain an association between destinations and outgoing interfaces, an `interface` field was added to routing table entries.

B. Kernel Modifications

B.1 IP Routing

When a packet arrives at a node's IP-layer from the application layer, IP checks whether it has a route to the destination by consulting its routing table. If it has either a route or a default router, it forwards the packet. If neither of these exists, IP informs the application that a route does not exist, and the session is aborted. In ad hoc routing, default routes typically do not exist, except possibly for specific connections to an established infrastructure. Often, due to node mobility, and especially with on-demand protocols, a valid route is not known for a given destination. Instead of notifying the application, IP must be changed to notify the routing daemon that a route needs to be found for the destination. Linux's `netlink` socket mechanism can be utilized so that IP sends a message to the AODV daemon, informing it to initiate route discovery for the destination. This prevents the session from aborting every time a route is not known.

We have designed a mechanism through which IP may notify AODV of the need to initiate route discovery. Through the creation of a new `netlink` socket type, NETLINK_AODV, AODV and IP can establish a channel through which these notifications are sent. The socket is initialized by AODV, and the only traffic sent on this socket are route discovery notifications from IP to the AODV daemon. IP informs AODV of the destination IP address and port number of the application which has initiated the session. In the event that AODV is unable to find a route to this destination, it can use the port number to notify the application that the session should be aborted.

To enable the operation of multicast and broadcast and to prevent routing loops in these communication forms, the *Ident* IP header field has been adopted for use as a sequence number for data packets. Sources of multicast or broadcast data packets maintain their own monotonically increasing sequence number for the *Ident* field. Each time a source initiates a new multicast or broadcast packet, it increments the value of this counter and records this value in the *Ident* header field. When a node first receives one of these data packets, it temporarily buffers the *Ident* field/Source IP Address combination. If it receives another packet with this same combination, it silently discards the packet and does not process it further.

B.2 Route Tables

AODV maintains its own route table of destinations for which it has routes. Each route table entry has associated with it a Lifetime field. When an entry's lifetime expires, that entry is invalidated. Each time a route to a destination is used, the lifetime associated with that route is updated so that the route table entry is not prematurely deleted. Because IP is responsible for forwarding data packets, however, AODV does not know when route entries are used. Hence it can not accurately use this feature within the routing daemon. To enable this functionality of AODV to be maintained, the kernel can be modified so that IP maintains a structure parallel to the IP Route table in which it stores a Last Use field.

Each time a route table entry is used to transmit a packet, IP updates the last use to be the current time. When a route table entry in AODV expires, before deleting the route AODV queries IP for the last use value of that routing table entry. If the route has been used within the previous lifetime period, the route should not be deleted. Instead, a new route timer is set for the time remaining until the expiration of the route table entry.

B.3 Data Packet Buffering

When an application wants to send a data packet to a destination but a route is not available, AODV initiates a route discovery operation. In the meantime, the packet has to be either dropped, or else stored for further processing. While IP is known to be a *best-effort* protocol, and higher layer protocols are often designed to recover from packet loss, it is still a good idea to avoid systemic problems that lead to almost certain losses.

In existing IP implementations, there are no facilities for saving these packets, so we have designed a new method for avoiding packet loss during route acquisition. This works as follows:

- IP determines that no route is available for delivery of the packet.
- IP signals AODV on the `netlink` socket that a route is required.
- IP builds a dummy route table entry for the expected route, with a relatively short expiration time.
- IP queues the packet in a simple linked list referenced from the dummy route table entry.
- If no route is found, the dummy route table entry expires.
- Otherwise, when AODV installs the route, the dummy route table entry is updated to point to the actual next hop.
- When the route table entry is updated with the valid route information, the queued packets are delivered.

The dummy route table entry referred to above can be an entry with either an invalid next hop field or a specific "invalid" flag. If IP receives another data packet before the route table for the intended destination has been updated with valid path information, then the new data packet can be stored in the queue to await future delivery.

With this strategy, there may be time-critical packets that are delivered late instead of dropped. We have not yet determined a useful solution to this problem, but typically the receiving end will discard stale data anyway according to some application-level sequence numbering scheme.

IV. FUTURE WORK

We intend to finish the Linux AODV implementation and make it available at the AODV web site (http://alpha.ece.ucsb.edu/~eroyer/aodv.html). We also want to make our `netlink` modifications fit better with the intended overall (inferred) operation of the netlink facility in Linux.

We also would like to apply the lessons we have learned with Linux to produce a FreeBSD implementation. Since `netlink` is not available with FreeBSD, we are likely to design a special device driver interface (/dev/aodv) with new ioctl()s replacing the Linux `netlink` functions. The basic control flow, however, should otherwise remain identical. It would be nice to be able to produce a Windows implementation, but so far we are hampered by the unavailability of source code or documented interfaces into the main Windows protocol stack.

We also intend to upgrade our simulations with the new features motivated by the implementation lessons described in this paper. Then, it should be possible to run the full implementation on networks containing a large number of simulated ad hoc nodes along with the nodes running the actual AODV implementation. This could give us valuable information towards selecting realistic and optimal values for the protocol parameters needed for any implementation of AODV.

The method by which packets are stored during route discovery needs further improvements. We might find that it is possible to store pending packets in the memory of the AODV daemon. We might be able to devise application profiles that help to determine how long a packet should be stored before it goes stale and has to be discarded. Local feedback to the application is also possible in conjunction with such discarding operations. We may also determine that almost identical buffering operations are needed during any *local repair* operations that may be carried out by intermediate routers along the path from source to destination. Again, any such buffering either at the source or the intermediate nodes should also be controllable according to any QoS parameters that have been associated with the route request.

V. CONCLUSION

Because protocol design is not yet an exact science, designers should take advantage of those tools which may aid them in validating the operation of their protocols. The use of design verification tools can aid in the examination of each possible usage case, and can validate the operation of a protocol in each of these situations. Because of the difficulty in enumerating all possible usage cases and node failure scenarios, these tools should be considered an important part of the protocol design process.

Implementing a routing protocol is a crucial step in verifying the correct design and operation of the protocol. While simulation is necessary for ad hoc routing protocols in order to establish a set of repeatable test scenarios where small variances can be made, certain assumptions and simplifications are often made in the simulation which do not hold true in a real world scenario. Hence it is essential to implement the protocol in order to ensure no over-simplifications are included in the final protocol specification.

We have developed an implementation of the AODV routing protocol based on the simulation of this protocol. In the course of writing the implementation, some key changes needed to be made to both the protocol and the Linux kernel to enable AODV to operate correctly. As AODV continues to be refined, it is possible that further changes will be required, particularly when QoS operation is implemented. Additionally, tunnel management may also indicate the need for further modifications. We look forward to the completion of the implementation, the design of a testbed in which to test the implementation, and interoperability testing with other existing implementations.

ACKNOWLEDGMENT

Thanks to Michael Speer for his help in designing the basic framework for using IGMP in the ad hoc networking environment.

References

[1] S. H. Bae, S.-J. 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*, 14(1):70–77, January/February 2000.

[2] K. Bhargavan, C. A. Gunter, and D. Obradovic. Fault Origin Adjudication. *Proceedings of the Workshop on Formal Methods in Software Practice*, Portland, OR, August 2000.

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

[4] D. A. Maltz, J. Broch, and D. B. Johnson. Experiences Designing and Building a Multi-hop Wireless Ad hoc Testbed. Technical Report CMU, CMU School of Computer Science.

[5] C. E. Perkins and E. M. Royer. Ad-hoc On-Demand Distance Vector Routing. *Proceedings of the 2^{nd} IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, February 1999.

[6] C. E. Perkins, E. M. Royer, and S. R. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. *IETF Internet Draft, draft-ietf-manet-aodv-05.txt*, March 2000. (Work in Progress).

[7] E. M. Royer and C. E. Perkins. Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol. *Proceedings of the 5^{th} ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 207–218, Seattle, WA, August 1999.