Using Active Networking's Adaptability in Ad Hoc Routing

Seong-Kyu Song and Scott M. Nettles

Electrical and Computer Engineering Department
The University of Texas at Austin
{sksong, nettles}@ece.utexas.edu

Abstract. The early goals of Active Networking (AN) were to increase the pace of network evolution and to facilitate application specific protocols. Our aim is to demonstrate that for a specific application domain, Ad Hoc network routing, these goals have been substantially met. We argue that Ad Hoc networking is a domain that is well suited for this demonstration, due to its needs for both evolution and adaptation.

We support our claim by building a series of Ad Hoc routing protocols, based on both DSR and AODV, that demonstrate heavyweight evolution, lightweight evolution, and routing adaptation. We based our design and implementation on our Mobile Active Networking Environment (MANE). MANE is a direct descendant of PLAN/PLANet and, as such, supports both Active Packets and Active Extensions as programmability mechanisms, thus giving us maximum flexibility in our demonstrations.

Keywords: MANET, Ad hoc Routing, Active Networking, Adaptation

1 Introduction

The original goals of Active Networking (AN) were clear: First, to make it easier to deploy new protocols or alter existing protocols to allow the network to evolve more readily; and, Second, to allow protocols to be customized to specific application needs. AN attempts to meet these goals by adding programmability to the network infrastructure. Although there have been a significant number of AN systems proposed and implemented [1–5], there has been less work done to show that these systems meet AN's original goals. The purpose of this paper is to show that, for a specific application domain, a mature, well-understood AN system can meet these original goals.

For the AN system, our Mobile Active Network Environment (MANE) [6] was the obvious choice. MANE is the most recent embodiment of our work on PLAN [1, 7] and is a direct descendent of PLANet [8]. Like our earlier work, MANE combines programmable Active Packets (APs) with downloadable node resident Active Extensions (AEs), thus allowing us to explore both of the two principle AN programmability approaches in the same context. In MANE, APs

carry PLAN programs that execute as the packet moves through the network. APs provide the programmable "glue" that binds the network together. AEs form the basis of the node-resident programmable infrastructure by allowing new functionality to be downloaded into the nodes, either to modify node behavior or to provide new services callable by APs. MANE goes further than any of our PLANet implementations in its support for AEs since it provides not just *plug-in extensions*, but also *dynamic-update extensions* [9, 6]. When used in conjunction with plug-in extensions, APs can use new node-resident services specialized to their needs if the standard services are not sufficient. Further, dynamic-update extensions can update a system's functionality while the node remains operational and can affect the operation of existing functionality, even if there has been no pre-planning to provide a plug-in interface. The distinction between plug-in and update extensions is discussed in more detail in [9, 6].

As an application domain, we chose routing for mobile ad hoc networks (MANETs) [10]. There are a number of reasons for this choice [11]. First, MANET routing is a very active area of research and the potential protocols of interest are still changing. Thus, if AN does facilitate evolution, it would be possible to deploy existing routing algorithms with the expectation that they could be easily replaced by better algorithms as they are developed. Second, MANET environments can vary greatly and the preferred routing algorithm can be different for different environments. Thus, if AN does facilitate application specific protocols, it should be possible to choose and dynamically deploy the best algorithm for the environment at hand. Third, the conditions present in a MANET may change so that the algorithm currently in use is no longer optimal. The ideal routing protocol may need to change dynamically. AN offers the possibility of adapting the algorithm dynamically as conditions change. Finally, because MANETs are not widely deployed or standardized, it is quite possible that a node will not have the desired algorithm present. It is even possible that a node will have no available MANET routing algorithm. AN can provide us with the ability to deploy the desired algorithm on-the-fly.

In this paper, we focus on two well-known MANET routing protocols, Dynamic Source Routing (DSR) and Ad-hoc On-demand Distance Vector routing (AODV). We chose these protocols because they are perhaps the most widely accepted and studied of the myriad of possible choices. Using these protocols, we demonstrate implementations that realize the possibilities discussed above. First, we show how a simple version of DSR could be deployed on a network where it was not currently deployed and where perhaps no ad hoc routing protocol was available. Second, we show how that simple version can be evolved dynamically into a superior version even without changing the code resident on the nodes. Third, we show how AN enables us to create a hybrid of DSR and AODV that allows us to adapt to changing conditions in the network dynamically.

The remainder of this paper is organized as follows. Section 2 is an overview of the two ad hoc routing protocols, DSR and AODV. In Section 3, we discuss AN technologies and our AN platform. Section 4 presents our implementation of a simple version of the DSR protocol. Section 5 demonstrates how we can deploy

a new ad hoc routing protocol on a network where no ad hoc routing protocol is available. In Section 6, we describe how to use active packet evolution to evolve our simple version of DSR into a more efficient one without modifying node-resident code. Section 7 presents a protocol that is a hybrid of DSR and AODV that can adapt to changing network conditions. Finally, Section 8 concludes the paper.

2 Ad Hoc Routing Background

Understanding the examples we present requires some basic knowledge of how the routing protocols we have chosen work. The two protocols, DSR and AODV, are both reactive (or *on-demand*) protocols. This means that rather than always maintaining a route to all destinations (proactive routing) they find a route on-demand when it is actually needed. When a packet needs to be sent and a route is not already known, both protocols find routes by flooding the network with a route request packet. When the destination is found, a route reply packet is sent, which sets up the needed data structures for each protocol to actually send the packet. The protocols differ in the exact nature of this discovery process, in the nature of the routes, and in many details of the basic process.

2.1 Dynamic Source Routing

The DSR protocol [12] uses data packets that carry source routes that specify each next-hop node directly in the packet. It is composed of *Route Discovery* and *Route Maintenance* operations. In the Route Discovery phase, when a route is needed, a source node (S) attempts to obtain a source route (the sequence of nodes that the packet should visit) to a destination node (D) by flooding ROUTE REQUEST packets throughout the network. The request packets collect route information as they are propagated through the network. The first route request packet to reach the destination returns a ROUTE REPLY packet with the sequence of nodes it visited. When the route reply packet reaches the source, the source route it contains is used to send the data packet. In order to reduce routing overhead and make the best possible use of route information, each node maintains a *route cache* into which the new route is also entered. As described in Section 6, in more highly optimized versions of the protocol, this route cache can be used to short-cut route requests. In the Route Maintenance phase, S is notified of the link failures, if any, by nodes adjacent to the broken link. Then, S will initiate another route discovery operation by generating a new route request packet.

2.2 Ad-hoc On-Demand Distance Vector

The AODV routing protocol [13] is the on-demand version of the Destination Sequenced Distance Vector routing protocol [14]. Unlike DSR, AODV data packets carry only a destination address; next-hop addresses are maintained in routing

tables on the intermediate nodes. However, AODV still has the same basic Route Discovery structure as DSR, the route reply packets simply must set up the intermediate nodes' routing table while returning to the source. AODV also uses sequence numbers to discern stale routes and maintain route freshness. AODV also has a Route Maintenance aspect, which is similar to DSR's. All of this means that the basic implementation structure of AODV is similar to DSR, but many of the key details are different.

In spite of their similarities, it has been shown that the two protocols perform differently under various network conditions, especially the degree of network mobility [15, 16]. It appears that DSR may be more sensitive to mobility than AODV. Under lower mobility, since there are relatively few link changes, DSR's aggressive caching strategy is effective in achieving better performance than AODV. However, in high mobility cases, AODV seems to do better than DSR because of more conservative routing management [16].

3 Active Networking and MANE

AN provides adaptability to facilitate application-specific customization and speedy network service evolution [17]. In this section, we describe the programmability mechanisms of AN and the modifications we made to MANE to support ad hoc networking.

3.1 Programmability Mechanisms

There are two basic mechanisms for adaptability in AN: *Active Packets* and *Active Extensions* [17]. APs carry programs that execute as they pass through the nodes. Packet execution can perform management actions on the nodes, affect their own routing, or form the basic distributed computational framework of larger protocols. Since packet programs can accomplish protocol implementation on-the-fly, they are a quick and effective way of deploying new services in existing networks. Also, packet-by-packet adaptivity enables the network to adjust agilely to changing environments.

Complementary to APs are AEs, which form the basis of the node-resident programmable infrastructure by providing the services callable by APs. AE's can be dynamically downloaded to modify a nodes behavior [18, 6]. When used in conjunction with *plug-in extensions*, packet programs can use new node-resident services specialized to their needs if standard services are not sufficient. Further, *update extensions* can update a system's functionality while the node remains operational. Update extensions can affect the operation of existing functionality, even if there has been no pre-planning to provide a plug-in interface.

The flexibility of these two mechanisms together makes AN a good choice for environments that require a high degree of adaptivity, such as MANETs. In MANETs, as the nodes move, link conditions may change frequently; thus the routing protocol needs to cope with those variations nimbly. Moreover, because ad hoc networks can occur without prior planning, it is entirely possible that

the ideal routing algorithm may not be known in advance and may change as the network is in use. To overcome such *routing heterogeneity*, it is desirable to promptly conform to a unified protocol. AN's ability to implement a protocol on-the-fly makes it possible to agilely evolve and adapt routing protocols.

3.2 MANE

Our Mobile Active Network Environment (MANE) [6] implements the SwitchWare architecture [1] and is the descendant of our previous AN testbed, PLANet [8]. Active packets are written in the Packet Language for Active Networks (PLAN) [7] and service functions are written in Popcorn [19], which is a C-like type-safe language based on TAL (Typed Assembly Language) [20]. Here we describe the modifications of MANE needed to support ad hoc networking in general and in particular to support on-demand routing protocols. Note that the first two modifications are really to our underlaying emulation, in a "real" network they would not be needed. The last two modifications would be needed in real networks and in Section 5 we discuss how they could be achieved dynamically using our AN mechanisms.

Addressing Like an IP address, MANE addresses are globally unique and hierarchical. A node is identified by a network number and a host number. The hierarchy is based on sub-nets of nodes and each node on a sub-net can broadcast to all other nodes. Communication with nodes on other networks must be mediated by routers. Based on this hierarchy, MANE supports Mobile-IP-like mobility by utilizing AN's evolution techniques [6]. For ad hoc networks, where each node works as a router, we modified MANE to use a flat addressing scheme, where host numbers are used as a unique address.

Mobility Emulation MANE emulates broadcast networks by keeping track of which nodes are on a particular sub-net and using UDP to communicate between neighbors. Broadcast is achieved by repeatedly unicasting to every neighbor[1]. This also supports emulation of physical node mobility, allowing a node to leave a sub-net and to join new sub-nets. Even though this emulation is transparent to higher-level protocols, MANE needed to inject special APs to disconnect and connect a node [6].

For ad hoc networks, we need a more scalable and distributed way of emulating physical mobility. Therefore, we adopted a method similar to that used by `ns-2` for wireless network simulations [21]. There is a pre-generated mobility file describing the physical movement. Also, there is a virtual master node with a global "bird's eye" view, whose role is to update neighbor information by sending neighbor information packets periodically to every node. The virtual

[1] It should be clear from this description that the goal of MANE is to allow flexible experimentation with models and functionality, not to provide a high performance AN implementation.

master obtains neighbor information from the mobility file. Neighbor information is used only in emulating physical mobility and wireless link broadcasting, not in network-layer routing.

Routing Buffer in the Network Layer Since we are experimenting with reactive ad hoc routing protocols, there needs to be a buffer – *the routing buffer* – to hold the packets during route discovery. When a route is discovered, the corresponding packets are released from the routing buffer and pushed into the lower layer queue for transmission. To support reactive routing protocols, MANE implements the routing buffer in the network layer. If there is no route information for a packet, a sender saves the packet in the routing buffer and initiates route discovery. Route reply packets cause the sender to free the packet from the routing buffer and resume the transmission of the packet.

Link Layer Acknowledgements Any link can be broken due to either node movements or channel deterioration and ad hoc routing protocols need to be able to discover these failures. For route maintenance and detecting link breakage, we added link-layer acknowledgements to MANE. After transmitting a packet, the link layer saves the packet in the *interface queue* and waits for acknowledgement. If there is no acknowledgement during a timeout period or if a negative acknowledgement is received, the link layer retransmits the packet. When a certain number of trials fail, the node recognizes it as link breakage.

4 A Simple Version of DSR

We first present a simple version of the DSR protocol[2], which we will later show how to deploy and evolve. In our initial simple version, no use of the route cache is made at the intermediate nodes. All intermediate nodes simply re-broadcast the first instance of a route request received after appending their own address, and ROUTE REPLY packets are generated only by the destination.

In MANE, a protocol is implemented in two levels; active extensions and active packets. AE's are node-resident and implement the service functions needed for the protocol, while APs serve to glue together the AE functionality and actualize the protocol. We first present the services needed for DSR, followed by the AP's that are used by the protocol.

4.1 An Active Extension for DSR

Table 1 shows node resident services needed by DSR. `Get_ID()` generates a unique identification number for a new route request. There are two functions, `LookUp_RouteCache()` and `SaveIn_RouteCache()`, for managing the *Route Cache*. To filter out duplicate requests, `Mark_Dup_Request()` and `Check_Dup_Request()` are used to manage the *Duplicate Request Check List*.

[2] In referring to the DSR protocol, we mean the basic idea of DSR, not literally the DSR standard.

Table 1. Service Functions for DSR

Functions	Types
Get_ID()	`null` $\Longrightarrow$ `int`
LookUp_RouteCache(dest)	`host` $\Longrightarrow$ `host list`
SaveIn_RouteCache(dest, srcRoute)	`host*(host list)` $\Longrightarrow$ `null`
Mark_Dup_Request(source, ID)	`host*int` $\Longrightarrow$ `null`
Check_Dup_Request(source, ID)	`host*int` $\Longrightarrow$ `bool`

4.2 Active Packets for Basic Route Discovery

Figure 1 shows the pseudocode for route discovery, while Figure 2 shows the PLAN implementation. The pseudocode shows that as the packet executes at each node duplicates are discarded. Then, if the packet is at the destination a route reply is sent and the route is saved, anticipating the possibility of data being sent back to the source. If the packet is not at the destination, the current address is simply added to the route and the packet is reflooded.

In addition to the service functions above, the PLAN code uses a number of PLAN core services and language constructs. `thisHostIs()` returns a boolean value indicating whether the given network address matches the address of the current node. `getSrcDev()` returns the interface on which the packet arrived, and `thisHostOf()` returns the network address corresponding to the given device. Using these functions and the list operator for concatenation, `::`, the route request packet can obtain the source route as it is propagated through the network (Lines 9–13). `OnNeighbor()` is a network primitive that generates a child AP executing on a neighbor of the current node. `getRB()` returns the amount of resource bound available in the packet.

The actual implementation corresponds closely to the pseudocode. In Line 2 route discovery starts by checking for duplicate requests. If the request has been already seen, this packet is discarded (Line 15). If not, it will save the tuple <source address, request id> in the Duplicate Request Check List (Line 3). If the request has arrived at the destination, D saves the source route to S and generates a route reply packet (Lines 4–7). Based on the assumption that links are bi-directional, the source route is reversed to be used as a route for the route reply. If this is an intermediate node, the nodes address is prepended to the current source route and `OnNeighbor` is used to broadcast the request to all the 1-hop neighbors (Lines 8–14).

Figure 3 shows the pseudocode for route reply, while Figure 4 shows the PLAN implementation. The pseudocode shows that a packet is simply forwarded at intermediate nodes, while at the source the route is saved in the cache and then any data destined for the destination is sent.

Again, the PLAN code corresponds closely to the pseudocode. If the reply has arrived at the source, the route is saved and route discovery exits, triggering (implicitly) the data packets to be sent. Lines 7–11 show how the reverse source

```
 1: INPUT: destination address D, list of hosts R
 2: if this is a duplicate request then
 3:    discard this packet
 4: else
 5:    if arrived at D then
 6:       send Route Reply with R
 7:       save R in route cache
 8:    else
 9:       append my address to R
10:       flood this request to all neighbors
11:    end if
12: end if
```

Fig. 1. Pseudocode for Basic Route Discovery

```
1: fun routeDiscovery(src, dst, id, srtRecord) =
2: if(not Check_Dup_Request(src, id)) then (
3:   Mark_Dup_Request(src, id);
4:   if(thisHostIs(dst)) then (
5:     SaveIn_RouteCache(src, srtRecord);
6:     routeReply(src, dst, srtRecord, reverse(srtRecord))
7:   )
8:   else (  (* intermediate nodes *)
9:     let val myAddr = thisHostOf(getSrcDev())
10:    in
11:      OnNeighbor(|routeDiscovery|(src, dst, id, myAddr::srtRecord),
12:                 broadcast, getRB(), getSrcDev())
13:    end
14:  )
15: else () (* dup req. discard *)
```

Fig. 2. PLAN for Basic DSR Route Discovery

route is used at an intermediate node. In Line 7 the `nextHop` is read from the front of the list and in Line 8 it is removed from the list. In Line 9–10 `OnNeighbor` is used to send the reply to the next hop, along with the truncated route.

5 Deploying DSR

Given the varied environments faced by MANETs, it is quite possible that the most appropriate routing algorithm will not already be deployed on all the nodes. In fact, given that MANETs are a new technology, it is possible that no routing algorithm of any kind is deployed. This is exactly the sort of problem that AN was designed to solve. In particular, let us consider how we could deploy our simple version of DSR.

Our DSR implementation has two components, the AE making up the service routines and the APs that use these routines. Since the APs carry their own code

```
1: INPUT: source address S, list of hosts R
2: if arrived at S then
3:    save R in cache
4:    exit route discovery
5:    send data using R
6: else
7:    forward this packet to S
8: end if
```

Fig. 3. Pseudocode for Basic DSR Route Reply

```
1: fun routeReply(src, dst, srcRoute, routing) =
2: if(thisHostIs(src)) then (
3:   SaveIn_RouteCache(dst, srcRoute);
4:   exitRouteDiscovery()
5: )
6: else (
7:   let val nexthop = hd(routing)
8:       val routing = tl(routing)
9:   in  OnNeighbor(|routeReply|(src, dst, srcRoute, routing),
10:                 nexthop, getRB(), getSrcDev())
11:  end
12: )
```

Fig. 4. PLAN for Basic DSR Route Reply

with them, deploying them is trivial, we simply inject the required APs into the network. Deploying the AE is only slightly more complex.

In MANE, code for an AE can be dynamically linked into a running node [9]. During this linking process, the AE can define new services that can be called from PLAN. Once this has been done the APs that use those services will be able to function. Now the only question is how to discover which nodes need to have the AE installed and how to transport the code to those nodes. There are many possible approaches, for example, we could imagine an ANTS-like [2] system where APs implicitly discover whether the needed code is node-resident and then download it from predecessor nodes or perhaps from some global repository. Another possibility is that AEs could be downloaded from a central repository, perhaps on demand.

For illustrative purposes and implementation simplicity, our implementation uses a simpler approach. The route request packet carries the extension in the packet itself and tests to see if it needs to be loaded as it floods the network. Figure 5 shows the pseudocode for this simple solution. In Line 2, the packet checks if the extension it needs is present. If not, it will dynamically load and install the extension on the node before executing route discovery. This simple use of *plug-in evolution* [6] allows us to deploy the DSR protocol dynamically and in a timely manner. Although simple and elegant, it does seem likely that

```
1: INPUT: destination address D, list of hosts R, Extension E
2: if DSR Service Not Present then
3:    Load DSR Extension From This Packet
4: end if
5: DSR Route Discovery
```

Fig. 5. Dynamic DSR Deployment

space and security considerations may make this approach less desirable in real systems.

We have swept one potentially important point under the rug. Most of the changes we made to MANE that were described in Section 3.2 were really concerned with improving our emulation of mobility and would not be needed for a real network. However, some of the changes would actually need to be made to support DSR or AODV. In particular, the proactive routing algorithms typically used in wired networks have no need to potentially queue packets when a route does not exist, they simply drop those packets. Adding this queue is not simply a matter of plugging in a new PLAN callable service function, it requires more fundamental changes to the node implementation.

This is an excellent example of where MANE's support for "update extensions" comes into play. Using dynamic updating technology [9], we can load an extension that makes significant changes to the node implementation, including inserting the new queuing mechanism. Similarly, we could used update extensions to add the link-level acknowledgements needed to support route repair.

6 Evolving DSR

The ability to deploy a new protocol on-the-fly using AEs is a powerful mechanism for evolving the network. However, it is also a heavyweight mechanism, requiring that code be dynamically linked into a running node. Using update evolution is even heavier weight, since it enables almost arbitrary changes to be made to a node.

It seems likely that only a few network users will be trusted to make these kinds of heavyweight changes to running network nodes. Does this mean that only those privileged users will be able to evolve or customize the network?

In this section, we show that significant protocol evolution can be achieved without resorting to making permanent changes to the node. The key mechanism is, of course, packet programmability. If there is a need to evolve or customize a routing protocol, APs can implement the new one without modifying the services of the nodes in the network. This kind of *Active Packet evolution* [6] enables the network to promptly evolve with the help of common and reusable AE's. PLAN plays an important role here, because its strong safety and security guarantees allow unprivileged, third party user to safely program the network.

```
1: INPUT: source address S, destination address D, list of hosts R
2: if this is a duplicate request then
3:    discard this packet
4: else
5:    save R in cache for S
6:    if arrived at D then
7:       send Route Reply with R
8:    else
9:       if route found in route cache then
10:         send Route Reply with R and found route
11:      else
12:         append my address to R
13:         flood this request to all neighbors
14:      end if
15:   end if
16: end if
```

Fig. 6. Pseudocode for Optimized Route Request

```
1: INPUT: source address S, destination address D, list of hosts R
2: save R in cache for D
3: if arrived at S then
4:    exit route discovery
5:    send data using R
6: else
7:    forward this packet to S
8: end if
```

Fig. 7. Pseudocode for Optimized Route Reply

6.1 Active Packets for Optimized DSR

Our initial DSR implementation is quite simple and does not take advantage of many of the optimizations that are possible. In particular, intermediate nodes simply implement flooding, despite having route caches that might contain the route that we are searching for. To utilize route control packets efficiently and to reduce routing overhead, the protocol needs to be optimized by allowing intermediate nodes to aggressively participate in routing. Specifically, request-broadcasting nodes can obtain a source route to S, and reply-forwarding nodes can acquire a source route to D. They keep those route information in their route caches for later use. Before re-broadcasting the request, intermediate nodes can search their route cache. If there is a valid entry, they can respond without re-broadcasting the request further. Most importantly, we can implement this optimized DSR by only re-programming APs, and we do not need to modify the DSR services in a node-resident AE.

Figure 6 and Figure 7 show the pseudocode for optimized DSR route discovery. The underlined portions indicate the parts that have been added to our

initial simple implementation. We have not included our PLAN code, as with the simple DSR implementation, it mirrors the pseudocode closely.

At intermediate nodes, route discovery changes in two basic ways. First, in addition to flooding the route discovery packet, the packet also saves the partial route in its cache (Line 5), thus increasing its knowledge of possible routes at essentially no cost. Second, the packet looks in the intermediate node's cache for a route to the destination (Line 9). If the route exists, then the node returns the packet's route record concatenated with the cached route (Line 10), thus short-cutting the route discovery process. Route reply adds a single optimization, replies also add routes to the route caches on intermediate nodes (Line 2).

Although in this example, new APs are used to perform a general optimization, they can also be used to perform application-specific customizations as well. For example, in the current protocol, if no route reply short cutting occurs, the route that is chosen is the one taken by the first route request packet to arrive at the destination. An application might desire to use a different metric, say the route that has the largest bottleneck bandwidth. Assuming we had service routines that could tell us link bandwidths, then we could easily program a route request packet that would measure the bottleneck bandwidth and return a route reply for any route request that arrived at the destination with a better value than previous route requests.

7 A Hybrid Routing Protocol

We have seen how AN can be used to deploy new, improved, or customized protocols in a MANET environment. These examples show that AN's adaptability can help to accommodate the wide variety of environments MANETs face. Because of their dynamic nature, not only do we expect MANETs to be used with widely varying network conditions, but we also would expect that those conditions may well change while a network is operational, perhaps rapidly. In this section, we show that AN can be used to adapt to such changing conditions.

In Section 2, we presented some background information on both DSR and AODV. A key point is that AODV appears to work better when levels of mobility are high, while DSR appears to work best when mobility is low. Thus, even if the preferred protocol is in use, it is entirely possible that the level of mobility may shift, making it desirable to change protocols.

Our approach is to build a hybrid protocol that can easily switch between AODV and DSR as mobility levels change. The possible design space for such hybrid protocols is immense and it is important to keep in mind that our goal is to demonstrate that AN has achieved its goals with respect to adaptability, not to explore this design space or to propose the "best" protocol. By showing a fairly simple example, it should be clear that AN techniques will facilitate the implementation, development, and exploration of a wide variety of such adaptive protocols.

Table 2. Service Functions for Hybrid Protocol

Functions	Types
LookUp_RIB (routing protocol, dest)	`string*host` ⇒ `host*int*int` *or* ⇒ `host list`
SaveIn_RIB(dest, destSeq, hopCount, nextHop) *or* (dest, source_route)	`host*int*int*host` *or* `host*(host list)` ⟹ null
Get_RREQ_ID()	`null` ⟹ `int`
Mark_Dup_Request (source, RREQ_ID)	`host*int` ⟹ `null`
Check_Dup_Request (source, RREQ_ID)	`host*int` ⟹ `bool`
Get_SrcSeq()	`null` ⟹ `int`
Get_DestSeq(dest)	`host` ⟹ `int`

7.1 An Active Extension for the Hybrid Protocol

The key to creating a hybrid protocol that can switch rapidly between differing algorithms is to create a set of generic AE services that can be used by all algorithms. Once this is done, we can then accomplish the actual switching between protocols quite easily using APs. This general idea is an important aspect of AN, by providing generic, reusable, composable node resident components, we can then use packet programs to create many different protocols and enable switching between protocols easily.

Here, we take this idea only so far by creating generic services common to both DSR and AODV as shown in Table 2. The most important of these, `LookUp_RIB()` and `SaveIn_RIB()`, manipulate a generic *Route Information Base (RIB)*, which is a combined form of DSR route cache and AODV route table. Notice that we have used parametric polymorphism so that these functions can take arguments and return values that are appropriate to either DSR or AODV. The next three services, `Get_RREQ_ID()`, `Mark_Dup_Request()`, and `Check_Dup_Request()`, are concerned with duplicate elimination during flooding. These are good examples of general services that we might expect to see reused by many different protocols and in fact, they have already appeared in our simple DSR implementation. The final two services, `Get_SrcSeq()` and `Get_DestSeq()`, are concerned with manipulating sequence numbers. Although here they are specific to the AODV aspect of our protocol, we can certainly imagine that with more experience, we could define a general set of sequence number manipulation services that would be reusable across a variety of protocols.

7.2 An Active Packet for the Hybrid Protocol

Using the services above, we can now program an AP that can adapt to changing conditions. If we actually wished to deploy an adaptable protocol, a key question would be *when* to adapt. However, our goals is to show that adaptation is feasible, not to research how best to do it. Thus we assume there exists some global policy module that monitors mobility and informs us as to when to adapt. That AN makes such a adaptive protocol feasible means that it would be interesting future work to explore how to build such a monitor.

Figure 8 shows the PLAN program for hybrid routing request. The AP for the hybrid route request contains three functions: `routeRequestAtSrc()`, `dsrRREQ()`, and `aodvRREQ()`. The source, S, evaluates `routeRequestAtSrc()` and decides which protocol to use. At low mobility, S injects a DSR route request packet by calling an `OnNeighbor()` that evaluates `dsrRREQ()` on all the neighbor nodes (Lines 2–4). At high mobility, S spawns a child AP that executes `aodvRREQ()` with the appropriate sequence numbers and a hop counter (Lines 5–7). The two functions, `dsrRREQ()` and `aodvRREQ()`, contain the algorithm for the route request of the corresponding routing protocol.

In the interest of space, Figure 8 shows only our functions for the route request. The complete AP would include the route reply functions as well. When there is valid information for the request (on intermediate nodes or the destination node), a reply packet is generated by the function call, `dsrRREP()` (Lines 14 & 20) or `aodvRREP()` (Lines 34 & 43). The optimized DSR protocol allows intermediate nodes to reply to the request (Lines 19–22). In replying with cached information, the reply-generating node needs to concatenate the route record and cached information (Lines 20–21). In AODV, the destination sequence number is compared to validate freshness of the cached information (Line 39)[3].

7.3 Discussion

Our results show that it is not difficult to take two protocols that are similar in structure, but which differ in many key details and essentially combine them. But what if the protocols differ significantly in their basic structure? An obvious example would be our current reactive algorithms compared to proactive algorithms which always maintain routes to all destinations. Designing a system that adapted between reactive and proactive would be more challenging than our current approach. However, the key point is that if, such a hybrid was designed, AN would make it easier to deploy and evolve. However, it is important to be clear that AN is just an implementation and deployment approach, it offers no silver bullet for making hard design problems easier.

7.4 Simulation of the Hybrid Protocol

One significant limitation of our MANE based implementation is that it is difficult to generate meaningful performance results. This is because MANE nodes

[3] In PLAN, $\#n$ returns the n-th element of a tuple. In Figure 8, #3 of `rt_entry` is a destination sequence number and #4 is a hop count.

```
1: fun routeRequestAtSrc(src, dst) =
2: if(mobility = 0) then
3:  OnNeighbor(|dsrRREQ|(src, dst, Get_RREQ_ID(), [ ]),
4:             broadcast, getRB(), getSrcDev())
5: else
6:  OnNeighbor(|aodvRREQ|(src, dst, Get_RREQ_ID(), Get_SrcSeq(),
7:        Get_DestSeq(dst), 0), broadcast, getRB(), getSrcDev())
8:
9:  fun dsrRREQ(src, dst, id, srtRecord) =
10: if(not Check_Dup_Request(src, id)) then (
11:   Mark_Dup_Request(src, id);
12:   SaveIn_RIB(src, srtRecord);
13:   if(thisHostIs(dst)) then
14:     dsrRREP(src, dst, srtRecord, reverse(srtRecord))
15:   else (  (* intermediate nodes *)
16:    let val myAddr = thisHostOf(getSrcDev())
17:        val newSrtRecord = myAddr::srtRecord
18:    in ( try (
19:      let val srcRt:(host) list = LookUp_RIB("DSR", dst)
20:      in dsrRREP(src, dst, listcon(reverse(srcRt),
21:                 newSrtRecord), reverse(srtRecord))
22:      end )
23:     handle NotFound => (
24:         OnNeighbor(|dsrRREQ|(src, dst, id, newSrtRecord),
25:            broadcast, getRB(), getSrcDev())
26:     ) ) end ) )
27: else () (* dup req. discard *)
28:
29: fun aodvRREQ(src, dst, id, srcSeq, dstSeq, hopCount) =
30: if(not Check_Dup_Request(src, id)) then (
31:   Mark_Dup_Request(src, id);
32:   SaveIn_RouteCache(src, srcSeq, hopCount+1, getSrc());
33:   if(thisHostIs(dst)) then
34:      aodvRREP(src, dst, dstSeq, 0)
35:   else (  (* intermediate nodes *)
36:    try (
37:     let val rt_entry:(host*dev*int*int) = LookUp_RIB("AODV", dst)
38:     in (
39:       if(dstSeq > #3 rt_entry) then (
40:         OnNeighbor(|aodvRREQ|(src, dst, id, srcSeq, dstSeq,
41:           hopCount+1), broadcast, getRB(), getSrcDev()))
42:       else
43:         aodvRREP(src, dst, #3 rt_entry, #4 rt_entry)
44:      ) end )
45:    handle NotFound => (
46:      OnNeighbor(|aodvRREQ|(src, dst, id, srcSeq, dstSeq, hopCount+1),
47:                 broadcast, getRB(), getSrcDev()) ) ) )
48: else () (* dup req. discard *)
```

Fig. 8. PLAN for Hybrid Route Request

are virtualized and typically run many instances on each real node and because the physical network is emulated by using unicast UDP transmission. Thus it is impossible for us to usefully quantitate the overheads imposed by our approach. Despite this, and despite our goal not being primarily to explore the design of hybrid routing algorithms, we still wanted to see if we could show that such an algorithm could indeed result in improved performance when faced with changing mobility. To explore this question we simulated our algorithm as well as DSR and AODV.

Experimental Setup As a simulator, we used `ns-2`, which is a discrete event simulator widely used in networking research [21]. As a measure of performance, we used the Packet Delivery Ratio (PDR). PDR is the ratio of the number of packets received to the number of packets transmitted and larger numbers are better. For a direct comparison, we used CBR traffic rather than TCP traffic because congestion control and flow control offer different loads according to network conditions for TCP. Each node moves according to the "random waypoint" model [12], in which the nodes repeatedly move and then pause. In this model, the pause time and the movement speed characterize the mobility of the network. In each simulation, the same scenarios of movements and traffic are used for DSR, AODV, and the hybrid protocol. The reported values are averages taken from ten simulations under different movements and traffic scenarios.

The packet size is 512 bytes, and 4 packets are generated per second. The number of CBR sources is 25 out of 50 total nodes. For each simulation, 50 nodes move around in a 1000 m $\times$ 1000 m square space for 1500 seconds. To simulate changing mobility, we divided the simulation time into 3 parts of 500 seconds each. In the first part (0–500 seconds), there is no movement and the network is stationary. In the second part (500–1000 seconds), all the nodes move at a maximum speed of 10 m/s with a pause time randomly selected between 0 and 250 seconds. In the last 500 seconds, the maximum speed is 20 m/s and the pause time is 0 seconds. For the hybrid protocol, initially DSR is used and as the mobility increases the nodes switch to AODV. Specifically, during the first half of the simulation, route control packets follow DSR semantics and data packets are routed using DSR. After 750 sec., the interface for the routing protocol is changed to AODV and route control packets follow AODV semantics. For the simulation of DSR and AODV, we used the existing `ns` versions developed by the Monarch project [22].

Results The simulation results are shown in Figure 9. The x-axis is simulation time and the y-axis is the PDR. We observe that in general as mobility increases, the PDR decreases because of more frequent link failures or changes. However, DSR and AODV have different rates of decrease and there is a crossing point where which is superior changes. In particular, while DSR's is better than that of AODV under low mobility, DSR shows more degradation as mobility increases. On the other hand, AODV is relatively robust to changes in mobility.

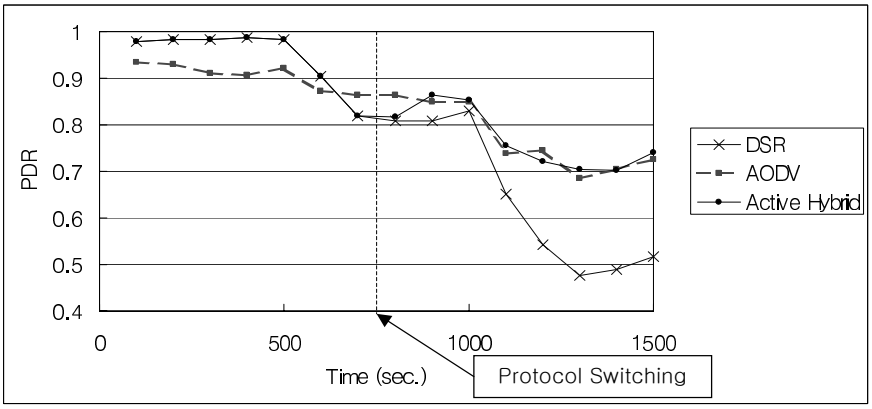

Fig. 9. PDR over time for DSR, AODV, and Hybrid

Not surprisingly, since the hybrid protocol switches between DSR and AODV, its performance basically follows the better protocol in the whole range of mobility. At low mobility, the hybrid protocol adopts DSR's aggressive route discovery and caching scheme and it performs similarly to DSR. However, as mobility increases, it works like AODV and becomes robust to increased mobility. The region from 500 to 750 seconds is the only exception, because during that period, we have not switched away from DSR. From the simulation results, we see that the hybrid protocol is adaptive to network mobility and suitable for networks under varying mobility environments.

8 Conclusion

In this work, we have demonstrated how AN can be used to deploy, evolve, and adapt ad hoc routing protocols. In some cases, this has used both heavyweight AE programmability and lightweight AP programmability. However, we have also seen that if the right generic services can be provided, lightweight AP programmability can be a powerful tool by itself. These demonstrations argue that AN has achieved its initial goals of facilitating network evolution and customization, at least in this domain. Further we believe these demonstrations show than AN can play a significant role in building MANETs that are easy to deploy, experiment with, and which can respond to the challenges of the diverse MANET environment.

References

1. Alexander, D., Arbaugh, W., Hicks, M., Kakkar, P., Keromytis, A., Moore, J., Gunter, C., Nettles, S., Smith, J.: The SwitchWare Active Network Architecture.

IEEE Network Magazine **12** (1998) 29–36
2. Wetherall, D., Guttag, J., Tennenhouse, D.: ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In: Proc. IEEE Conference on Open Architectures and Network Programming (OPENARCH'98). (1998) 117–129
3. Schwartz, B., Jackson, A., Strayer, W., Zhou, W., Rockwell, R., Partridge, C.: Smart Packets for Active Networks. In: Proc. of IEEE OPENARCH'99. (1999) 90–97
4. Wetherall, D., Tennehouse, D.: The ACTIVE IP Option. In: Seventh ACM SIGOPS European Workshop. (1996)
5. Silva, S., Yemini, Y., Florissi, D.: The NetScript Active Network System. IEEE Journal on Selected Areas in Communications **19** (2001) 538–551
6. Song, S., Shannon, S., Hicks, M., Nettles, S.: Evolution in Action: Using Active Networking to Evolve Network Support for Mobility. In: Proceedings of the Fourth International Working Conference on Active Networks (IWAN'2002). (2002) 146–161
7. Hicks, M., Kakkar, P., Moore, J., Gunter, C., Nettles, S.: PLAN: A Packet Language for Active Networks. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages, ACM (1998) 86–93
8. Hicks, M., Moore, J., Alexander, D., Gunter, C., Nettles, S.: PLANet: An Active Internetwork. In: Proceedings of the Eighteenth IEEE INFOCOM'99, IEEE (1999) 1124–1133
9. Hicks, M., Moore, J., Nettles, S.: Dynamic software updating. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM (2001) 13–23
10. Perkins, C.E., ed.: Ad Hoc Networking. Addison-Wesley (2000)
11. Plattner, B., Sterbenz, J.P.: Mobile Wireless Active Networking: Issues and Research Agenda. In: Proceedings of the First IEICE International Workshop on Active Network Technologies and Applications (ANTA). (2002) 71–74
12. Johnson, D., Malz, D.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Mobile Computing. Kluwer Academic Publishers (1996) 153–181
13. Perkins, C., Royer, E.: Ad hoc On-Demand Distance Vector Routing. In: Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications. (1999) 90–100
14. Perkins, C., Bhagwat, P.: Highly Dynamic Destination-Sequenced Distance-Vectore Routing (DSDV) for Mobile Computers. In: Proceedings of the conference on Communications architectures, Protocols, and Applications (SIGCOMM'94). (1994) 234–244
15. Broch, J., Maltz, D., Johnson, D., Hu, Y., Jetcheva, J.: A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In: Proceedings of the Fourth Annual ACM/IEEE MobiCom'98. (Dallas, TX, Oct., 1998) 85–97
16. Perkins, C., Royer, E., Das, S., Marina, M.: Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks. IEEE Personal Communications **8** (2001) 16–28
17. Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D., Minden, G.: A Survey of Active Network Research. IEEE Communications Magazine **35** (1997) 80–86
18. Hicks, M., Nettles, S.: Active Networking means Evolution (or Enhanced Extensibility Required). In: Proceedings of the Second International Working Conference on Active Networks. (2000)
19. : (Typed Assembly Language) http://www.cs.cornell.edu/talc/releases.html.

20. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A Realistic Typed Assembly Language. In: ACM SIGPLAN Workshop on Compiler Support for System Software. (1999) 25–35
21. : (The Network Simulator - ns-2) http://www.isi.edu/nsnam/ns/.
22. : (The Monarch Project) http://www.monarch.cs.rice.edu/.