AODV Implementation Design and Performance Evaluation

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

Elizabeth M. Belding-Royer
Dept. of Computer Science
University of California, Santa Barbara
Email: 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 object, which is clearly defined and accessible. On the other hand, creating an implementation requires use of a system with many components. Consequently, the implementation developer must understand not only the routing protocol, but all the system parts 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 proper operation, the design possibilities and the decisions for our Ad hoc On-demand Distance Vector (AODV) routing protocol implementation, AODV-UCSB. We also summarize the design of other publicly available AODV implementations. Finally, we provide a comparison of different design approaches. 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.

I. 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 verifies that the routing protocol specification performs under real conditions. Otherwise, assumptions made by the protocol design cannot be confirmed 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 include 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, and support for these events must therefore 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 [2], we created the AODV-UCSB implementation [3]. We then performed experiments and validated the AODV routing protocol design using our implementation [4]. This paper describes the operation and design process of both our own and many other AODV implementations. We hope that this paper will help other researchers with the development of their own ad hoc routing protocols. Identifying the strengths and weaknesses of each available implementation also helps system designers decide which AODV implementation fits their requirements. Specifically, the contributions of this paper are the following:

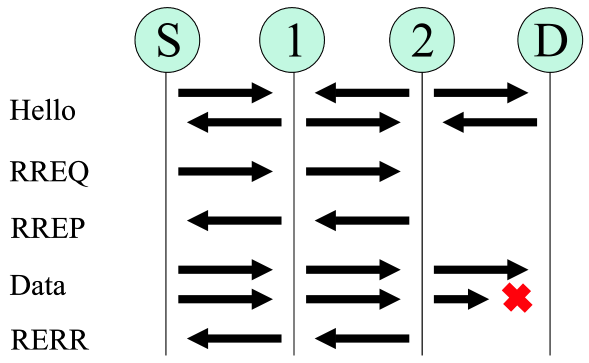

Fig. 1. AODV Protocol Messaging.

- Definition of AODV triggers currently unsupported by operating systems
- Discussion of different design strategies
- Description of the chosen design for our AODV-UCSB implementation
- Description of other publicly available AODV implementation designs
- Evaluation of different AODV forwarding designs
- Measurement of link break detection latency

The outline for the remainder of the paper is as follows. An overview of the key components of our system is presented in Section II. Section III enumerates the currently unsupported events needed by the AODV routing protocol and discusses possible techniques for determining them. Section IV describes design approaches taken by other AODV implementations. We also evaluate packet forwarding design strategies and measure the latency of link break detection for various designs. Finally, Section V concludes the paper.

II. 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 enable wireless communication. Finally, we discuss Netfilter, a mechanism inside the Linux protocol stack that allows packet manipulation.

A. AODV Protocol Overview

The AODV [2, 5] routing protocol is a reactive routing protocol; therefore, routes are determined only when needed. Figure 1 illustrates 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. Hello message behavior is described in more detail in Section III-B.

When a source has data to transmit to an unknown destination, it broadcasts a Route Request (RREQ) for that destination. When a RREQ is received by an intermediate node, a route to the source is created. If the receiving node has not received the 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 begins sending data. If multiple RREPs are received by the source, the route with the shortest hop count is chosen.

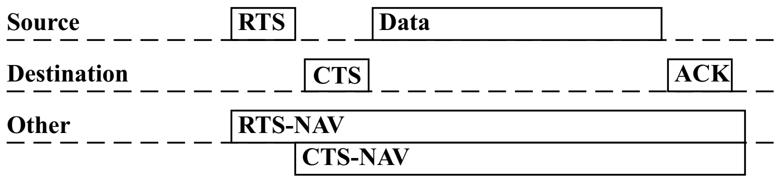

Fig. 2. IEEE 802.11 Distributed Coordination Function.

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) message 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.

B. IEEE 802.11 Standard

The IEEE 802.11 Standard [6] is by far the most widely deployed wireless LAN protocol. This standard specifies the physical, MAC and link layer operation utilized in our testbed. Multiple physical layer encoding schemes are defined, each with a different data rate.

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 unicast data packet, it sends an acknowledgment (ACK) to the source. This signifies 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, because they can actually hinder performance [7].

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.

C. Netfilter

Netfilter [8] 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

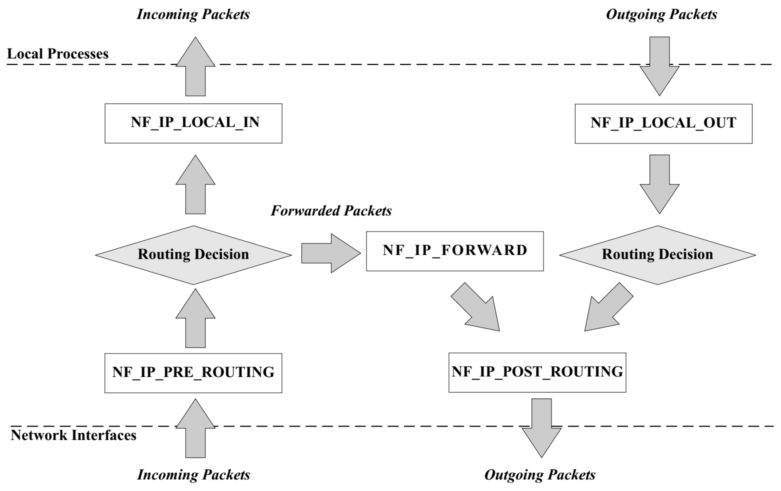

Fig. 3. Netfilter Hooks.

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 whether the packet is bound for the local host or another host on the network. In our example, the packet is found to be destined for a remote host, so 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.

III. 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 be determined 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 RREQ:* 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 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.

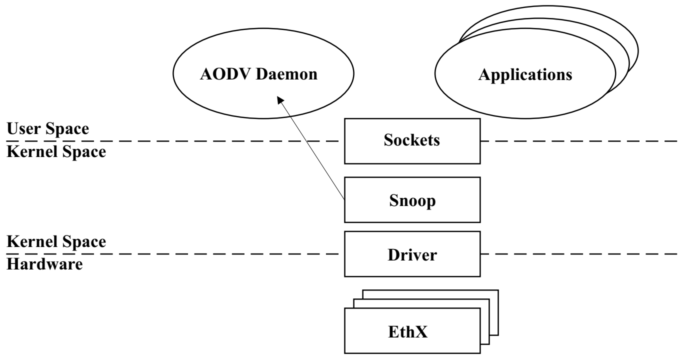

Fig. 4. Snooping Architecture.

- *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 valid route to the destination, the node must send a RERR so that the previous hops and the source stop 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 Section III-A, we discuss various design approaches for determining these events. 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 our choice of a user-space daemon with a small kernel module.

In addition to the AODV events listed, each node must also detect link breaks in active routes. When a link breaks, AODV invalidates affected routes. Afterward, if a data packet for the affected route is received, a RERR is sent to the source. If local connectivity is not monitored for link breaks, intermediate nodes and sources continue to forward undeliverable data packets, wasting resources. The main two methods for monitoring local connectivity and detecting link breaks are described in Section III-B.

A. Design Possibilities

There are many ways for AODV implementations to receive the needed AODV events. Possible opportunities for obtaining the events include:

- Snooping
- Kernel modifi cation
- Netfi lter

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

1) Snooping: One possibility for determining the needed events is to promiscuously snoop all incoming and outgoing packets [9]. The code to perform snooping is built into the kernel and is available to user-space programs by using the Packet Capture Library (*libpcap*), as shown in Figure 4. The snooping feature can be used to determine the events listed in Section III. For instance, each packet that is transmitted is passed to the routing daemon using libpcap. When the daemon sees that a packet was transmitted along an active route, the lifetime for that route is updated so that it does not expire, since it is in use. In a similar manner, all the other AODV events may be determined by monitoring the incoming and outgoing packets.

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, an ARP packet is generated when a node does not know the MAC address of the next hop. Using this inference, if an ARP request packet

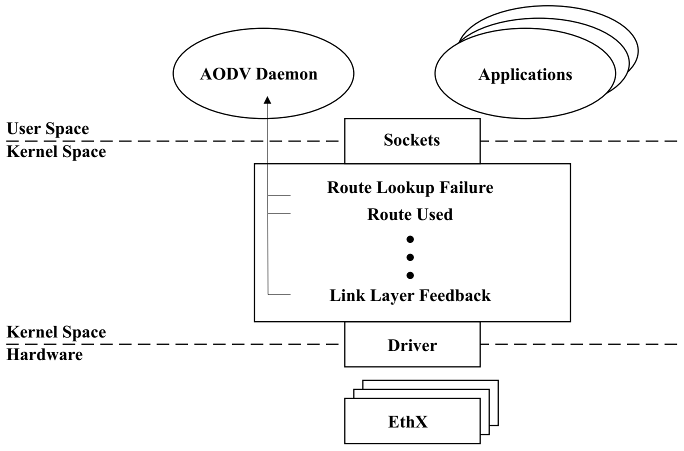

Fig. 5. Kernel Modifi cation Architecture.

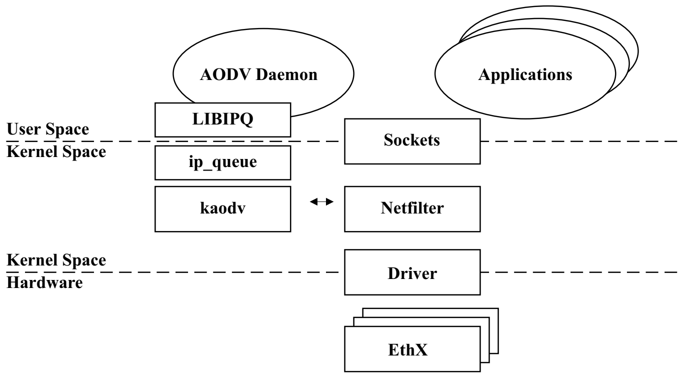

Fig. 6. Netfi lter Architecture.

is seen for an unknown destination and it is originated by the local host, then a route discovery needs to be initiated. 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 the destination 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 incorrectly.

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 III 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.

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 modifi cations requires a complete kernel recompilation. This is a diffi cult 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 signifi cant amount of uncommented, complex code.

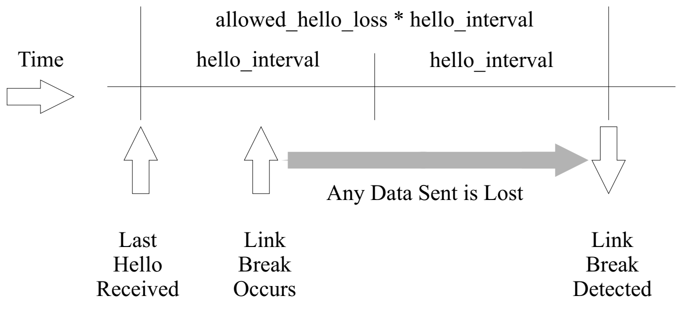

Fig. 7. Hello Message Link Loss Detection.

3) Netfilter: Netfilter is a set of hooks at various points inside the Linux protocol stack, as described in Section II-C. Netfilter redirects packet flow through user defined code, which can examine, drop, discard, modify or queue the packets for the user-space daemon. Netfilter is similar to the snooping method, described in Section III-A.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 III. 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 a 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 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.

To detect broken links in active routes our implementation utilizes Hello messages. This is one of the most common approaches for monitoring local connectivity. In the next section, we describe how Hello messages and link layer feedback are used to detect link breaks.

B. Determining Local Connectivity

To avoid wasting bandwidth and energy, it is beneficial for the sender of a data packet to verify that the next hop is within transmission range and is likely to receive the packet. To accomplish this, local connectivity must be monitored. Notification of the inability to send data packets to an active 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. The primary methods used by AODV to detect the loss of local connectivity to a neighbor are Hello messages and link layer feedback.

Hello messages are periodic locally broadcast messages that are utilized to indicate link availability. In AODV, Hello messages and broadcast messages can serve the same function. For example, RREQ messages are broadcast IP packets; therefore, reception of a RREQ indicates the presence of a link. Hence, the term Hello message is used to loosely refer to all broadcast control messages. In AODV, reception of a Hello message indicates bidirectional connectivity to the sender. Once a link is established, failure to receive several Hello messages from a neighbor indicates a loss of connectivity.

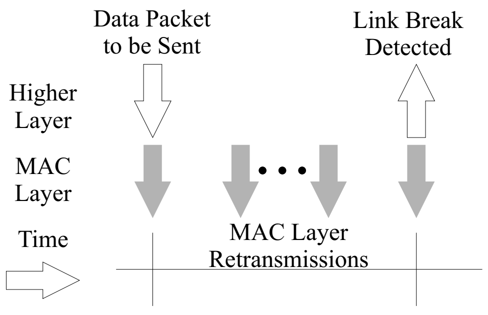

Fig. 8. IEEE 802.11 Link Layer Feedback.

When Hello messages are used, each active node broadcasts a control message at least once every $hello\_interval$ seconds. Failure to receive a Hello message for $allowed\_hello\_loss * hello\_interval$ seconds indicates a loss of connectivity to that neighbor. The recommended value of $allowed\_hello\_loss$ is two and that of $hello\_interval$ is one second [5].

This solution, while simple to implement, reacts slowly when compared with link layer feedback. As shown in Figure 7, using Hello messages may require as long at $allowed\_hello\_loss * hello\_interval$ seconds to detect the loss of connectivity to a neighbor. During this time all packets sent to that neighbor may be lost. In addition to being slow, Hello messages increase the control overhead. This is undesirable because Hello message transmissions consume energy and may collide or otherwise hinder data transmission. In addition, Hello messages are known to perform poorly in a number of common scenarios [4, 10].

In contrast to Hello messages, link layer feedback is able to quickly identify link failures during transmission of a data packet to another node. This feedback is assumed to be provided by the underlying MAC protocol, in our case IEEE 802.11. In IEEE 802.11, a data packet is first queued for transmission at the MAC layer, as shown in Figure 8. If the packet cannot be transmitted after multiple MAC layer retries, an indication is given, to the higher layers, that a failure has occurred. This results in immediate notification of a broken link as soon as a packet fails to be transmitted. Hence, this approach has significantly lower latency in detecting link breaks than the use of Hello messages. As a consequence, fewer data packets are lost and the route can be repaired more quickly.

Until recently link layer feedback was not available. However, Linux now supports link layer feedback. The Wireless Extensions [11] are a standard API for wireless drivers and programs. Using the Wireless Extensions, hardware drivers are abstracted from user programs and vice versa. After the Wireless Extensions added the ability to communicate MAC layer transmission failures, support was added to the HostAP driver [12]. The driver now passes transmission failure events from the wireless hardware to user applications using the Wireless Extensions API. Other hardware drivers are in the process of adding support for transmission error notification. Since the Wireless Extensions API and drivers are recent additions to Linux, none of the current AODV implementations support link layer feedback. Instead, Hello messages are utilized to detect the loss of local connectivity. Note that both of these link break detection techniques may inaccurately signal link breaks in congested environments.

IV. Evaluation of Design Strategies

In this section we first describe the design choices of publicly available AODV implementations. Then we evaluate the design approaches by measuring the latency of kernel-space and user-space forwarding, as well as the difference in link break detection latency when using Hello messages and link layer feedback. This section identifies the strengths and weaknesses of different design decisions.

A. AODV Implementation Comparison

Recently there have been many AODV routing protocol implementations, including Mad-hoc [9], AODV-UU [13], AODV-UCSB [14], Kernel-AODV [15] and AODV-UIUC [16]. Each implementation was developed and designed independently; however, they all perform the same operations and many have been shown to interoperate [17].

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 [18]. Some problems are related to its dependence on 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.

AODV-UU and AODV-UCSB share the same basic design; they both use kernel modules to attach to the netfilter hooks. The main protocol logic resides in a user-space daemon. A significant feature of the AODV-UU implementation is that it 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 improve 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.

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 the implementation suffered from some intermittent problems, such as performing unnecessary route requests. 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 now 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 II-C. 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.

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) [16]. 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-space 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 [5].

B. Forwarding Design Performance

In Section IV-A we described the design strategies of public implementations. In AODV-UU and AODV-UCSB, all protocol logic is located in the user-space. This requires all packets to traverse the user-space/kernel-space boundary twice; once when the packet is received, and then again if the packet is forwarded. In Kernel-AODV, all protocol logic is located in the kernel. In AODV-UIUC, most protocol logic is located in user-space. However, packet forwarding takes place in the kernel-space to minimize the per packet cost, in terms of delay, load and energy. In general, as shown in Table I, the designs fall into two categories for packet forwarding: user-space and kernel-space forwarding.

TABLE I

FORWARDING LOGIC LOCATION.

AODV Protocol Implementation	Forwarding Logic Location
Mad-hoc	Kernel-space
AODV-UCSB	User-space
AODV-UU	User-space
Kernel-AODV	Kernel-space
AODV-UIUC	Kernel-space

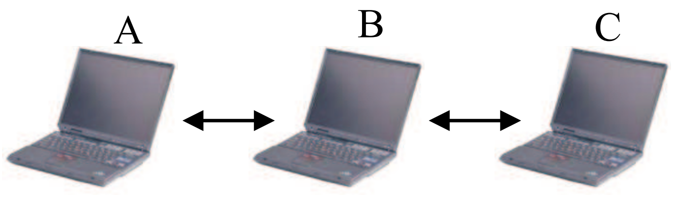

Fig. 9. Network Topology for Forwarding Strategy Tests.

In order to evaluate the two different design strategies we performed a series of experimental tests. In the tests three nodes were configured in a line, as shown in Figure 9. Neighbor connectivity was controlled using *iptables*, which performed MAC layer filtering. During each test run 200 ICMP packets were exchanged between nodes A and C (100 in each direction at an interval of one second). *tcpdump* was used to record the time each packet was received and forwarded by node B. The time between reception and forwarding of a packet gives insight to the amount of processing needed to forward the data packet.

Each test was performed using a kernel-space (Kernel-AODV) and a user-space (AODV-UU) implementation. Two sets of tests were performed, one using a laptop and one using a handheld as the forwarding node (B). In the laptop tests, each node was an IBM Thinkpad Pentium III 1 GHz laptop with 256M RAM running Linux 2.4.23. In the handheld tests, nodes A and C were IBM Thinkpads and the center node (B) was a Compaq iPAQ 3670. The iPAQ contained a 206 Mhz StrongARM processor, 32M RAM and ran Linux 2.4.18. In comparison to the laptop, the handheld is a resource limited device. Consequently, we expect the limitations of each forwarding technique to be more pronounced with the iPAQ.

The results of all the tests are presented in Table II. When using laptops for all nodes, the kernel-space forwarding time is very short, on the order of tens of microseconds. On the other hand, the user-space forwarding strategy requires additional processing time, when compared to the kernel design. On average it took ten times longer to process each packet in the user-space AODV implementation. Though the user-space forwarding took much longer than kernel-space forwarding, the overall impact on application traffic was not noticeable from the overall ping RTT time.

In the handheld tests, both forwarding techniques took significantly longer, since the handheld is a resource limited device. Comparing kernel-space to user-space forwarding, the increase in latency is only

TABLE II

EXPERIMENTAL FORWARDING RESULTS.

Forwarding Node Type	Forwarding Strategy	Delay (us)
Laptop	Kernel-space (Kernel-AODV)	14
Laptop	User-space (AODV-UU)	137
Handheld	Kernel-space (Kernel-AODV)	282
Handheld	User-space (AODV-UU)	774

TABLE III

LINK BREAK DETECTION LATENCY.

Strategy	Latency
Hello Messages	1-2 (s)
Link Layer Feedback	11.4 (ms)

a factor of about 2.75. However, the overall impact in terms of latency is much larger, nearly a half millisecond longer for user-space forwarding. Over multiple hops this additional latency is more likely to be noticeable to user applications.

C. Link Break Detection Latency

In Section III-B we described the two most common methods for determining local connectivity and motivated the importance of monitoring neighbor connectivity. In this section we compare the latency for link break detection using Hello messages and link layer feedback.

The amount of time it takes to detect a link break in AODV using Hello messages depends on two parameters: $allowed\_hello\_loss$ and $hello\_interval$. Using these two variables it takes between $hello\_interval * allowed\_hello\_loss - 1$ seconds and $hello\_interval * allowed\_hello\_loss$ seconds to detect the loss of a link. For example, using the values suggested in the AODV RFC [5], $allowed\_hello\_loss = 2$ and $hello\_interval = 1$ second, it takes between one and two seconds to detect a link break. One second corresponds to the link break happening just before a Hello message is scheduled to arrive. Two seconds corresponds to a link break just after reception of a Hello message.

Link layer feedback is not easily quantified by static parameters. In order to determine the latency of link layer feedback we performed an experiment. In the test we attempted to transmit packets to a destination that did not exist. By logging the time that each packet was passed to the wireless hardware and when each transmission error failure was received we determined the link layer feedback latency.

In this test, we utilized a IBM Pentium III 1 Ghz laptop with the HostAP driver [12] and a Netgear MA401RA IEEE 802.11b wireless card. In order to transmit packets to a destination that does not actually exists, we inserted static entries into the routing and ARP tables for the destination. Without these entries, the kernel would not send the packets to the wireless hardware. Then 100 packets at one second intervals were passed to the wireless card for transmission. The time when each packet exited the kernel and was buffered by the wireless hardware was logged. The program *iwevent* [11] was used to receive wireless link layer feedback events and the time when each transmission failure indicator arrived was recorded.

The time that elapsed between the packet reaching the wireless hardware and *iwevent* receiving notification of a packet transmission failure is presented in Table III. From our tests link layer feedback took on average 11 ms. In comparison to Hello messages, link layer feedback responds quickly, approximately one hundred times faster. This clearly motivates the addition of support for link layer feedback into current AODV implementations. Using link layer feedback will significantly improve performance by reducing packet losses due to mobility.

V. CONCLUSION

In this paper we analyzed design possibilities for AODV implementations. 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. We presented the design of many publicly available AODV implementations. Finally, we compared forwarding strategies and link break detection designs. 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, the additional features and performance 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] Nova Engineering, “NovaRoam,” http://www.novaroam.com/.
[2] C. E. Perkins and E. M. Royer, “The Ad hoc On-Demand Distance Vector Protocol,” in *Ad hoc Networking*, C. E. Perkins, Ed. Addison-Wesley, 2000, pp. 173–219.
[3] I. D. Chakeres and E. M. Belding-Royer, “AODV Routing Protocol Implementation Design,” in *Proceedings of the International Workshop on Wireless Ad hoc Networking (WWAN)*, Tokyo, Japan, March 2004.
[4] 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)*, Honolulu, Hawaii, October 2002, pp. 504–508.
[5] C. E. Perkins, E. M. Belding-Royer, and S. Das, “Ad hoc On-Demand Distance Vector (AODV) Routing,” *RFC 3561*, July 2003.
[6] IEEE Computer Society, “IEEE 802.11 Standard, IEEE Standard For Information Technology,” 1999.
[7] G. Bianchi, “Performance Analysis of the IEEE 802.11 Distributed Coordination Function,” *IEEE Journal Selected Areas in Communications*, vol. 18, March 2000.
[8] J. Kadlecsik, H. Welte, J. Morris, M. Boucher, and R. Russell, “The netfilter/iptables Project,” http://www.netfilter.org/.
[9] F. Lilieblad, O. Mattsson, P. Nylund, D. Ouchterlony, and A. Roxenhag, “Mad-hoc AODV Implementation and Documentation,” http://mad-hoc.flyinglinux.net.
[10] H. Lundgren, E. Nordstrm, and C. Tschudin, “Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks,” Uppsala University Department of Information Technology, Tech. Rep. 2002-022, June 2002.
[11] J. Tourrilhes, “Wireless Tools for Linux,” http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html.
[12] J. Malinen, “Host AP driver,” http://hostap.epitest.fi/.
[13] 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.
[14] I. D. Chakeres, “AODV-UCSB Implementation from University of California Santa Barbara,” http://moment.cs.ucsb.edu/AODV/aodv.html.
[15] L. Klein-Berndt, “Kernel AODV from National Institute of Standards and Technology (NIST),” http://w3.antd.nist.gov/wctg/aodv_kernel/.
[16] 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)*, San Francisco, CA, June 2003, pp. 99–112.
[17] E. M. Belding-Royer, “Report on the AODV Interop,” University of California Santa Barbara, Tech. Rep. 2002-18, June 2003.
[18] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, “CMC: A Pragmatic Approach to Model Checking Real Code,” in *Proceedings of the* 5^{th} *Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, December 2002, pp. 75–88.