AODV Routing Protocol Implementation Design

Ian D. Chakeres
Dept. of Electrical & Computer Engineering
University of California, Santa Barbara
idc@engineering.ucsb.edu

Elizabeth M. Belding-Royer
Dept. of Computer Science
University of California, Santa Barbara
ebelding@cs.ucsb.edu

Abstract

To date, the majority of ad hoc routing protocol research has been done using simulation only. One of the most motivating reasons to use simulation is the difficulty of creating a real implementation. In a simulator, the code is contained within a single logical component, which is clearly defined and accessible. On the other hand, creating an implementation requires use of a system with many components, including many that have little or no documentation. The implementation developer must understand not only the routing protocol, but all the system components and their complex interactions. Further, since ad hoc routing protocols are significantly different from traditional routing protocols, a new set of features must be introduced to support the routing protocol. In this paper we describe the event triggers required for AODV operation, the design possibilities and the decisions for our Ad hoc On-demand Distance Vector (AODV) routing protocol implementation, AODV-UCSB. This paper is meant to aid researchers in developing their own on-demand ad hoc routing protocols and assist users in determining the implementation design that best fits their needs.

1. Introduction

Simulation is an important tool in the development of mobile ad hoc networks; it provides an excellent environment to experiment and verify routing protocol correctness. However, simulation does not guarantee that the protocol works in practice, because simulators contain assumptions and simplified models that may not actually reflect real network operation.

After a protocol is thoroughly tested in simulation, an implementation is the logical next step. A working implementation is necessary to validate that the routing protocol specification performs under real conditions. Otherwise, assumptions made by the protocol design cannot be verified as correct. Additionally, an implementation can be used to perform testbed and field tests. Eventually it can be used in a deployed system, such as [1].

Creating a working implementation of an ad hoc routing protocol is non-trivial and more difficult than developing a simulation. In simulation, the developer controls the whole system, which is in effect only a single component. An implementation, on the other hand, needs to interoperate with a large, complex system. Some components of this system are the operating system, sockets, and network interfaces. Additional implementation problems surface because current operating systems are not built to support ad hoc routing protocols. A number of required events are unsupported; support for these events must be added. Because these events encompass many system components, the components and their interactions must also be explored. For these reasons it takes significantly more effort to create an ad hoc routing protocol implementation than a simulation.

Nevertheless, as an important step in studying the AODV routing protocol [12], we created the AODV-UCSB implementation. We performed experiments and validated the AODV routing protocol design using our implementation.

Understanding the operation and design process of our system will help other researchers with the development of their own ad hoc routing protocols. Identifying the strengths and weaknesses of our implementation also helps system designers decide whether our AODV implementation fits their requirements. Specifically, the contributions of this paper are the following:

- Definition of needed AODV triggers currently unsupported by operating systems.
- Discussion of different design strategies.
- Description of the chosen design for our AODV-UCSB implementation.
- Presentation of publicly available AODV implementation designs.

The outline for the remainder of the paper is as follows. An overview of the key components of our system is presented in section 2. Section 3 enumerates the currently unsupported events needed by the AODV routing protocol and

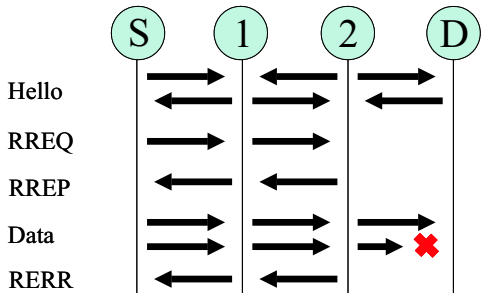

Figure 1. AODV Protocol Messaging.

discusses possible techniques for determining them. Section 4 discusses other AODV implementations, and finally section 5 concludes the paper.

2. Background

Before we describe the requirements and design, we highlight some of the key components of our system. First we describe the AODV routing protocol and its basic operation. Next the IEEE 802.11 standard is described. In our testbed we utilize IEEE 802.11 for the physical, MAC and link layer of the nodes to accomplish wireless communication. Finally, we discuss Netfilter, a mechanism inside the Linux protocol stack that allows packet manipulation.

2.1. AODV Protocol Overview

The AODV [11, 12] routing protocol is a reactive routing protocol; therefore, routes are determined only when needed. Figure 1 shows the message exchanges of the AODV protocol.

Hello messages may be used to detect and monitor links to neighbors. If Hello messages are used, each active node periodically broadcasts a Hello message that all its neighbors receive. Because nodes periodically send Hello messages, if a node fails to receive several Hello messages from a neighbor, a link break is detected.

When a source has data to transmit to an unknown destination, it broadcasts a Route Request (RREQ) for that destination. At each intermediate node, when a RREQ is received a route to the source is created. If the receiving node has not received this RREQ before, is not the destination and does not have a current route to the destination, it rebroadcasts the RREQ. If the receiving node is the destination or has a current route to the destination, it generates a Route Reply (RREP). The RREP is unicast in a hop-by-hop fashion to the source. As the RREP propagates, each intermediate node creates a route to the destination. When the source receives the RREP, it records the route to the destination and can begin sending data. If multiple RREPs are received by the source, the route with the shortest hop count is chosen.

As data flows from the source to the destination, each node along the route updates the timers associated with the routes to the source and destination, maintaining the routes in the routing table. If a route is not used for some period of time, a node cannot be sure whether the route is still valid; consequently, the node removes the route from its routing table.

If data is flowing and a link break is detected, a Route Error (RERR) is sent to the source of the data in a hop-by-hop fashion. As the RERR propagates towards the source, each intermediate node invalidates routes to any unreachable destinations. When the source of the data receives the RERR, it invalidates the route and reinitiates route discovery if necessary.

2.2. IEEE 802.11 Standard

The IEEE 802.11 Standard [4] is by far the most widely deployed wireless LAN protocol. This standard specifies the physical, MAC and link layer operation we utilize in our testbed. Multiple physical layer encoding schemes are defined, each with a different data rate. Part of each transmission uses the lowest most reliable data rate, which is 1 Mbps.

At the MAC layer IEEE 802.11 uses both carrier sensing and virtual carrier sensing prior to sending data to avoid collisions. Virtual carrier sensing is accomplished through the use of Request-To-Send (RTS) and Clear-To-Send (CTS) control packets. When a node has a unicast data packet to send to its neighbor, it first broadcasts a short RTS control packet. If the neighbor receives this RTS packet, then it responds with a CTS packet. If the source node receives the CTS, it transmits the data packet. Other neighbors of the source and destination that receive the RTS or CTS packets defer packet transmissions to avoid collisions by updating their network allocation vector (NAV). The NAV is used to perform virtual channel sensing by indicating that the channel is busy, as shown in Figure 2.

After a destination properly receives a data packet, it sends an acknowledgment (ACK) to the source. This signi-

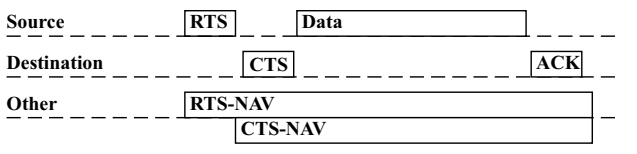

Figure 2. IEEE 802.11 Distributed Coordination Function.

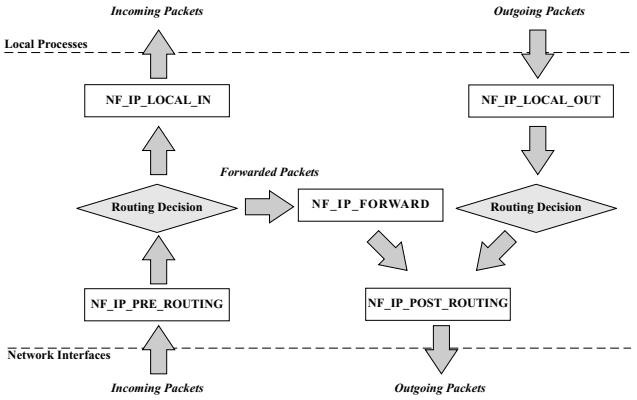

Figure 3. Netfilter Hooks.

fies that the packet was correctly received. This procedure (RTS-CTS-Data-ACK) is called the Distributed Coordination Function (DCF). For small data packets the RTS and CTS packets may not be used.

If an ACK (or CTS) is not received by the source within a short time limit after it sends a data packet (or RTS), the source will attempt to retransmit the packet up to seven times. If no ACK (or CTS) is received after multiple retries, an error is issued by the hardware indicating that a failure to send has occurred.

Broadcast data packets are handled differently than unicast data packets. Broadcast packets are sent without the RTS, CTS or ACK control packets. These control messages are not needed because the data is simultaneously transmitted to all neighboring nodes.

2.3. Netfilter

Netfilter [5] is used by our implementation to identify many of the events that trigger routing protocol action. Netfilter consists of a number of hooks at various points inside the Linux protocol stack. It allows user-defined kernel modules to register callback functions to these hooks. When a packet traverses a hook, the packet flows through the user-defined callback method inside the kernel module.

There are five hooks defined in the Netfilter architecture, shown as boxes in Figure 3. At the top of the figure there are two hooks, NF_IP_LOCAL_IN and NF_IP_LOCAL_OUT. These hooks are for all packets to and from local processes. At the bottom of the figure there are two hooks, NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING. These are for all packets from and to other hosts on the network. There is also a hook for packets that are forwarded by the current host, NF_IP_FORWARD. As an example of how packets traverse these hooks, suppose a packet is created by a local process for a remote process. It first traverses the NF_IP_LOCAL_OUT hook. Next, a routing decision is performed to see if the packet is bound for the local host or another host on the network. The packet is found to be destined for a remote host, and the packet is passed through the NF_IP_POST_ROUTING hook and then onto a network interface.

To demonstrate how Netfilter is used in practice, we describe a simple example that drops all locally created outgoing packets to a particular destination address. First, a kernel module is created that attaches the NF_IP_LOCAL_OUT Netfilter hook to a callback method written to examine packets. This callback method is called for each locally created outgoing packet. If the packet's destination address matches the destination address being filtered, then the callback method drops the packet. After compiling and loading the kernel module, any packet locally created and destined for that particular destination address is dropped. In this manner a kernel module can examine, drop, discard, modify or queue packets at any of the defined Netfilter hooks.

3. Implementation Design

The AODV-UCSB implementation was developed on the Linux 2.4 kernel. A user-space daemon was chosen to keep as much logic as possible out of the kernel. This is a common design for routing protocols because code within the kernel operates with different privileges, and a single error in the kernel space can cause the whole operating system to fail.

For the AODV routing daemon to function it must determine when to trigger AODV protocol events. Since the IP stack was designed for static networks where link disconnections are infrequent and packet losses are unreported, most of these triggers are not readily available. Therefore, these events must be extrapolated and communicated to the routing daemon via other means. The events that must be determined are:

- *When to initiate a route request:* This is indicated by a locally generated packet that needs to be sent to a destination for which a valid route is not known.
- *When and how to buffer packets during route discovery:* During route discovery packets destined for the unknown destination should be queued. If a route is found the packets are be sent.
- *When to update the lifetime of an active route:* This is indicated by a packet being received from, sent to or forwarded to a known destination.
- *When to generate a RERR if a valid route does not exist:* If a data packet is received from another host and there is no known route to the destination, the node

must send a RERR so that the previous hops and the source halt transmitting data packets along this invalid route.

- *When to generate a RERR during daemon restart:* After the AODV routing protocol restarts, it must send a RERR message to other nodes attempting to use it as a router. This behavior is required in order to ensure no routing loops occur.

In the remainder of this section we discuss various design approaches. First, we examine how to determine these events and where to place the AODV protocol logic. We describe the advantages and disadvantages of each solution, and we justify why we chose a user-space daemon with a small kernel module. In addition, we discuss the importance of monitoring neighbor connectivity and how it is performed.

3.1. Design Possibilities

There are many ways to design the AODV protocol to extrapolate the needed AODV events. Possible opportunities for obtaining the events include:

- Snooping
- Kernel modification
- Netfilter

In the following sections, each of these possibilities is described and their respective strengths and weaknesses examined.

3.1.1. Snooping. One possibility for determining the needed events is to promiscuously snoop all incoming and outgoing packets [8]. The code to perform snooping is built into the kernel and is available to user-space programs, as shown in Figure 4. The snooping feature can be used to determine the events listed in section 3. For instance, an ARP packet is generated when a node does not know the MAC layer address of the next hop. Using this inference, if an ARP request packet is seen for an unknown destination and it is originated by the local host, then a route discovery needs to be initiated. In a similar manner, all the other AODV events may be determined by monitoring the incoming and outgoing packets.

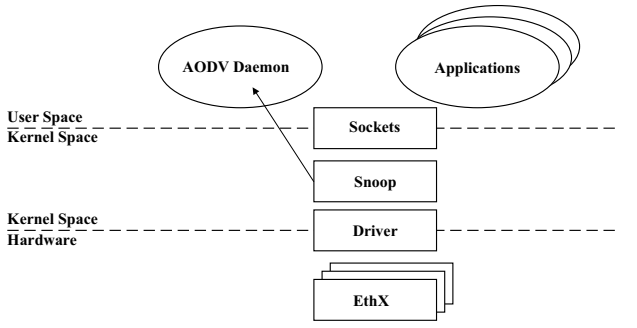

Figure 4. Snooping Architecture.

The most important advantage of this solution is it does not require any code to run in the kernel-space. Hence this solution allows for simple installation and execution. The two main disadvantages of this solution are overhead and dependence on ARP. For example, the determination of the need for route discovery is indicated by an ARP request. Since route discovery is initiated by outgoing ARP packets, these outgoing packets are unnecessary overhead, and they waste bandwidth. There are also problems with the dependence on ARP. If the routing table and ARP cache become out of sync, it is possible that the routing protocol may not function properly. For example, if the ARP cache contains an entry for a particular unknown destination, then an ARP packet will not be generated for this destination even though it is not known by the routing daemon. Consequently, route discovery will not be initiated. For proper operation the routing protocol must monitor and control the ARP cache in addition to the IP routing table, because disagreement between the two can cause the routing protocol to function improperly.

3.1.2. Kernel Modification. Another possibility to determine the AODV events is to modify the kernel. Code can be placed in the kernel to communicate the events listed in section 3 to an AODV user-space daemon. For example, to initiate route discovery, code is added in the kernel at the point where route lookup failures occur. Given this code in the kernel, if a route lookup failure happens, then a method is called in the user-space daemon. Figure 5 shows the architecture of the AODV daemon and the required support logic.

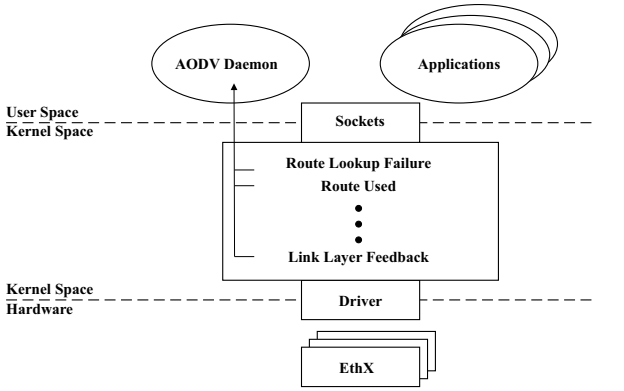

Figure 5. Kernel Modification Architecture.

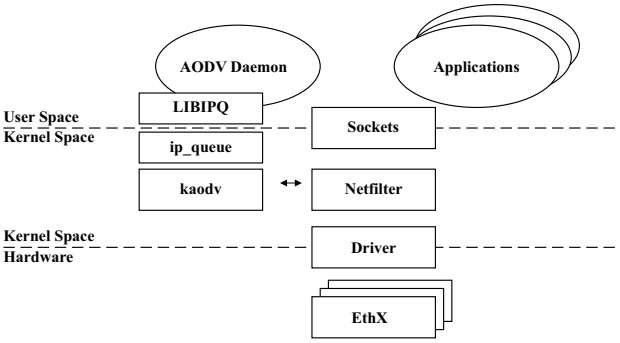

Figure 6. Netfilter Architecture.

The advantages of this solution are that the events are explicitly determined and there is no wasted overhead. The main disadvantages of this solution are user installation and portability. Installation of the necessary kernel modifications requires a complete kernel recompilation. This is a difficult procedure for many users. Also, kernel patches are often not portable between one kernel version and the next. Finally, understanding the Linux kernel and network protocol stack requires examining a significant amount of uncommented, complex code.

3.1.3. Netfilter. Netfilter is a set of hooks at various points inside the Linux protocol stack, as described in section 2.3. Netfilter redirects packet flow through user defined code, which can examine, drop, discard, modify or queue the packets for the user-space daemon. Using Netfilter is similar to the snooping method, described in section 3.1.1; however, it does not have the disadvantage of unnecessary overhead or dependence on ARP.

Compared to the other possibilities, this solution has many strengths. These include that there is no unnecessary communication, it is highly portable, it is easy to install and the user-space daemon can determine all the required events in section 3.

On the other hand, the disadvantage of this solution is that it requires a kernel module. However, a kernel module is easier to install than a kernel modification. Since only the kernel module must be recompiled, there is no need to recompile the complete kernel. Also, the kernel module can be loaded or unloaded at any time. Finally, a kernel module is more portable than kernel modification because it depends only on the Netfilter interface. This interface does not change from one kernel version to the next.

Since Netfilter has the fewest and least significant disadvantages of the strategies examined, we utilize it in our final implementation architecture, as shown in Figure 6. Our implementation uses Netfilter hooks to redirect packets that arrive from the local machine (NF_IP_LOCAL_OUT), from other machines (NF_IP_PRE_ROUTING), as well as all packets that are sent to other machines (NF_IP_POST_ROUTING). These hooks are used by the *kaodv* kernel module. The *ip_queue* module is used to queue these packets for the user-space daemon. There the AODV daemon uses *libipq* to make control decisions about each packet.

3.2. Determining Local Connectivity

To avoid wasting bandwidth and energy, it is beneficial for the sender of a data packet to have assurance that the next hop is within transmission range and is likely to receive the packet. In order to verify that the next hop is receiving data packets, local connectivity must be monitored. Notification of the inability to send data packets to a neighbor is needed to promptly notify the source that a path is broken; otherwise, the source continues to send data packets, wasting resources. The AODV routing protocol uses RERR messages to notify the source and all nodes on the route to the source of the broken link. Because other solutions are not currently available, all current implementations utilize Hello messages. Unfortunately, Hello messages are known to perform poorly in a number of common scenarios [3, 10].

4. AODV Implementation Comparison

Recently there have been many AODV routing protocol implementations, including Mad-hoc [8], AODV-UCSB [2], AODV-UU [9], Kernel-AODV [7] and AODV-UIUC [6]. Each implementation was developed and designed independently; but, they all perform the same operations and many interoperate.

The first publicly available implementation of AODV was Mad-hoc. The Mad-hoc implementation resides completely in user-space and uses the snooping strategy to determine AODV events. Unfortunately, it is known to have bugs that cause it to fail to perform properly. These problems are related to its use of ARP. Another feature missing from the Mad-hoc implementation is proper queuing of data packets during route discovery. Mad-hoc is no longer actively researched, supported or available.

The first release of AODV-UCSB used the kernel modification strategy. This AODV-UCSB implementation was developed before Netfilter was well documented. We found that it suffered from some intermittent problems. These were due to unforeseen dependencies within the kernel that were brought out by our specific kernel modifications. After Netfilter had matured, AODV-UCSB was updated to use Netfilter. AODV-UCSB uses the Netfilter kernel modules from the AODV-UUv0.4 release. Using these kernel modules, all interesting packets are passed to the user-space daemon for processing, as described in section 2.3. In addition

to the base AODV specification, a number of Hello message options are available. These include requiring reception of multiple Hello messages before neighbor connectivity is established. This avoids creating routes to neighbors based on a single spurious message reception.

AODV-UU has the same design as AODV-UCSB; it uses kernel modules to utilize the netfilter hooks. The main protocol logic resides in a user-space daemon. AODV-UU has also been ported to the NS-2 simulator. This allows the real-world implementation code to be run in a simulation environment. The authors have also added a number of supplemental features, not part of the AODV draft, to increase the performance of Hello messages [10] (e.g., unidirectional link support and a signal quality threshold for received packets). In addition, AODV-UU also includes Internet gatewaying and multiple interface support. Since AODV-UU is well documented and able to run in simulation, a number of patches are available (e.g., multicast and subnetting) to further extend its functionality.

Kernel-AODV uses Netfilter and all of the routing protocol logic is placed inside the kernel module; therefore, no user-space daemon is needed. This improves the performance of the implementation, in terms of packet handling, since no packets are required to traverse from the kernel to the user-space. This implementation also supports Internet gatewaying, multiple interfaces and a basic multicast protocol. There is also a *proc* file interface for users to monitor signal strength to neighbors when certain wireless hardware is used.

The AODV-UIUC implementation uses Netfilter wrapped by the Ad hoc Support Library (ASL) [6]. This design is similar to AODV-UCSB and AODV-UU except it explicitly separates the routing and forwarding functions. Routing protocol logic takes place in the user-space daemon, while packet forwarding is handled in the kernel. This is efficient because forwarded packets are handled immediately and fewer packets traverse the kernel to user-space boundary.

All of the implementations discussed use Hello Messages to determine local connectivity and detect link breaks. In addition, all implementations (except Mad-hoc) support the expanding ring search and local repair optimizations [11].

5. Conclusion

In this paper we analyzed the design possibilities for an AODV implementation. We first identified the unsupported events needed for AODV to perform routing. We then examined the advantages and disadvantages of three strategies for determining this information. This analysis supported our decision to use small kernel modules with a user-space daemon. Finally, we presented the design of many publicly available AODV implementations. We hope that the information in this paper aids researchers in understanding the trade-offs in ad hoc routing protocol implementation development. Further, the description of the design structure and additional features of each implementation can assist users in deciding which implementation best fits their needs.

Acknowledgment

This work is supported in part by Intel Corporation through a UC Core grant and by a NSF Infrastructure grant (EIA - 0080134).

References

[1] NovaRoam. http://www.novaroam.com/.

[2] I. D. Chakeres. AODV-UCSB Implementation from University of California Santa Barbara. http://moment.cs.ucsb.edu/AODV/aodv.html.

[3] I. D. Chakeres and E. M. Belding-Royer. The Utility of Hello Messages for Determining Link Connectivity. In *Proceedings of the* 5^{th} *International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pages 504–508, Honolulu, Hawaii, October 2002.

[4] IEEE Computer Society. IEEE 802.11 Standard, IEEE Standard For Information Technology, 1999.

[5] J. Kadlecsik, H. Welte, J. Morris, M. Boucher, and R. Russell. Netfilter. http://www.netfilter.org/.

[6] V. Kawadia, Y. Zhang, and B. Gupta. System Services for Implementing Ad-Hoc Routing: Architecture, Implementation and Experiences. In *Proceedings of the* 1^{st} *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 99–112, San Francisco, CA, June 2003.

[7] L. Klein-Berndt. Kernel AODV from National Institute of Standards and Technology (NIST). http://w3.antd.nist.gov/wctg/aodv_kernel/.

[8] F. Lilieblad, O. Mattsson, P. Nylund, D. Ouchterlony, and A. Roxenhag. Mad-hoc AODV Implementation and Documentation. http://mad-hoc.flyinglinux.net.

[9] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrm, and C. F. Tschudin. A Large-scale Testbed for Reproducible Ad hoc Protocol Evaluations. In *IEEE Wireless Communications and Networking Conference 2002 (WCNC)*, March 2002.

[10] H. Lundgren, E. Nordstrm, and C. Tschudin. Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks. Technical Report 2002-022, Uppsala University Department of Information Technology, June 2002.

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

[12] C. E. Perkins and E. M. Royer. The Ad hoc On-Demand Distance Vector Protocol. In C. E. Perkins, editor, *Ad hoc Networking*, pages 173–219. Addison-Wesley, 2000.