System Services for Ad-Hoc Routing: Architecture, Implementation and Experiences

Vikas Kawadia[*]
ECE Dept and CSL, UIUC
kawadia@uiuc.edu

Yongguang Zhang
HRL Laboratories, LLC
ygz@hrl.com

Binita Gupta
Qualcomm Inc.
bgupta@qualcomm.com

Abstract

This work explores several system issues regarding the design and implementation of routing protocols for ad-hoc wireless networks. We examine the routing architecture in current operating systems and find it insufficient on several counts, especially for supporting on-demand or reactive routing protocols. Examples include lack of mechanisms for queuing outstanding packets awaiting route discovery and mechanisms for communicating route usage information from kernel to user-space. We propose an architecture and a generic API for any operating system to augment the current routing architecture. Implementing the API may normally require kernel modifications, but we provide an implementation for Linux using only the standard Linux 2.4 kernel facilities. The API is provided as a shared user-space library called the Ad-hoc Support Library (ASL), which uses a small loadable kernel module. To prove the viability of our framework, we provide a full-fledged implementation of the AODV protocol using ASL, and a design for the DSR protocol. Through this study, we also reinforce our belief that it is profoundly important to consider system issues in ad-hoc routing protocol design.

1 Introduction

Routing datagrams in a mobile ad-hoc network (MANET) is a difficult problem. Existing protocols to solve this problem include, but are not limited to AODV [25], DSR [20], DSDV [26] and TORA [24]. However most of the existing studies of these protocols are simulation based, with few real implementations. Validating MANET algorithms in real systems is necessary for their proliferation in the real world. But the sophisticated system-level programming so often required in the implementation of an ad-hoc routing protocol discourages MANET researchers from pursuing the much needed experimental studies.

We believe the core reason for having such difficulties in implementation is the lack of system support and programming abstractions in general purpose operating systems (such as Unix/Linux). As we will explain later in this paper, ad-hoc routing protocols often employ new routing models or have special requirements that are not directly supported by the current operating systems. Without proper systems support and convenient programming abstractions, implementors are forced to do low-level system programming, and often end up making unplanned changes to the system internals in order to gain the additional functionality required for ad-hoc routing. Not only is this a non-trivial task, but in practice it can also lead to unstable systems, incompatible changes (by different implementations), and undeployable solutions.

To address these issues, we develop the system support and programming abstractions needed to facilitate MANET protocol implementations and deployment. Our solution provides a set of system services that provide the necessary system support to meet the requirements of most ad-hoc routing protocols. The new programming abstractions also allow easy programming of ad-hoc routing protocols without the need for low-level system programming.

In this paper, we explore the difficulties encountered in implementing MANET routing protocols in real operating systems, and study the common requirements imposed by MANET routing on the underlying operating system services. Then, we propose a general modification of the current IP routing architecture, and a specific implementation of this architecture in Linux. Finally, we present our experiences in implementing several MANET routing protocols under this framework.

[*]Part of this work was performed at HRL Laboratores, LLC when the author was a summer intern.

2 Challenges in Mobile Ad-Hoc Routing

2.1 Current Routing Architecture

The current Internetworking architecture segregates the routing functionality into two parts: *packet forwarding* and *packet routing* (see Section 4.2 of Peterson & Davie's "Computer Networks" [28] for a good discussion on this topic). Packet forwarding refers to the process of taking a packet, consulting a table (the forwarding table), and sending the packet towards its destination as determined by that table. Packet routing, on the other hand, refers to the process of building the forwarding table. Forwarding is a well-defined process performed locally at each node, whereas routing involves a complex distributed decision making process commonly referred to as the routing algorithm or the routing protocol. The forwarding table contains enough information to accomplish the forwarding function, whereas the routing table contains information used by the routing algorithm to discover and maintain routes. Strictly speaking, these two tables are different data structures but the two terms are often used interchangeably.

In modern operating systems, packet forwarding is implemented inside the OS kernel whereas routing is implemented in the user space as a daemon program (the routing daemon). Figure 1 illustrates the general routing architecture. The forwarding table is inside the kernel and is often called the kernel routing table or route table. Whenever the kernel receives a packet, it consults this table, and forwards the packet to the "next-hop" neighbor through the corresponding network "interface". The kernel routing table is populated by the routing daemon according to the routing algorithm it implements.

There are numerous reasons for separating forwarding and routing [28], and placing packet forwarding inside the kernel and packet routing in user-space. Packet forwarding must make decisions for every packet and therefore should be efficient. It should reside inside the kernel so that a packet can traverse this node as fast as possible. On the other hand, packet routing involves complex and CPU/memory intensive tasks, which are properly situated outside the kernel. This principle of separation has made routing in modern operating systems efficient and flexible. It allows the routing function to continue evolving and improving without changing the OS kernel.

2.2 Challenges in On-demand Routing

It is desirable to fit mobile ad-hoc routing into the above architecture. Most ad-hoc routing protocols can be classified into two categories: proactive and reactive protocols. Pro-active (or table-driven) routing protocols maintain routes to all possible destinations by periodically exchanging control messages. Reactive (or on-demand) protocols, on the other hand, discover routes only when there is a demand for it. Proactive protocols (such as DSDV [26]) can be easily implemented as user-space routing daemons in the current routing architecture, in much the same way as routing protocols of the wired world (such as RIP, OSPF, or BGP). However, problems arise with reactive or on-demand routing protocols, such as AODV [25] and DSR [15]. We now describe these challenges in detail.

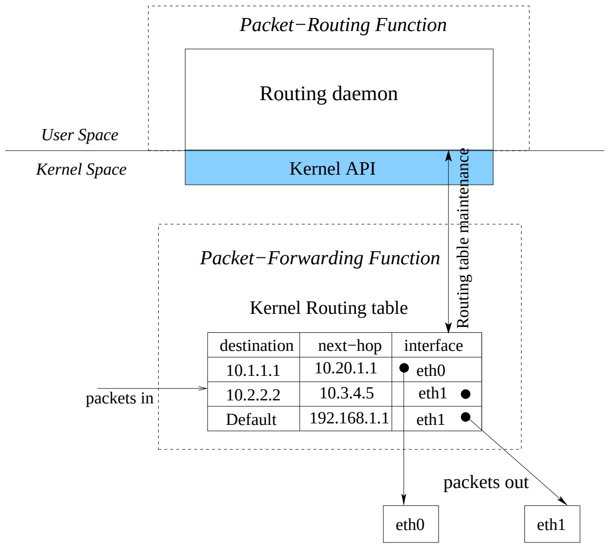


Figure 1. Current Routing Architecture

Challenge 1 *Handling Outstanding Packets*

Normally, each packet traversing the packet forwarding function will be matched against the kernel routing table. If no entry matches its destination, the kernel will drop the packet immediately. However, this is not a desirable behavior for on-demand ad-hoc routing. In on-demand ad-hoc routing, not all routes will exist *apriori*; some must be "discovered" when needed [15]. In such cases, the correct behavior should be: to notify the ad-hoc routing daemon of a route request, and to withhold the packet until the discovery finishes and route table updated. Unfortunately, there is no mechanism in modern operating systems to support this new packet forwarding behavior, and there is insufficient kernel support to implement tasks like queuing of all outstanding packets. Therefore, the operating system should implement the following functions for on-demand ad-hoc routing:

1. Identify the need for a route request.

2. Notify ad-hoc routing daemon of a route request.

3. Queue outstanding packets waiting for route discovery.

4. Re-inject them after successful route discovery.

Challenge 2 *Updating the Route Cache*

On-demand routing protocols typically maintain a cache of recently used routes in user space to optimize the route discovery overhead. Each entry in this route cache has an expiration timer, which needs to be reset when the corresponding route is used. The entry should be deleted (both from the user-space route cache and the kernel routing table) when the timer for that entry expires. Therefore, when an entry in the kernel routing table remains unused (i.e., has not been looked up) for a predefined time period, this information should be accessible to the the user-space routing daemon. This is difficult to achieve under the current routing architecture, because no record of route usage in the kernel is available to user-space programs.

Challenge 3 *Intermixing Forwarding and Routing Functions*

Certain ad-hoc routing protocols do not have a clean separation between the forwarding and routing functions in their design. Many of these protocols (notably DSR [15]) are based on the "on-demand behavior", where actions are taken only on reaction to data packets [21]. Since there is no periodic activities such as router advertisements, link/neighbor status sensing, or even the timely expirations of unused routing table or cache entries, routing activities must be made part of the forwarding activities. This protocol design presents a big implementation challenge to fit in the current routing architecture. There are two basic implementation approaches. The in-kernel approach typically requires the bulk of their routing logic to be implemented inside the kernel, making it difficult to program and to modify. On the other hand, the user-space approach requires the forwarding action to take place in user space, forcing every packet into user space.

In some cases, the principle of separation is violated for subtle optimizations aimed at reducing routing overhead. Such optimizations are usually simple to implement in a simulation environment, but present significant system design challenges. In the course of this study, we have found such examples in protocol design. We will present them in the later sections when we describe our experiences in implementing them.

Challenge 4 *New Routing Models*

Some ad-hoc routing protocols adopt unconventional routing models such as source routing ([15]), flow-based forwarding ([13]), etc. These new routing models deviate from the current IP routing architecture and present new challenges in system design. For example, in source routing the entire path that a packet should traverse is determined by the origin of the packet and encoded in the packet header, whereas in traditional IP routing the forwarding decision is made hop-by-hop and driven by the local routing tables. In flow-based forwarding each packet carries a flow id, and each node in the network has a flow table, which maintains the next-hop address and other information for each flow id. The forwarding is driven by table lookup on the flow id, and routing is the process to establish the flow table in each node.

Most general purpose operating systems are not flexible enough to provide a blanket support for all these and other new routing models. Implementation of these routing protocols will either modify the kernel IP stack to incorporate new routing architecture, or use kernel extension mechanisms (such as loadable modules) to bypass the IP stack.

Challenge 5 *Cross-Layer Interactions*

The wireless channel presents a lot of opportunities for optimizations through cross-layer interactions. For example, some ad-hoc routing protocols use physical and link layer parameters like signal strength, link status sensing, and link layer supported neighbor discovery, etc., in their routing algorithm. The problem of dealing with cross-layer interactions is a more difficult problem, both at the conceptual level and the implementation level. At the conceptual level, substantial work is needed in laying down guidelines for systematizing cross-layer interactions. This is necessary, since even though cross-layer design may provide optimizations, an indiscriminate access to all lower layer parameters would seriously damage the neat architecture which lies at the foundation of all networking.

At the implementation level, such cross-layer interactions obviously depend on the hardware capabilities and whether the hardware allows access to that information. It may be possible to provide access to lower layer parameters for some particular hardware, but a general solution for all hardware would require some standardization across the plethora of wireless interface cards and drivers available.

We believe that Challenges 1 and 2 can be met with enhanced system services in the operating systems. However, Challenge 3 and 4 may have to be dealt with on a case by case basis for every protocol. In this work, we first develop a general architecture to meet the first two challenges. And we will illustrate how we deal with the remaining challenges through our implementations of some ad-hoc routing protocols. Cross-layer interactions will also be the subject of future investigations.

3 New Architecture and API

We first develop a general solution to support on-demand routing in general purpose operating systems. The purpose is to suggest modifications to these operating systems so that ad-hoc routing can be easily supported in the future. We propose enhancements to the current packet-forwarding function with the following mechanisms.

An additional flag should be added to each kernel routing table entry to denote whether it is an *on-demand* entry, defined as those entries for which the kernel is prepared to queue packets in case of route unavailability. An on-demand route entry is said to be *deferred* if it has empty next-hop or interface fields, meaning that the route is yet to be discovered. Instead of getting dropped in the normal packet forwarding path, packets matching a deferred route will be processed as described in Challenge 1. We note that it is not necessary to include every possible on-demand destination in the routing table. Flagging a subnet-based route or the default route as on-demand can serve the same purpose.

A new component, called the on-demand routing component (ODRC), should be added to complement the kernel packet-forwarding function and implement the desired on-demand routing functionalities. When it receives a packet for a deferred route, it first notifies the user-space ad-hoc routing daemon of a *route request* for the packet's destination. Then, it stores the packet in a temporary buffer and waits for the ad-hoc routing daemon to return with the route discovery status. Once this process finishes and the corresponding kernel routing table entry is populated, the stored packets are removed from the temporary buffer and re-injected into packet forwarding.

To address Challenge 2, a timestamp field needs to be added to each route entry to record the last time this entry was used in packet forwarding. This timestamp can be used to retire a stale route that has not been used for a long time.

Finally, we provide a programming abstraction (API) so that these new mechanisms can be conveniently used in an ad-hoc routing daemon program. The API should contain the following functions:

-
  ```
  int route_add(addr_t dest,
          addr_t next_hop, char *dev);
  int route_del(addr_t dest);
  ```

  These basic routines add or delete an on-demand entry from kernel routing table. To add a deferred route entry, specify `next_hop` to be 0. (Here, `addr_t` is a generic type for the network address, such as `unsigned long` for IPv4 address.)

-
  ```
  int open_route_request();
  int read_route_request(int fd,
           struct route_info *r_info);
  ```

  ODRC notifies the ad-hoc routing daemon about the route requests in the form of an asynchronous stream. The first function returns a file descriptor for this stream and the second function fills in information about the route requests in the second argument, `struct route_info*` which is defined as follows :

  ```
  struct route_info {
          addr_t dest;
          addr_t src;
          u_int8_t protocol;
  };
  ```

  This structure contains information about the packet that triggers the route request, which can be useful for some routing daemons. For example a different action may be warranted if the packet was generated locally rather than being forwarded for some other host (which can be deduced from the `src` field). Similarly, some routing protocols may need to know if the route request is for a TCP packet or a UDP packet.

  The file descriptor semantics allows the ad-hoc routing daemon to use either event-driven or polling strategy. The function `read_route_request()` blocks until the next route request becomes available.

-
  ```
  int route_discovery_done(addr_t dest,
             int result);
  ```

  This function informs the ODRC that a route discovery for the given destination has finished and the kernel routing table populated. The result field indicates whether the route discovery was successful or not.

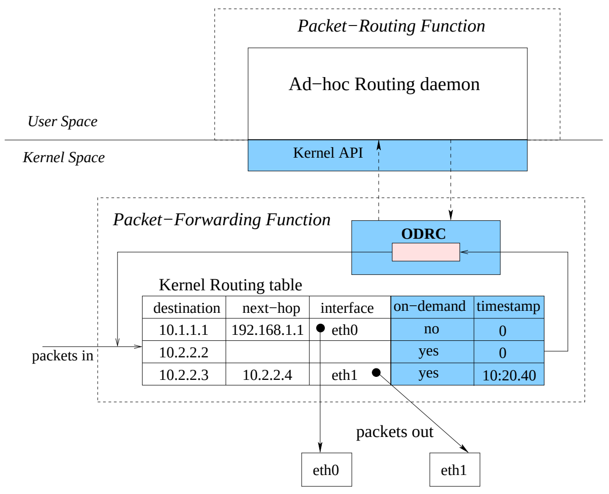
-
  ```
  int query_route_idle_time(addr_t dest);
  ```

  Given a destination, this function returns the idle time recorded in the kernel routing table for this entry (elapsed time since the last use of the route).

-
  ```
  int close_route_request(int fd);
  ```

  This function is called by the routing daemon when it no more desires to receive any more route requests. This enables the ODRC to free the memory used up by packets already queued up and close the fd.

Figure 2 illustrates our new architecture and its components. The shaded parts are our proposed additions.



destination	next−hop	interface	on−demand	timestamp
10.1.1.1	192.168.1.1	eth0	no	0
10.2.2.2			yes	0
10.2.2.3	10.2.2.4	eth1	yes	10:20.40

Figure 2. New Routing Architecture

The API we have provided takes care of Challenges 1 and 2. However, it is not yet complete as Challenges 3 and 4 have not yet been completely addressed.

Implementing this API in a Unix-like modern operating system usually requires some changes to the system internals. The ideal way is to integrate the above mechanisms into the kernel IP stack. This would involve implementing queuing for every deferred route and adding a special file descriptor for route-request stream. Depending on the operating system's kernel facilities and extensibility, this architecture can also be implemented as kernel modules, or largely in user-space.

It is debatable whether the ODRC functionality should be implemented outside the kernel and whether it is better to queue all deferred-routing packets in user-space. The advantage of a user-space approach is that it reduces the kernel complexity and memory usage. If the routing protocol requires prolonged route discovery procedure, it will be compelling to buffer packets outside the kernel. The disadvantage is the need to copy every deferred-routing packet (i.e., packets awaiting route discovery) from kernel to user-space and to re-inject them back to kernel when the routes are ready, but it can be argued that such overhead is insignificant compared with the time and overhead in an average route discovery.

4 Implementation in Linux: Ad-hoc Support Library

Our long-term objective is to implement this solution in common operating systems and make it standard in future versions. However, our immediate goal is to make it available to the current Linux 2.4 users because getting changes accepted into a standard operating system is a tedious process. We would like to find a way to provide the services described in Section 3 without being intrusive. This strategy certainly has practical value as few users are willing to modify their operating system kernels. To do this efficiently needs a careful design, which we describe in this section.

Linux provides several mechanisms for extending the kernel functionalities. These include loadable modules, where a new kernel function can be inserted into a running kernel without recompiling or rebooting, and a packet filtering and mangling facility called Netfilter [5]. In particular, Netfilter provides a set of hooks in the kernel networking stack where kernel modules can register callback functions, and allows them to mangle each packet traversing the corresponding hooks. We use these two mechanisms to implement our system services and make it our goal not to change the kernel source code.

4.1 Design and Mechanisms

We place the ODRC function in user-space to reduce the kernel complexity and memory requirements. There are two possible ways to implement this. One approach is to put ODRC in a shared library and link any routing daemon that wishes to use this ODRC function with this library. Another approach is to put ODRC in a separate daemon program and let it communicate with the routing daemon using some inter-process communication mechanism like sockets. Both approaches have their pros and cons. The library approach is more efficient because it does not have the overhead of inter-process communication, but any bug in the library is likely to crash the routing daemon also. The library approach gives a more natural picture of the ODRC functionalities as system services, i.e., the API is available as direct function calls once the appropriate header files are included.

We thus implement ODRC as a user-space library. We call it the Ad-hoc Support Library (ASL) or libASL. ASL implements the API we described in Section 3. The implementation consists of two main components: the first component is completely in user-space and implements the common functionalities which are needed by most on-demand routing daemons; the other part is specific to particular routing protocols and is implemented as loadable kernel modules. For example, for the AODV protocol there is the aodv-helper kernel module which provides additional API for some subtle optimizations prescribed by the AODV draft. The dsr-helper module is more complicated to accommodate for DSR's sophisticated features. We also provide a generic helper module called the route-check, which provides a simple solution to the route caching problem. This architecture consisting of libASL and helper modules provides ASL with the flexibility to incorporate future routing protocols or modifications to current ones in their respective helper modules.

4.1.1 Handling Outstanding Packets

To solve the problem of identifying the need for a route request, we need to filter all packets for which there exists no route. Without modifying the routing table structure, there is no simple way to do that in kernel. We solve this with an unused local tunnel device called Universal TUN/TAP (`tun`) as the "interface" device for these destinations. To catch packets for all such destinations, we can use the default route which is used for packets which do not match any other entry in the routing table. The default route can be setup like this :

```
ifconfig tun 127.0.0.2 netmask 255.255.255.255 \
         broadcast 127.0.0.2 up
route add default dev tun
```

TUN/TAP is a virtual tunnel device that makes available all received packets to a user-space program through the /dev/net/tun device. In our implementation, this device is opened by a call to `open_route_request()`, hence it receives all packets that kernel writes to `tun`, i.e., all packets for which there is no route. This also solves the problem of passing and storing packets in user-space.

Whenever a new packet is read from the virtual device, `/dev/net/tun`, the ad-hoc routing daemon which has opened a route request gets notified on that fd. It can read the details of the route request through `read_route_request()`, and then can initiate route discovery for the requested destinations. These packets are temporarily queued in a hash table keyed by the destination IP address. This functionality is implemented in the Ad-hoc Support Library. Since the buffer is in user-space, a large buffer is available to queue packets. This means that packets would not be lost even if the route discovery delays are large.

The next issue is to re-inject packets back into the IP stack after a successful route discovery. The mechanism we use is a raw IP socket[1]. A packet sent through a raw socket is inserted as is (bypassing any IP and header processing) to the kernel output chain just before the packet-forwarding function. Here, we use a raw socket to send the queued packets out. These packets are appropriately routed in the kernel using the newly discovered routes.

A natural question to ask is that why don't we re-inject the packets back into the IP stack by writing it on the user end of the same virtual interface used earlier, i.e., /dev/net/tun. To the kernel it appears as if a packet has been received on the tun virtual interface, and it can do the routing as if it were a normal incoming packet. This approach works fine for packets which a node forwards, but unfortunately does not work for packets generated locally by the node. Packets which are generated locally already pass through the IP output routines, and when re-injected through tun appear on the forwarding chain. The forwarding chain does not allow packets in which the source IP address matches the local IP address, since this is an indication that the node's IP address is being spoofed by somebody else. Hence, we have to resort to raw IP sockets as described above.

[1] Raw sockets are normally used to handle packets that the kernel does not support explicitly. The `ping` program, for example, uses raw sockets to generate ICMP packets.

4.1.2 Updating the Route Cache

Now we come to Challenge 2, to refresh entries in the user-space route cache when a route is used in the kernel. Since we are not making changes to the kernel routing table, the only way is to maintain a separate timestamp table for each entry in the routing table. We thus design a simple kernel module called `route_check` to maintain this table and register it at Netfilter's `POST_ROUTING` hook (after routing table lookup and before entering the physical network interface). This means that every outgoing packet will pass through this module. It simply peeks at the packet header and updates the corresponding timestamp value. This timestamp information is made available to user-space programs using an entry in the `/proc` file system. The `query_route_idle_time()` function exposed by the ASL API reads this file (`/proc/asl/route_check`) to determine the idle time for a destination. The routing daemon can check the freshness of a route by reading this file, and delete the stale routes from the kernel routing table accordingly. The route_check module is a generic helper module, which is available to all the routing daemons.

Actually, the current Linux kernel does maintain a cache of most frequently used routes to make routing lookups efficient. When a route is first used it is looked up from the Forwarding Information Base (FIB) which is a complex data structure maintaining all the routes. After first use this entry is inserted in the route cache for fast lookup. It expires from the cache if not used for some length of time. Information about this route cache is exposed through the files `/proc/net/rt_cache` and `/proc/net/rt_cache_stat`. Unfortunately these files do not include information about the `last_use_time` of the entries. It is a very simple modification to the Linux kernel to make it output this information, but since we are not making any changes at all in the core kernel source as it would require kernel recompilation, we have adopted the `route_check` module approach just described. We emphasize that a very small change in the Linux kernel would make the route_check

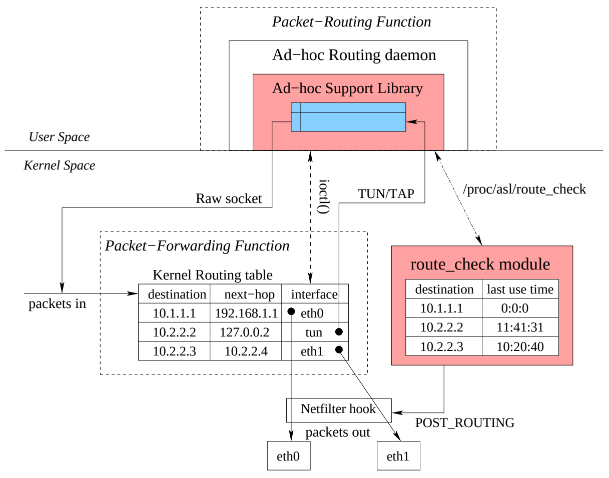


Figure 3. ASL software architecture

module unnecessary.

4.2 ASL Implementation Details

Figure 3 illustrates the structure of this implementation. The two main components are the user-space library ASL and the kernel module `route_check`. The library implements the API described in Section 3. We now describe how we implement these functions in our library.

`route_add()` and `route_del()` functions add or delete routes to the kernel using the `ioctl()` interface. When the user indicates that the route be a deferred route by specifying an empty next-hop, the device for the route is made to be `tun`. `open_route_request()` initializes the tun device, the raw socket, the data structures to queue deferred packets, and also inserts the `route_check` module in the kernel. The data structure to store the packets is a hash table of queues, keyed by the destination IP address. The function `open_route_request()` returns the descriptor of the tun device which can be monitored using a polling or event driven strategy. `read_route_request()` blocks reading from this tun device. When a packet is received on tun, this functions stores the packet and delivers information about the packet in the form of `struct route_info`. Based on this the routing daemon initiates route discovery, and calls the `route_discovery_done()` function on completion of this process. If the route discovery was successful then this function retrieves the packets for that destination from the storage and sends them out on the raw socket. If a route could not be found then the packets are thrown away and the memory used for them is freed. `query_route_idle_time()` reads the `last_use_time` for that destination from `/proc/asl/route_check` and returns the idle time. This needs to be called whenever the routing daemon has to make a decision to expire routes from its user-space route cache. The function `close_route_request()` simply shuts down all the sockets, frees all the memory for storing the packets, and removes the `route_check` module from the kernel.

Below we give the pseudo code of an example routing daemon which uses this library.

```
aslfd = open_route_request()
route_add(default,0) /* add deferred route */
loop  /* this could be select or poll */
  wait for input from {aslfd or other fd’s}
  if input from aslfd
    dest = read_route_request()
    if(route request is new)
      do route discovery for dest
      if successful
        add route for dest to kernel
        route_discovery_done(success)
      else
        route_discovery_done(failure)
      end
    else
      continue
    end
  end
  if input from other fd’s
    process according to protocol semantics
    /*call before expiring routes*/
    query_route_idle_time()
  end
end
close_route_request()
```

5 Implementing Routing Protocols: Experiences using ASL

To evaluate the utility of the Ad-hoc Support Library, we set out implementing the various routing protocols that have been proposed. We provide a full-fledged implementation of the AODV protocol. We also provide implementation design guidelines for some other protocols.

5.1 Ad-hoc On-demand Distance Vector Routing (AODV)

Our implementation of AODV is a user-space routing daemon which uses the Ad-hoc Support Library for system services. It follows a modular architecture in C++ to provide a clean and extensible implementation. The current implementation supports all the features of AODV draft version 10 [27]. In the following sections we describe both the user-space design as well as the special system support (in addition to standard ASL) required for AODV, which we implemented as the aodv-helper kernel module.

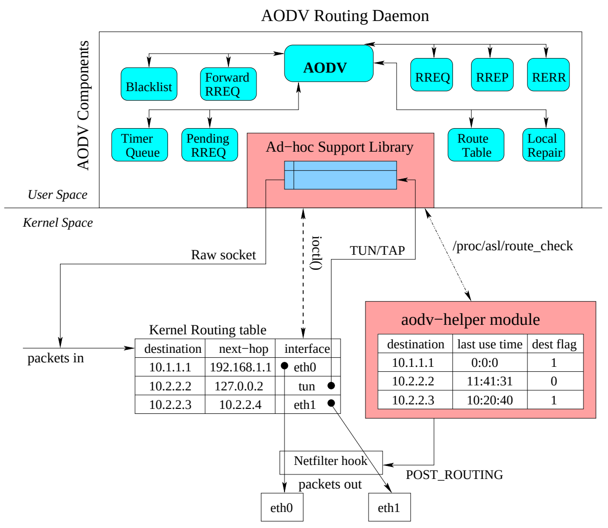


Figure 4. Software architecture of the AODV-UIUC routing daemon.

5.1.1 AODV Components

This section talks about the different components of our AODV routing daemon, their functionalities and the interactions among these components to implement various features of the AODV protocol. This is illustrated in Figure 4. Details of this implementation, called AODV-UIUC, are provided in [11].

The component called AODV defines the main flow of control inside the AODV routing daemon. The control flow is based on an event-driven design. The set of possible events include reception of routing control packets, expiration of various timers, and reception of route requests on the ASL socket. Possible actions include sending out packets, setting new timers and updating various data structures. The daemon program is essentially a big select() loop which monitors various file descriptors for the events and takes the appropriate actions. This component also initializes ASL by calling the functions `int route_add()` and `open\_route\_request()`.

The RREQ, RREP and RERR components take care of both generating as well as processing incoming route requests, route replies and route error packets respectively. The Routing Table component (routeTable) handles updates to the aodv routing table as well as to the kernel routing table. It also maintains a route cache using the aodv-helper module through the corresponding API function `query_route_idle_time_aodv()`, as explained in the next subsection. The Pending Route Request component (rreqPendingList) implements the expanding ring search and RREQ retransmission features of the AODV routing protocol. The Forward Route Request component ensures that a node does not process a particular RREQ packet multiple times, by storing a list of recently seen RREQ packets. The Local Repair component attempts to repair links locally and the BlackList component takes care of routing in the presence of unidirectional links. Finally, the TimerQueue component maintains various AODV timers including reboot timer, periodic refresh timer, hello timer and rreq retransmission timer.

5.1.2 ASL and Aodv-helper

Using ASL makes efficient on-demand routing possible in our AODV implementation. The generic route-check module can be used for maintaining the user-space route cache. However, the AODV protocol requires that whenever a packet is forwarded to any destination by a node using a particular route, the node should update the lifetime values (in its route-cache) associated with the destination, the previous hop and the next hop nodes on that route. Previous hop is defined as the next-hop along the reverse path back to the source. Updating the previous hop node, when a route is used was not possible using the generic route-check module, since the information about the previous hop is not available in the packet but only in the routing table . We had to redesign our data structures and the query process for updating the lifetime of the previous hop for a route. These substantially more complicated new data structures and query process were made part of the aodv-helper module.

Like the generic route_check module, the aodv-helper module also registers at the Netfilter's POST_ROUTING hook and peeks at the every outgoing packet to log in the timestamp information. But, unlike the route-check module, the aodv-helper module also logs an additional flag parameter called the destination flag with every entry in the /proc file. A value of 1 for this flag signifies that the entry correspond to the destination of the packet, and a value of 0 implies that the entry corresponds to the source of the packet. Thus, the aodv-helper module logs two entries for every outgoing packet, one for the destination and one for the source.

The `query_route_idle_time()` API function interface has also been modified as follows :

```
int query_route_idle_time_aodv(addr_t k,
                  int destination_flag);
```

Given a destination k, this function returns the idle time recorded in the kernel routing table for this entry (elapsed time since the last use of the route). The destination_flag is used to differentiate between the return value. A value of 1, for the flag implies that the query is for the idle time since a packet was last forwarded to this destination, whereas a value of 0 denotes a query for the idle time since a packet was last received from this source. The aodv daemon, uses this API as follows:

1. Whenever the routing table entry for a destination d expires, the AODV routing daemon queries the aodv-helper module for the idle time for that destination with the following API function call (note the destination flag passed is 1): `query_route_idle_time_aodv(d,1)`.

2. If the destination d is just one hop away, then the routing daemon goes through the entire routing table looking for the nodes for which this destination acts as the next hop. It then queries aodv-helper module for the idle time corresponding to each of these nodes (with the destination flag set to both 1 and 0) and chooses the minimum idle time of all such idle times. The idle time for previous hops can be determined by a call to `query_route_idle_time()` API function with the destination flag set to 0, when the timer for such an entry expires. This ensures compliance with all the features of the draft.

5.1.3 Experiences

We found that it was pretty straightforward to implement AODV as a user-space daemon, once all the kernel interaction issues were taken care of by the Ad-hoc Support Library. The user-space daemon is about 5000 lines of C++ code and the aodv-helper module is about 800 lines. ASL itself is about 2500 lines of C code. Using ASL, AODV development was easier as all the debugging was confined to user-space. We have tested our implementation (See [11]) on a testbed of about 10 laptops.

We also realized that some of the subtle optimizations which needed hard work were probably not very important. For example, after implementing the aodv-helper, we realized that updating the previous hop and next hop lifetimes, when a route is used, is probably completely redundant. This is because the draft also says that nodes should maintain connectivity information with all one-hop neighbors, using either explicit hello messages, link layer notification mechanism or passive acknowledgments, and update the lifetime field for all the one-hop neighbors if the link is determined to be up. Since the set of possible previous hops and next hops is a subset of the set of one-hop neighbors, the lifetime for all such hops is automatically updated by the neighbor detection mechanism. Thus, updating next-hop and prev-hop lifetimes during route caching is probably unnecessary, as it comes into play only on those very rare occasions when a few consecutive hello packets from a neighbor were lost, but that neighbor was somehow still used as a previous/next hop on some route. Thus, if we ignore this redundant feature in the draft, aodv-helper is no more needed and the generic route-check module will be sufficient.

5.2 Dynamic Source Routing (DSR)

Our second attempt is to implement DSR within the ASL framework. We choose DSR because it is another popular and maturing ad-hoc routing protocol with significant research backing, and is also architecturally, a different protocol from AODV. The implementation starts with the DSR Internet Draft [14].

5.2.1 Difficulties

Implementing DSR within ASL framework is a big challenge. First, it is a source routing protocol and has its own protocol format to specify source routes (Challenge 4). Second, it is based entirely on on-demand behavior and does not have a separable routing and forwarding function (Challenge 3). Neither of these features fits directly in the ASL architecture.

In particular, we consider the following issues:

- *Interfacing with the kernel's IP stack.* DSR specifies its own protocol header, which is immediately after the IP header and before any IP payload. This implies that the naturally appropriate place for inserting the DSR implementation will be in the IP stack at the IP multiplex/demultiplexing point. There are other alternatives, such as intercepting every DSR packets at the packet input/output chains and bypassing the kernel IP stack, or processing the DSR packets in a virtual device driver below IP.

- *Processing every packet.* Every data packet carries the source route in the DSR header. As part of the packet forwarding process, this header information needs to be looked up and modified at each intermediate node. Further, the source route is also used to update the forwarding node's route cache. Route shortening in forms of gratuitous route reply may be applied if the forwarding note has a better route to the destination in its route cache. In addition, DSR control information (e.g., gratuitous route error messages) may be piggybacked on any packet, to reduce routing overhead. Therefore, all packets require significant processing. This makes fast packet forwarding difficult.

- *Maintaining routes* In the absence of periodic neighbor/link sensing, DSR relies on data packets to detect broken links. It requires every packet forwarded by a node to be acknowledged by the next hop, either through the possible built-in link-layer acknowledgment mechanism (such as 802.11 MAC), or by passive acknowledgment where the sending node overhears the next hop further forwarding the packet, or by explicit network layer acknowledgment from the next-hop back to the sender. This requires each node to keep a copy of all forwarded packets for a short period of time until being acknowledged. Packets unacknowledged after timeout period will trigger route error messages, and optionally salvaging actions where a forwarding node rewrites the packet's source route option to choose a different path. Route maintenance further complicates the system design.

- *Listening in the promiscuous mode.* DSR allows the optional use of promiscuous mode listening for performance improvement. Promiscuous listening is defined as the process by which the network card can overhear packets not intended for its hardware address, and deliver it to the network stack. Using this feature, a node can overhear a data packet and can add the source route to its route cache. Promiscuous mode requires support from hardware and device driver, but not all wireless devices supports this type of operation for security considerations.

5.2.2 A Split Design

Our goal, in accordance with the philosophy of this work, is to do a reasonably simple and maintainable implementation of DSR in Linux, with minimum modifications to the kernel source. As we have explained earlier, due to the inseparable forwarding and routing functions, there are usually two ways to implement such protocols: a complete in-kernel approach (such as in [19]), and a complete user-space approach (such as in [10]). Both approach have pros and cons. A complete user-space approach will be inefficient for the forwarding function, but an in-kernel approach is different to maintain, different to modify, and different to port to other operating systems.

In our implementation, we attempt a split-system approach. The idea is to segregate the forwarding and routing functions to some extent, even though they are intermixed in the protocol design (Challenge 3). We believe that the core of the source-routing based forwarding activities, i.e., to send a data packet to the next-hop based on its DSR header, should be as efficient as possible and reside inside the kernel. We call this the *source forwarding* function. The majority of other source routing activities, which are induced by source forwarding, need to be flexible and can reside in user-space.

Figure 5 illustrates the overall design of this DSR implementation; the shaded parts indicate the various DSR components. It consists of a user-space DSR Routing Daemon and two kernel modules: `DSR-forwarding-helper` and `DSR-maintenance-helper`. The user-space daemon performs majority of the DSR routing functions, including route discovery and route maintenance. It relies on ASL to manage on-demand route

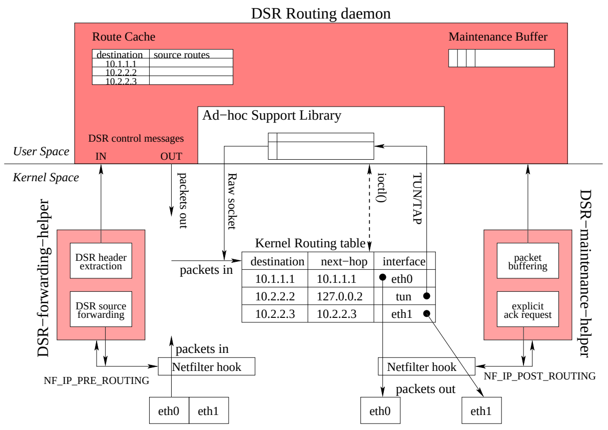


Figure 5. Design of DSR Routing Daemon

requests, and relies on the two kernel modules to interact with the forwarding function.

The `DSR-forwarding-helper` module handles all incoming DSR packets from the network devices. If it receives a DSR packet with a source route option, it executes the forwarding function, which involves making changes to the IP and the DSR headers. At the same time, it also extracts necessary DSR header information into a buffer. Later, this header information is passed on upward to the user-space route daemon and processed in the background, after the in-kernel forwarding is done. If the DSR packet is meant for this node, it is demultiplexed here (with the DSR header removed). Or, if it is a Route Request packet, it is sent upward to the DSR route daemon for source routing functions.

The `DSR-maintenance-helper` module inspects all outgoing DSR packets before sending to the network devices. Its only purpose is for route maintenance. It makes a copy of every outgoing packet and sends them upward to DSR route daemon for temporary buffering (in DSR Maintenance Buffer). Optionally, if the DSR route daemon determines that an explicit acknowledgment should be used, this module can insert a DSR Acknowledgment Request option in the DSR header of selected packets. Other than this, the route maintenance is almost entirely handled in user-space. The reason for this design is the following. We believe that route maintenance is not part of the core source forwarding function and should not stand in the way inside kernel. Once the packets are sent and copies are made, the route maintenance can work "in the background". When the acknowledgment comes back in the form of a DSR Acknowledgment option, the `DSR-forwarding-helper` module will forward it to the DSR route daemon, where the matching is done. If an entry in the Maintenance Buffer times out, the route daemon can update the Route Cache and generates Route

Error messages.

Under this design, a DSR data packet can pass through the kernel quickly without being delayed by non-critical DSR activities. Majority of other DSR activities are performed in user-space without getting in the way of the kernel source forwarding path. The kernel routing table will only contain entries for its neighbors. All other nodes will be marked as deferred (i.e., use `tun0`) so ASL can catch all the outstanding packets. ASL's `route_check` module is not used because DSR does not require periodic deletion of unused route cache entries.

5.2.3 Implementation

Following the same philosophy of the ASL work, our DSR implementation uses the standard Linux 2.4 kernel facilities only. All the kernel additions are implemented in two loadable kernel modules. No kernel recompilation is required.

We use the Netfilter facility extensively. The `DSR-forwarding-helper` module attaches itself to the Netfilter `NF_IP_PRE_ROUTING` hook to capture all incoming DSR packets from the network devices. The `DSR-maintenance helper` module attaches itself to the Netfilter `NF_IP_POST_ROUTING` hook to capture all outgoing DSR packets before passing to the network devices.

Our experience shows that an intermixing routing/forwarding protocol like DSR is more difficult to implement in a modern operating system, even with the help from the ASL framework. To achieve both efficiency and portability in the system design, we have to excise the routing/forwarding functional separation to some extent. Compared with the prior approaches (either all-in-kernel as in [19] or all-in-user-space as in [10]), our split design is better than the all-in-user-space approach because we now copy only the DSR header information, not the entire packet, to the user-space. The forwarding is entirely in kernel, while this header information can be processed later in the background, after the forwarding is done. This ensures a high performance forwarding function for DSR. Further, this design is better than the all-in-kernel approach, because kernel now process only the most critical function, not rest of the tedious on-demand behavior logics. The user-space implementation of the non-critical functions ensure that it is portable, maintainable, and extensible.

5.3 Other On-demand Routing Protocols

Temporally ordered routing algorithm (TORA) [24] is an adaptive, distributed routing algorithm based on the concept of link reversal. It strives to minimize communication overhead due to network topological changes. The TORA protocol specifications [23] are very amiable to the framework we have developed in this work. The algorithmic details can be implemented in user-space as the the TORA daemon, which uses the Ad-hoc Support Library for the on-demand mode of operation. For route caching, the generic route-check module seems sufficient.

Associativity based routing [30], is an on-demand, distance-vector routing protocol in which the metric is link-stability instead of the traditional hop-count based metric as in AODV. The link-stability is determined by associativity ticks which is essentially a count of beacons received from the neighbors. Since ABR specifies lots of features [31], which depend critically on the granularity of the associativity metric, it assumes that the data link layer is capable of getting a reactive estimate of the associativity metric efficiently. If the network card or the driver does not support this, ABR is not practicable for those devices. If such support is available, then ABR is quite cleanly implementable using our framework.

5.4 Pro-active Routing Protocols

Pro-active routing protocols can be easily implemented with the current routing architecture in all operating systems. They do not need the framework which we present. The DSDV (Destination Sequenced Distance Vector) routing protocol and the Adaptive DSDV protocol have been implemented in [11]. Another pro-active protocol called VDBP (Virtual Dynamic Backbone Routing) [18] has also been implemented for Linux. Optimized Link State Routing (OLSR) and Topology Broadcast with Reverse Path Forwarding (TBRPF) are two other popular proactive routing protocols which have also been successfully implemented in Linux and FreeBSD respectively.

Hybrid routing protocols use a combination of pro-active and reactive routing schemes. For example, the Zone Routing Protocol (ZRP [12]) divides the network dynamically into zones, and uses a pro-active protocol for intra-zone routing whereas a reactive protocol for inter-zone routing. From a systems viewpoint, ZRP does not present any new challenges compared to AODV; ASL can be used for efficiently implementing the inter-zone routing part of ZRP.

6 Existing Implementations and Related Work

There have been several implementations of some on-demand ad-hoc routing protocols. These implementations address some or all of the on-demand routing problems, but very few attempt to provide a general framework as we do. In this section, we provide a comparison on how these implementations attempt to address the problems we described in Section 2, and suggest how our approach can help improving them.

An implementation study of AODV routing protocol [29] raises issues similar to what we have discussed here. To address on-demand routing problems for AODV, it suggests significant modifications to the existing kernel code. First, IP layer builds a short lived dummy routing table entry for every unroutable destination. It then uses netlink socket to inform AODV routing daemon about the need to initiate a route discovery. Data packets are buffered inside the kernel in a simple linked list referenced from the dummy routing table entry. IP is also modified to add a Last Use field for every route, which is used by the routing daemon when deleting the routes. Our independent investigations led to the identification of similar issues and development of the API we presented in Section 3. However instead of modifying the Linux kernel, we focused on providing a user-space implementation in the form of a shared library, which we hope will be of more immediate use in implementing adhoc routing protocols.

Madhoc is a user-space implementation of AODV [25]. To address the on-demand routing problem, it snoops ARP (Address Resolution Protocol) packets and uses them as an indication that the destination has no route and route discovery should be triggered. This scheme has a few serious drawbacks. First, the kernel generates an ARP request only if the destination belongs to the subnet of one of the network interfaces, or a host-specific route entry exists for this destination. This limits the applications to certain types of network configurations. Secondly, ARP will time out in relatively short time, and mad-hoc provides no mechanism to queue outstanding packets. This means that these packets might be dropped before the route discovery can be completed. Finally, ARP cache has a time-out value and snooping on ARP requests can result in spurious route requests when a next-hop node has been timed out in the ARP cache but the route is still valid.

AODV-UU [2] and AODV-UCSB [1] are two implementations of the AODV routing protocol. The kernel interaction part of the two implementations is the same. They differ only in the AODV protocol logic implementation, which is done in user-space. The kernel part consists of two Linux kernel modules (`kaodv` and `ip_queue_aodv`). To address the on-demand routing problems, these implementations use Netfilter to copy all packets from the kernel space to user-space. `kaodv` uses Netfilter to collect all packets before they enter the packet-forwarding function and `ip_queue_aodv` queues them to user-space. By matching these packets against the entries in the user-space route cache, packets for which there is no route can be identified and route request initiated. There are two obvious drawbacks of this approach: every packet has to cross the user kernel address space twice, inducing much overhead, and for every packet the routing is done twice as well, once in user-space and once again in the kernel. In our AODV-UIUC implementation, all such system interactions are cleanly taken care of by the Ad-hoc Support Library. The overhead of processing every packet in user-space is eliminated. The result is a much simpler, cleaner, and more efficient implementation.

Kernel-AODV [4] is another AODV implementation by NIST. The entire implementation is in the form of Linux kernel modules. As we have discussed earlier in Section 2, it is not a good system design to put the entire routing protocol in kernel-space. The complex protocol processing can slow the kernel, hog the memory, and crash the whole system if there are any bugs in the routing protocol.

Another kernel implementation of AODV [9] has been done by extending ARP. Note that unlike Mad-hoc which uses the ARP mechanism just to detect route requests, this approach reuses the ARP code in the kernel to do a complete implementation of the AODV protocol. Modified ARP requests and replies are used to generate AODV RREQ and RREP packets. ARP table essentially acts as the AODV routing table. Modified ARP reply with a special flag is used to generate RERR messages. Data packets are buffered inside the kernel in an ARP queue. This approach is a smart idea, but has the limitations of an in-kernel implementations which we have discussed above.

Dynamic Source Routing (DSR) [15, 19] has been implemented by the Rice University Monarch project in FreeBSD [3]. This implementation is entirely in-kernel and does extensive modifications in the kernel IP stack. While the implementation is a commendable project, it is difficult to maintain and update due to its complexity. For example, this implementation of DSR was developed on FreeBSD 2.2.7 and has only been upgraded to FreeBSD 3.3, whereas the current version of FreeBSD is 4.6.2 when this paper is written. Porting it to other operating systems will be prohibitively difficult.

Internet Manet Encapsulation Layer (IMEP) [8] is an encapsulation protocol proposed for manet routing, which provides certain common functions like single-interface abstraction, link status sensing, control message aggregation, and reliable broadcasting. IMEP attempts to provide a unified framework for all other adhoc routing protocols at the protocol processing level, whereas we attempt to provide a common module and a common interface at the implementation level. These two approaches are complementary. A simple user-space implementation of IMEP is possible using our framework. Additional support may be needed from the network device driver for link status sensing.

Temporally ordered link reversal algorithm (TORA) [24] has been implemented in Linux by University of Maryland [6]. It has been implemented over IMEP and hence can benefit from our framework. TORA can independently be implemented over our framework too.

University of Colorado has implemented several routing protocols [32, 22, 10] using MIT's Click Modular Router [17]. Click is a software architecture for building flexible and configurable routers. It provides basic protocol processing modules called elements, each of which implements a specific function like packet classification, queuing, and scheduling. Protocol developers write Click configuration files to instantiate elements and to connect them in a way to implement the protocol. In this aspect, Click can be a good candidate to meet Challenge 4. Nevertheless, our work focuses on common operating systems, whereas Click is designed for a special purpose application (fast and configurable routers).

A recent work [16] aims to provide a library of utilities for manet routing protocols, much like the GNU Zebra project does for wired routing protocols. It plans to provide utilities for timer management, neighbor discovery, managing tables etc. It however does not systematically deal with all the system issues, as we have done. ASL could be used in building this framework.

MagnetOS [7] is a distributed, power-aware, adaptive operating system which abstracts an ad-hoc network as a unified Java Virtual Machine. It allows for objects and components to automatically migrate among nodes in the network to optimize system performance. MagnetOS focuses on sensor networks, using distributed operating system techniques.

7 Conclusions

In this work, we study the operating system services for mobile ad-hoc routing, and propose a generic architecture and API for implementing ad-hoc routing protocols in modern OS. We implement these services and API in Linux. With the helps of standard Linux primitives, we were able to do so without any modification in the core kernel source, i.e., without the need to recompile the kernel. The software consists of a user-space library (ASL) and protocol dependent loadable kernel modules. To demonstrate the flexibility of this approach, we implement AODV using ASL. We also give detailed design for implementing other routing protocols in this framework, and share our experience in implementing these different protocols.

We believe that the principle of separating routing and forwarding has profound importance in ad-hoc routing system design. Protocols that follow this principle are easier to implement in a clean way. The code will be efficient, portable, maintainable, and extensible. Protocols that violate this principle and mix routing and forwarding will require significantly more efforts to produce a clean and well-structured implementation. Unfortunately, little attention is paid in today's ad-hoc network research on the considerations of system issues in protocol design. Even if the protocol architecture follows the principle of separation, many protocols still suggest subtle optimizations that violate this rule. These optimizations are often easy to simulate but very difficult to implement in real systems. In some cases the complications encountered can nullify the benefits of the intended optimizations.

Implementing a routing protocol is very important to validate its design. Coming up with a clean implementation not only helps better understanding of the protocol nuances, but also allows extensions to explore the protocol design space. For example, many ad-hoc routing research efforts have extended upon AODV and DSR; having clean and extensible implementations of these two protocols would benefit these entire research directions.

Note: Source code for the Ad-hoc Support Library and AODV-UIUC is available under the GNU Public License from `http://aslib.sourceforge.net`. Source code for DSR implementation will also be available at this URL.

References

[1] AODV homepage. http://moment.cs.ucsb.edu/AODV/aodv.html.

[2] AODV-uppasala university. http://www.docs.uu.se/ henrikl/aodv.

[3] Implementation of DSR. http://www.monarch.cs.cmu.edu/dsr-impl.html.

[4] Kernel AODV. http://w3.antd.nist.gov/wctg/aodv_kernel/.

[5] Netfilter/Iptables homepage. http://www.netfilter.org.

[6] TORA/IMEP. http://www.cshcn.umd.edu/tora.shtml.

[7] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM Operating Systems Review*, 36(2):1–5, Apr. 2002.

[8] S. Corson, S. Papademetriou, P. Papadopoulos, V. Park, and A. Qayyum. An internet MANET encapsulation protocol (IMEP) specification, Aug 1999. IETF Draft, draft-ietf-manet-imep-spec02.txt, work in progress.

[9] S. Desilva and S. Das. Experimental evaluation of a wireless ad hoc network. In *Proceedings of the 9th International Conerence. on Computer Communications and Networks*, 2000.

[10] S. Doshi, S. Bhandare, and T. X. Brown. An on-demand minimum energy routing protocol for a wireless ad hoc

network. *Mobile Computing and Communications Review*, 6(2), July 2002.

[11] B. Gupta. Design, implementation and testing of routing protocols for mobile ad-hoc networks. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[12] Z. J. Haas. The routing algorithm for the reconfigurable wireless networks. In *Proceedings of IEEE International Conference on Universal Personal Communications (ICUPC'97)*, San Diego, California, Oct. 1997.

[13] Y.-C. Hu and D. B. Johnson. Implicit source routes for on-demand ad hoc network routing. In *Proceedings of the 2001 ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'01)*, pages 1–10, Long Beach, California, Oct. 2001.

[14] D. Johnson, D. Maltz, Y.-C. Hu, and J. Jetcheva. The dynamic source routing protocol for mobile ad hoc networks (DSR). IETF Internet-Draft, draft-ietf-manet-dsr-07.txt, Feb. 2002.

[15] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In T. Imielinski and H. Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[16] F. Kargl, J. Nagler, and S. Schlott. Building a framework for manet routing protocols. URL: http://medien.informatik.uni-ulm.de/∼frank/research/manetframework.pdf.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[18] U. C. Kozat, G. Kondylis, B. Ryu, and M. K. Marina. Virtual dynamic backbone for mobile ad hoc networks. In *Proceedings of IEEE ICC'01*, 2001.

[19] D. Maltz, J. Broch, and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed. Mar. 1999.

[20] D. A. Maltz. *On-Demand Routing in Multi-hop Wireless Mobile Ad Hoc Networks*. PhD thesis, Carnegie Mellon University, 2001.

[21] D. A. Maltz, J. Broch, J. Jetcheva, and D. B. Johnson. The effects of on-demand behavior in routing protocols for multi-hop wireless ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1439–1453, August 1999.

[22] N. K. Palanisam. Modular implementation of temporally ordered routing algorithm. Master's thesis, University of Colorado, 2001.

[23] V. Park and S. Carson. Temporally-ordered routing algorithm (TORA) version 1 functional specification. IETF Internet-Draft, draft-ietf-manet-tora-04.txt, July 2001.

[24] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of IEEE INFOCOM*, 1997.

[25] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, Feb. 1999.

[26] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of ACM SIGCOMM'94*, London, U.K., Sept. 1994.

[27] 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-10.txt, work in progress, Jan. 2002.

[28] L. L. Peterson and B. S. Davie. *Computer Networks*. Morgan Kaufmann Publishers, 2nd edition, 2000.

[29] E. M. Royer and C. E. Perkins. An implemenatation study of the aodv routing protocol. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, 2000.

[30] C.-K. Toh. A novel distributed routing protocol to support ad hoc mobile computing. In *Proceedings of IEEE 15th Annual International Conference on Computers and Communications*, pages 480–486, Phoenix, March 1996.

[31] C.-K. Toh. Long-lived ad hoc routing based on the concept of associativity. IETF Internet-Draft, draft-ietf-manet-longlived-adhoc-routing-00.txt, Mar. 1999.

[32] A. Tornquis. A modular framework for implementing ad hoc routing protocols. Master's thesis, University of Colorado, 2000. http://systems.cs.colorado.edu/Networking/modular-adhoc.html.