

Modular and Adaptive Ad Hoc Routing in Click

by

Audun Tornquist

B.S., South Dakota School of Mines and Technology, 1998

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science

2001

This thesis entitled:
Modular and Adaptive Ad Hoc Routing in Click
written by Audun Tornquist
has been approved for the Department of Computer Science

Dirk Grunwald

Prof. John K. Bennett

Prof. Shivakant Mishra

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Tornquist, Audun (M.S., Computer Science)

Modular and Adaptive Ad Hoc Routing in Click

Thesis directed by Prof. Dirk Grunwald

Several experimental **ad hoc** routing protocols are currently undergoing development. These protocols allow networks to form without any pre-existing infrastructure, between any nodes that are within communication range. In order to fully implement these protocols, extensive testing and modifications are usually needed.

This paper describes an **ad hoc** toolkit built using Click, a modular and extensible framework for creating software routers. I have decomposed a number of **ad hoc** routing protocols into distinct components; these components are easy to understand, simple to modify and can be used in both simulation and deployment. Moreover, these components allow rapid experimentation with different protocol configurations. I will also show how two existing **ad hoc** routing algorithms (AODV and DSR) as well as variants of DSDV and ZRP can be implemented using these components. Finally I present the design of a hybrid router that utilizes elements from all of these protocols to provide a fully adaptive and modular protocol.

Acknowledgements

I would like to thank my advisor Dirk Grunwald for all the excellent ideas and helpful suggestions while working on this project. I would like to thank Mike Neufeld for always helping me out as well as providing so much valuable input in addition to the **ns-2** implementation and simulation data. I would also like to thank Mark Burgess and my parents for their support of my academic career.

Contents

Chapter

1	Introduction	1
1.1	Background and Related Work	2
1.2	Terminology	3
2	Implementing Ad Hoc Protocols	4
2.1	Click	4
2.1.1	Click Router Configuration Scripts	5
2.1.2	Modifying a Configuration	5
2.2	Click Router Elements	5
2.2.1	Ad Hoc Element Classification	6
2.2.2	Packet Annotations	7
2.3	Ad Hoc On-Demand Distance-Vector Protocol	7
2.4	aodv-router.click Configuration	8
2.4.1	Input	8
2.4.2	Output	10
2.4.3	Packet Forwarding	11
2.4.4	Route Discovery	11
2.4.5	Route Maintenance	13
2.4.6	Link Layer Encapsulation	14

2.5	Dynamic Source Routing	14
2.6	dsr-router.click Configuration	16
2.6.1	Input	16
2.6.2	Output	16
2.6.3	Forwarding	18
2.6.4	Route Discovery	18
2.6.5	Route Maintenance	18
2.6.6	'Piggybacking' and diffserv Packet Queuing	20
2.6.7	Variable DSR Packet Lengths	21
2.6.8	Link Layer Encapsulation and Route Expiration	22
2.6.9	Modular Inefficiency	22
2.7	Router Properties	23
2.7.1	Detecting Changes in Network Topology	23
2.7.2	ICMP	24
2.7.3	IP Protocol Numbers	25
2.7.4	Multiple Interfaces	25
3	Extending the Base Protocols	27
3.1	Implementing New Protocols	27
3.1.1	Protocol Properties	28
3.1.2	Generic Ad Hoc Elements	28
3.1.3	DSDV	29
3.1.4	Implementing a Variant of The Zone Routing Protocol	31
3.1.5	Building IARP	32
3.1.6	Modifying AODV/DSR to comply with IERP	33
3.1.7	Adding BRP	35
3.2	Adaptive Hybrid Router	36

3.2.1	Hybrid Router Configuration	37
3.2.2	Shared Routing Table	38
3.2.3	AODV Revisited	39
3.2.4	Choosing a Protocol	40
3.2.5	Analyzing Network Activity	41
4	Evaluation	43
4.1	Simulation Environment	43
4.1.1	Link Layer Feedback	43
4.1.2	Link Layer Collisions	44
4.2	Test Bed Simulations	45
4.3	ns-2 Simulations	45
5	Conclusions	47
5.0.1	Future Work	48
	Bibliography	49
	Appendix	
A	Elements	51
A.1	Ad Hoc Elements	51
A.2	AODV Elements	55
A.3	DSR Elements	60
A.4	IARP Elements	65
A.5	BRP Elements	66

Tables

Table

2.1 Packet user annotations	7
---------------------------------------	---

Figures

Figure

2.1	AODV Input Elements	9
2.2	AODV Output Elements	9
2.3	AODV Route Forwarding Elements	9
2.4	AODV Route Discovery Elements	12
2.5	AODV Route Maintenance Elements	12
2.6	DSR Input Elements	17
2.7	DSR Output Elements	17
2.8	DSR Route Discovery Elements	17
2.9	DSR Route Maintenance Elements	19
3.1	IARP Input Elements	34
3.2	BRP Input Elements - Elements Added to DSR Configuration are Shaded	34
4.1	Comparison of Click implementation to NS implementation	45

Chapter 1

Introduction

As wireless network cards are becoming more common and available in ever smaller and more mobile devices, it's starting to get interesting to put these to use even when there is no pre-configured network available. Several **ad hoc** routing algorithms have been developed in the last few years, but they have mostly stayed within the realm of simulators and very little has been done to produce and release actual, working implementations. In addition, it is often difficult to add functionality to **ns-2** - the simulator most often used in testing and simulating these protocols. There is a considerable threshold in setting up a simulation of a wireless network as well as verifying the test-results with a real-world implementation. This is, obviously, a problem for anyone that want to test the published properties of these protocols in actual use. As a result I have looked at Click [13, 12], a software router that is both fast and easily configurable. Click has also been integrated with **ns-2** by our research group, allowing any single protocol implementation to be tested and simulated as well as directly loaded into the kernel to route packets. Using this framework, I have built elements that implement two **ad hoc** routing protocols - AODV [17, 18] and DSR [11]. A complete router is created by combining these elements into a configuration. I provide a detailed discussion of this modular approach in Chapter 2 and describe the elements used in the overall design.

Having a single implementation consisting of interchangeable elements certainly

makes it easier to test, modify and experiment with the protocol. In fact, any running configuration can be changed in real-time, allowing the router to dynamically decide which elements to use. Imagine, for instance, if we could combine the desired properties of each protocol. One protocol may not support multicast, while another does. Another protocol may offer better performance during high mobility, while another provides less overhead. My approach simplifies the construction of hybrid routers that combine any number of existing components to achieve the desired properties. In Chapter 3 I explain how I extended and combined the existing **ad hoc** elements to create variations of DSDV and the Zone Routing Protocol (ZRP) [6]. Chapter 4 then presents a measurement study with evaluation data from simulating AODV in **ns-2** and compares this with the existing implementation. Performance data from forwarding packets in a small-scale test bed using the Linux routing table as well as AODV and DSR is also provided. Future work and the conclusion follow in Chapter 5.

1.1 Background and Related Work

A recent paper by [3] details the difficulties in building an actual **ad hoc** protocol. The implementation presented only runs in userspace, while other optimizations used in the existing **ns-2** simulations would not work. For instance, **route replies** could not be cached by intermediate nodes as the packets were forwarded by the kernel-level IP router and not by the actual AODV code. A second implementation [14] has similar problems. My implementation resolves all these issues by using the Click framework.

There have been several papers published that compare different **ad hoc** protocols. [15] presents a study comparing performance of DSDV, TORA, DSR and AODV, showing simulated performance and discussing the differences and similarities between each protocol. More recently, [4] compares DSR and AODV with the conclusion that there are significant differences in performance between the two and that each protocol have advantages under different scenarios as well as properties that can be improved.

It's therefore interesting to both build a router that can switch between the existing AODV and DSR protocol specifications, as well as being able to replace elements in each protocol to test and verify that these changes do improve performance.

1.2 Terminology

In this paper the term **router node** refers to any instance of a Click router configuration. An **element** is a C++ class instant of a node in the Click router graph. An element's **input port** is the connection where a packet is received, while an **output port** is where a packet is sent after the element is done processing it. **Push** and **pull** refers to an element's ability to send packets downstream or grab packets from the upstream node, respectively. An element is considered **agnostic** if it supports either push or pull for any of its output or input ports. The following abbreviations are used in describing a protocol's route maintenance functionality: **RREQ** is short for **route request**, **RREP** is short for **route reply** and **RERR** is short for **route error**, **ACK** is short for **acknowledgement**, while **AREQ** is short for **acknowledgment request**.

Chapter 2

Implementing Ad Hoc Protocols

To date, it has been difficult to effectively evaluate and deploy **ad hoc** routing protocols since these protocols are usually designed and specified as monolithic algorithms. In this chapter I discuss the implementation of two complete reactive ad hoc routing protocols; Ad Hoc On-Demand Distance Vector (AODV) and Dynamic Source Routing (DSR). This provides the building blocks for an **ad hoc routing protocol toolkit** built using the Click modular framework.

2.1 Click

Click is a modular software router that is both fast and powerful, while easy to configure and use. It's been extensively tested in both commercial and academic research projects, which is what led me to take a closer look at how this tool could help the development of wireless, **ad hoc** technology.

While Click offers modularity, it's essentially a flat framework. Click only do routing, but it also has to cooperate with the existing network stack, blurring the lines between these two concepts. In fact, the familiar, layered, network might not be the most useful abstraction in this context. As an example, my ad hoc routers share the same routing table across the link layer and the network layer, as well as between different protocols.

2.1.1 Click Router Configuration Scripts

The Click router kit is composed of a number of **elements**; each element performs specific routing tasks, such as encapsulating a packet or reducing the TTL. Each element has one or more inputs and one or more outputs. Each Click router is configured using a routing graph written as a Click configuration file, which defines the “flow” of packets between the different elements. The configuration file specifies how instances of routing elements should be initialized and connected. Finally, the operating-system specific Click interface is responsible for loading the Click configuration file, creating instances of each routing element and interfacing the routing configuration to the underlying network hardware.

2.1.2 Modifying a Configuration

The **ad hoc** router elements can be replaced or removed as long as **packet annotations**, such as the destination annotations, are updated accordingly. For instance, a router configuration that decides not to validate incoming AODV **route request** packets (the **AODVValidateRREQ** element) can remove this element but must insert a **GetIPAddress** element to set the packet’s destination annotation to that of the destination of the Route Request. In other words, **GetIPAddress** is functionally equivalent to the **AODVValidateRREQ** element as far as elements further down the graph are concerned. These requirements are listed in Appendix ?? for each element as “input” and “output” specifications.

2.2 Click Router Elements

Elements are the actual building blocks of each Click router configuration. They implement **simple** router functions like packet classification, processing, queuing and scheduling but are connected together to create complex routers. My current implemen-

tation introduces some 56 elements that provide a toolkit for further research, experimentation and implementation of ad hoc protocols. These can further be divided into 7 different categories: AODV and DSR (which compromise the majority of elements), BRP, IARP, generic ad hoc elements (e.g. **GetNextHop**) and ad hoc information elements (e.g. **RoutingTable**). The implementation also makes use of a fair amount of stock Click elements to do input/output, IP processing, packet classification, etc. Looking at the relationship between the different elements in a configuration (i.e. how they're connected) is the key to understanding how a protocol work. In fact, this relationship provides a graph that is easy to understand, validate and make changes to. The following subsections provide more details as to how these elements are integrated.

2.2.1 Ad Hoc Element Classification

I extended the existing Click tools by constructing a set of routing components that support the AODV and DSR routing protocols. While each element provide basic processing functionality, it's useful to group these elements together into higher-level components that are needed for **ad hoc** protocols. In fact, we can essentially divide each routing configuration into 5 parts:

- Input; receive packets from input device(s) and do initial packet classification
- Output; unicast or broadcast packets to output device(s)
- Forwarding; find the next node to forward the packet to
- Route maintenance; handle ad hoc protocol control packets and update the routing table
- Route discovery; route requests (reactive routing) or link state announcements (proactive routing)

2.2.2 Packet Annotations

A limited amount of state data can be associated with each packet with the use of **packet annotations** that is sent, along with the packet data, between each element. Some of this is allocated by Click for exclusive use by destination address, header pointers, etc. In addition, 12 bytes are available as **user annotations** for use by any element in the configuration. I have allocated the following annotations for the *ad hoc* routers:

Table 2.1: These are the allocated **user annotations** used in the *ad hoc* router protocols.

<i>offset</i>	<i>description</i>
0	paint (interface number)
1 - 7	packet link-layer source address (i.e. mac source)
8	<i>adhoc</i> header option offset
9	<i>adhoc</i> header offset

2.3 Ad Hoc On-Demand Distance-Vector Protocol

AODV is a **distance vector protocol**, but unlike similar protocols (such as DSDV, on which it is built), it uses **on-demand routing**. AODV has been shown to be a simple algorithm that works well in **ad hoc** environments [4]. It's therefore a good starting point for implementing new protocols in Click. AODV is a **reactive** protocol because it only sends routing messages when a route is not known; a **proactive** routing algorithm (such as DSDV) sends periodic route updates and maintains routing entries for all nodes, even in the absence of other communication. In AODV, a **route request** message contains a query for a route to a single destination. **Route discovery** is started when the the RREQ message is sent to all neighboring nodes. If those nodes have a route to the destination (or they are the destination) they reply using a **route reply** message. If the intermediate nodes do not have a route, they forward the RREQ

message if the TTL field is greater than zero.

Once the source receives the RREP message, it can begin sending data packets to the destination. To reduce network-wide broadcasts, the source node sends requests in an “expanding ring”, gradually increasing the TTL of the RREQ messages. The RREP messages are returned to the source node using the same route taken by the original RREQ message; intermediate nodes establish a routing entry containing the destination address, IP address of the next hop and the distance to the destination as well as a timer that is used to expire unused routes. A node will forward the first RREP it receives and any later RREP that provides a better route or have a higher sequence number.

Routes are **active** as long as they are periodically used. When a node receives a packet for which it no longer has an **active** route or if a next hop node moves away, a **route error** (RERR) message is used for **route maintenance**. The RERR message is sent to the source node to indicate that a new route request should be started. The RERR messages are returned using a return route, and intermediate nodes delete their routing entry for that destination to limit further forwarding using the broken route. Before issuing the RERR, however, a node that is unable to forward a data packet can initiate a **route discovery** and try to locally repair the broken link.

2.4 aodv-router.click Configuration

It is difficult to understand the complete AODV router by examining the full configuration, so I have decomposed the graph into sections corresponding to the **input**, **output**, **forwarding**, **route discovery** and **route maintenance** components. The following sections describe the the most important elements within these components.

2.4.1 Input

The input section of the AODV router is shown in Figure 2.1. Input arrives either from the various Ethernet interfaces (**FromDevice**, to the left) or the Linux

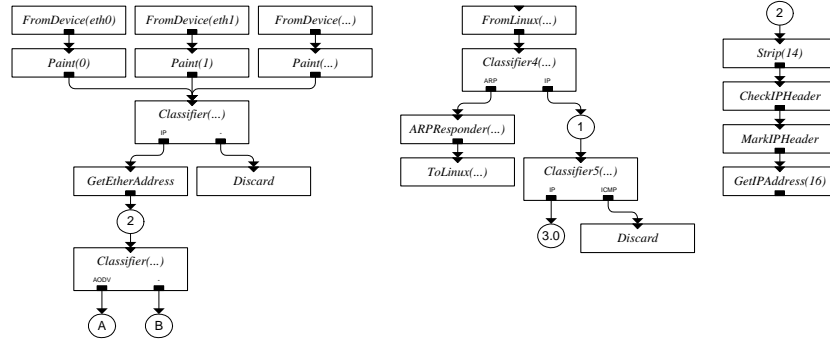


Figure 2.1: AODV Input Elements

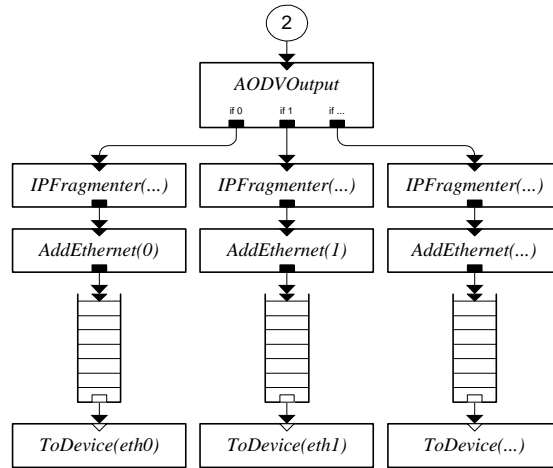


Figure 2.2: AODV Output Elements

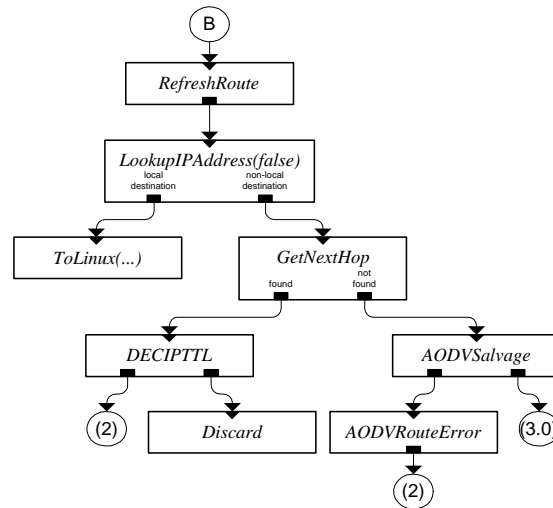


Figure 2.3: AODV Route Forwarding Elements

kernel (**FromLinux**, in the middle). When packets arrive from a network interface it's "painted" with a **user annotation** to indicate the input interface – this is used in later routing decisions. Packets from interfaces are then classified (**Classifier**) into IP or other packets. This configuration assumes a pure AODV network and thus discards all other traffic, including ARP requests and responses since they are not needed. Following this, we extract the Ethernet MAC address from the packet and then perform a common transformation (shown expanded on far right) that removes the Ethernet header (**Strip**), checks for a valid IP header (**CheckIPHeader**), extracts and caches information from the IP header (**MarkIPHeader**) and extracts the source and destination IP address (**GetIPAddress**). These steps prepare the incoming packet for further processing; the next step classifies a packet as an AODV control packet, in which case it is sent to the route maintenance code, or as a data packet, in which case it's sent to the forwarding code.

Packets received from the kernel follow a simpler sequence: they are classified as IP or ARP packets, and all IP packets follow the same sequence as transiting packets (**Strip** → **CheckIPHeader** → **MarkIPHeader** → **GetIPAddress**). Although AODV does not use ARP, it's possible that unsolicited ARP responses are sent by the Linux kernel.

2.4.2 Output

Figure 2.2 shows a common packet output sequence. The **AODVOutput** element will send a packet either out a particular interface (as specified by the routing table for the next hop destination), broadcast the packet to all interfaces (e.g. **AODVhello** messages) or send it to selected interfaces for **route errors**; it uses the IP packet type and destination to determine what should be done. Packets are fragmented if they're larger than the output MTU, then encapsulated with an Ethernet header based on the source address of the output interface and the destination address of the next hop.

Before being sent out, packet(s) are queued for the different interfaces. The **ToDevice** elements are the only “pull” elements in the configuration.

2.4.3 Packet Forwarding

Packet forwarding, shown in Figure 2.3 is responsible for taking an existing packet and sending it to the local node or forwarding it to the next routing hop (or gateway). In fact, packets routed with next hop forwarding fall into its own category of which AODV and DSDV are subcategories and for which a common, single routing/forwarding sequence is sufficient. As packets are forwarded, the first step in this process is to set the “last used” time for routes specified by the source and destination in the packet (using **RefreshRoute**) to keep the routes in the route cache. After this, the destination IP address is extracted and the packet is either delivered to the local node or the next hop address is selected (**GetNextHop**). This involves a lookup in the routing table; if a route is present, the IP TTL of the packet is decremented and the packet is sent to the output section. If there is no route, a **route error** message is returned to the source unless we decide to try to “salvage” the packet and send it to our **route discovery** component. In other words, assume node *A* is trying to send a message to *Z* via node *B*. If *B* receives the message but it does not have a route to *Z*, it will start a **route discovery** for *Z* to try to locally repair the route. Node *B* queues the packet until it receives an appropriate **route reply** or the **route request** times out, in which case node *B* should return a **route error** to the source node and drop the data packet.

2.4.4 Route Discovery

Route discovery, shown in Figure 2.4, is more complex than it may first appear. As packets arrive from the kernel, the **AODVRouteRequest** element asks the routing table to queue the packet in a “send buffer” until a route becomes available or the packet time limit expires. The element can handle both data and AODV control packets. For

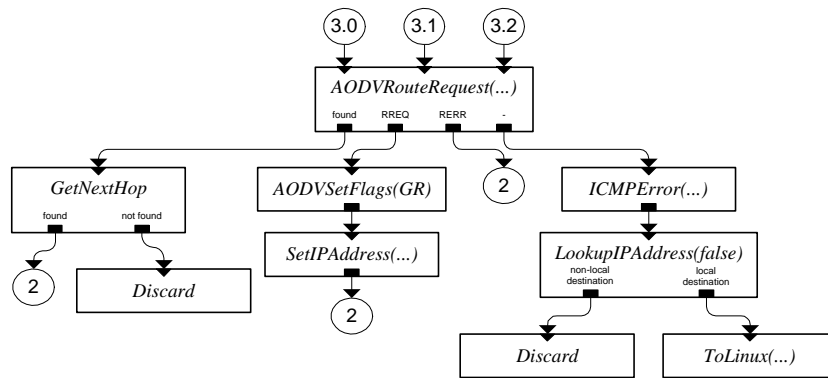


Figure 2.4: AODV Route Discovery Elements

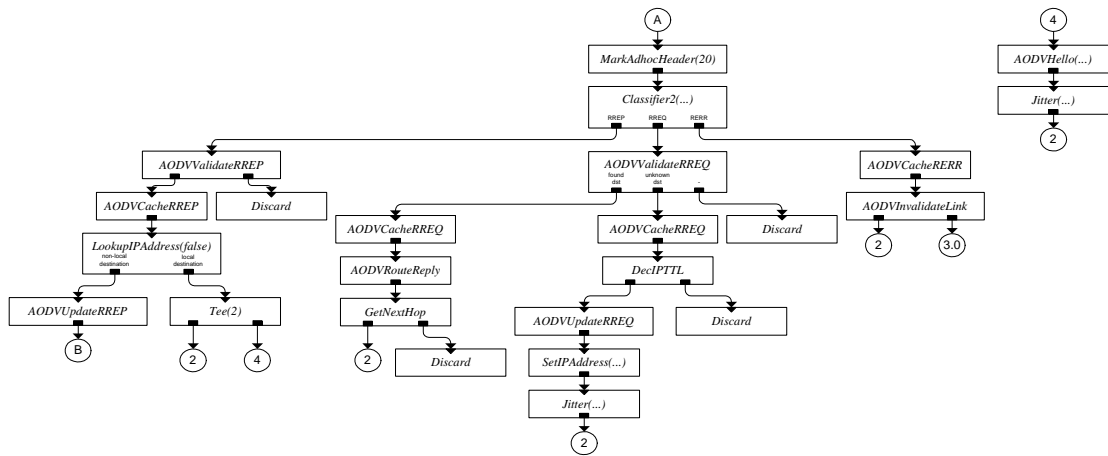


Figure 2.5: AODV Route Maintenance Elements

example, incoming RERR packets cause RREQ retransmissions to find a new route to the destination if there is a waiting packet in the send buffer. Once **route replies** are processed and routes are established, queued packets waiting for those routes are removed from the send buffer, the next hop address is found and the message is output. **Route requests** are marked as broadcast packets and then sent to the output.

2.4.5 Route Maintenance

Figure 2.5 shows route maintenance, the last and the most complicated section in the AODV router. This section only processes AODV control messages. The first step is to mark certain parts of the AODV header (**MarkAdhocHeader**) so they can be quickly located by later elements; the messages are divided into RREP, RREQ and RERR flows. Route responses (on the left of Figure 2.5) destined for the local node are cached (**AODVCacheRREP**) and sent to **AODVRouteRequest**. the local machine and then the RREP record is updated (**AODVUpdateRREP**) to set the next-hop address to that of the current node; they are then sent to the output. Route responses are discarded by the **AODVValidateRREP** element if a better route has already been found.

Route requests are handled by the middle portion of Figure 2.5. **AODVValidateRREQ** classifies the **route request** into three categories; if the RREQ has been seen before, it is discarded. Otherwise, **route requests** for known destinations are cached by **AODVCacheRREQ**,¹ and a route reply is generated (**AODVRouteReply**) and sent to the next hop of the reply path. Requests for unknown destinations are also cached; then the TTL of the RREQ is decremented and the RREQ is either updated and broadcast on all interfaces, or it is discarded if the TTL hop count has expired.

Route errors are the last control message; these simply update route information

¹ i.e. to avoid multiple responses if the host receives another copy of the RREQ

from the sender of the RERR message and then invalidate entries in the routing table that used the link specified in the RERR message. Route error messages are then sent to the **AODVRouteRequest** element in route discovery; this may cause new RREQs to be generated.

There are some elements not shown in these graphs. In Click routers, routing tables don't actually accept and forward packets; they are instantiated and a different communication mechanism is used since the use of the routing table is so pervasive. I created a single routing table class that supports the AODV and DSR protocols.

2.4.6 Link Layer Encapsulation

For a node to be able to forward packets, it has to know the MAC addresses of the nodes within communication range that we wish to forward packets to. To cache and associate destination addresses with the actual MAC addresses of neighboring nodes, we can use the IP header source field of any AODV control packet (RREQs, RREPs or RERRs), which is updated by each intermediate node. Data IP packets use the source and destination addresses of the two end-nodes communicating, which may be separated by several hops, so that the Ethernet header of the incoming packet may not match the IP source address if the packet was forwarded to us. The use of hello messages, then, keeps any node up to date on its neighbors and corresponding MAC addresses. AODVHello sends out these messages at a specified interval plus a randomized jitter delay to avoid repeated packet collisions. The interval can also dynamically change to accommodate higher mobility in the net (see NetMonitor).

2.5 Dynamic Source Routing

The DSR protocol needs complete source routes when routing packets through the network and therefore requires a little more bookkeeping at each node. The full protocol supports unidirectional links; my implementation was designed for an 802.11b

network which supports bidirectional links, so that I can always “reverse” routes to get a path back to the source.

Like AODV, DSR is **reactive** and **on-demand**, sending out a **route request** (RREQ) message when a route is not available. Unlike AODV, each RREQ message contains the full route from the source to the intended destination (or **target**), including all intermediate nodes and a unique RREQ identifier. When the target receives the RREQ, it replies with a **route reply** (RREP) message indicating the full route. Intermediate nodes discard duplicate incoming RREQ messages or requests that already contain their node addresses; otherwise, they append their own address to the route record and forward the request.

Since DSR was designed to support unidirectional links, the target would normally route the RREP using a route from its own route cache; this may elicit a route discovery. In a bidirectional network, the target can use the reverse route from the RREQ message if it has no existing route. When the source receives a RREP message, it caches the full route for future messages.

Packets awaiting routes are stored in a **send buffer** and discarded after a specified period of time (or when too many packets are allocated). A node periodically re-initiates route requests for packets in the send buffer, again in an “expanding-ring” fashion, backing off exponentially.

A node forwarding a packet should look up the next hop in the source route header. If the specified next hop is no longer a neighbor, the intermediate node can attempt to salvage the packet by replacing the existing source route with a valid route from its own cache. If a message can still not be delivered, a **route error** (RERR) is generated. The RERR message is sent to the packet source and causes the cached route to be removed. The source can then either resend the packet using an alternate cached route or initiate another route discovery.

Intermediate nodes can cache routing information they learn while propagating

routing information, including information from RREQs in bidirectional networks. Intermediate nodes can respond to a route request using information from their own route caches as long as the resulting route contains no duplicate nodes. Those cached routes must then be invalidated by any RERR messages intercepted by the intermediate node.

2.6 dsr-router.click Configuration

The DSR implementation is slightly more complicated than the AODV implementation which is reflected by the somewhat larger Click configuration. This is mainly because I allow DSR to piggy-back multiple headers on a single packet. I also use explicit **acknowledgement** messages to confirm packet delivery; **acknowledgements** can then be piggy-backed on other DSR messages. The process of adding, removing or updating the DSR source headers further complicates the process but is explained in more detail in the following sections. I have again divided the router configuration into several subgraphs to reflect the basic components so that it's easier to follow the discussion.

2.6.1 Input

The input sequence is very similar to that used in AODV. Fig. 2.6 shows how Ethernet packets are routed at the top level as they arrive at a node. The different **Classifier** elements categorize incoming packets as local, non-local or broadcast Ethernet packets and further as IP or DSR packets.

2.6.2 Output

Packets created by the local node will always pass through the processing sequence macro shown to the right in 2.7, which will add source route and acknowledgement request header options based on the next hop for the packet's destination. Packets that are forwarded will typically already have the appropriate headers in place and can simply

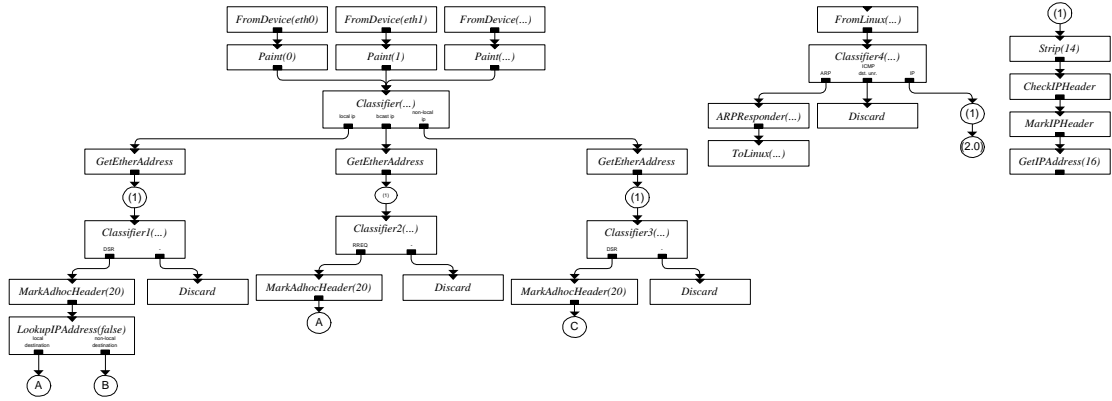


Figure 2.6: DSR Input Elements

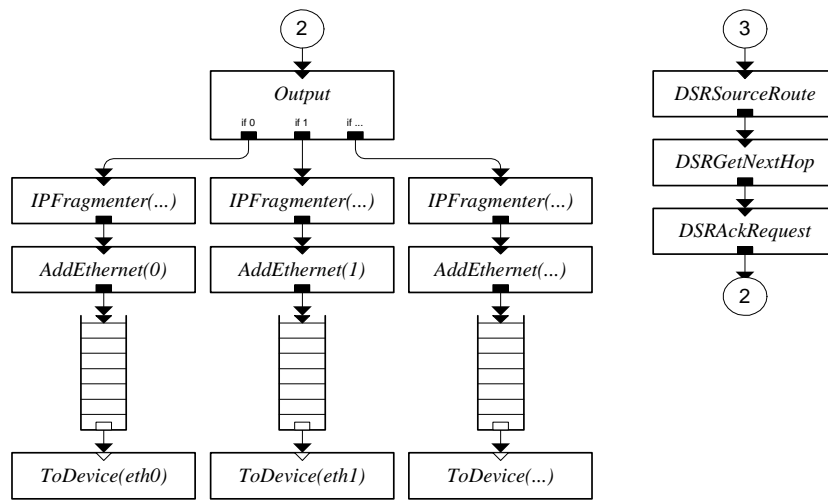


Figure 2.7: DSR Output Elements

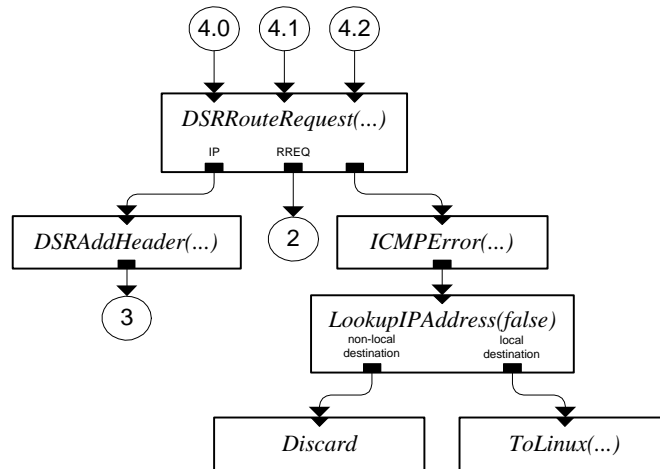


Figure 2.8: DSR Route Discovery Elements

be output to the sequence of elements on the left. DSR cannot fragment packets, so packet sizes exceeding the configured MTU will not be output by the router (see 2.6.7). The **Output** element used by DSR is more generic than its AODV counterpart since no special action is needed to process **route errors**. Broadcast packets are simply output to all interfaces, while unicast packets are only sent to a single interface.

2.6.3 Forwarding

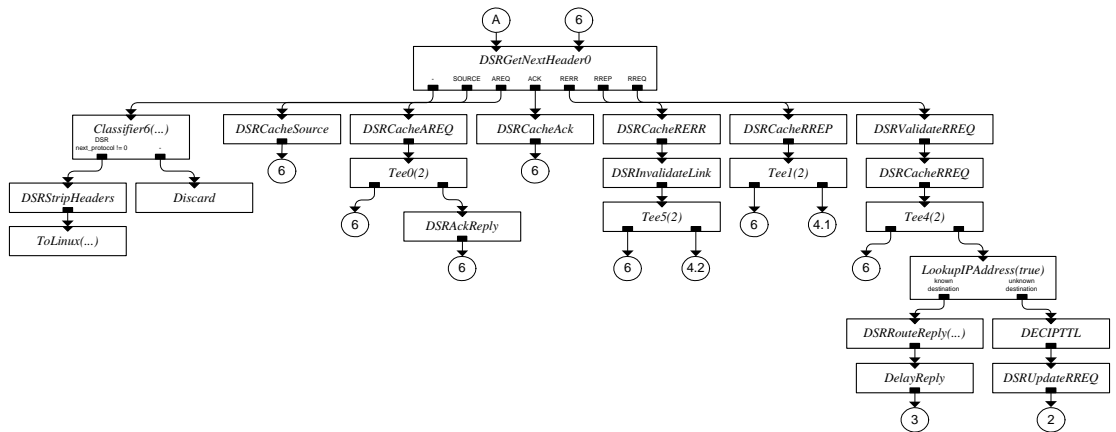
Forwarding is shown in Fig. 2.9 (b), which handle non-local IP packets. Local data packets are handled separately by 2.9 (a), where incoming packets are stripped of their DSR headers and handed over to the kernel. Non-local packets are handled much the same way as with AODV, with **DSRGetNextHop** finding the next hop destination (but from the source route header instead of the routing table), decrementing the TTL and sending the packet to output. If the next hop is no longer a valid destination, a **route error** is sent back to the source, but the node will also try to replace the packet's source route with a valid route if possible.

2.6.4 Route Discovery

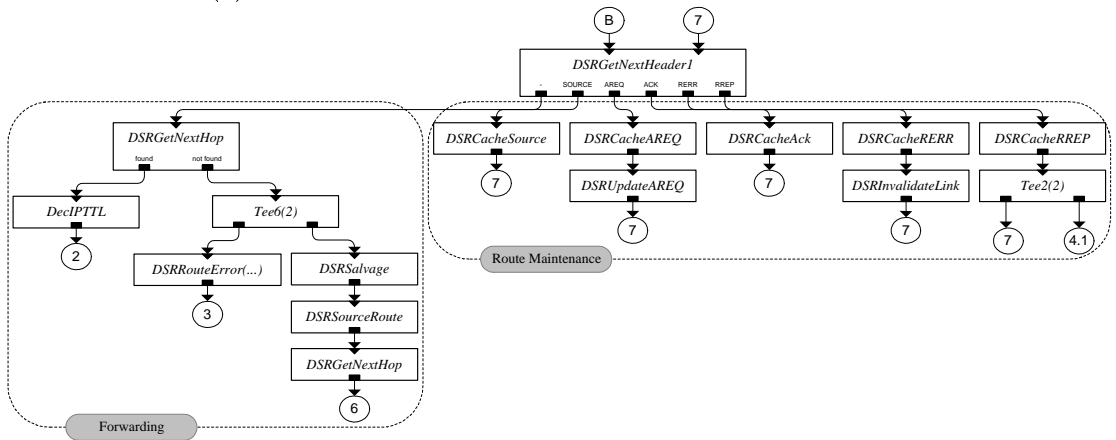
Figure 2.8 shows the route discovery process for DSR. This is structurally similar to the AODV route discovery process shown in Figure 2.4; the **DSRRouteRequest** is similar to **AODVRouteRequest** and encapsulates the core of the routing algorithm as well as the “send buffer” that caches packets with no available route.

2.6.5 Route Maintenance

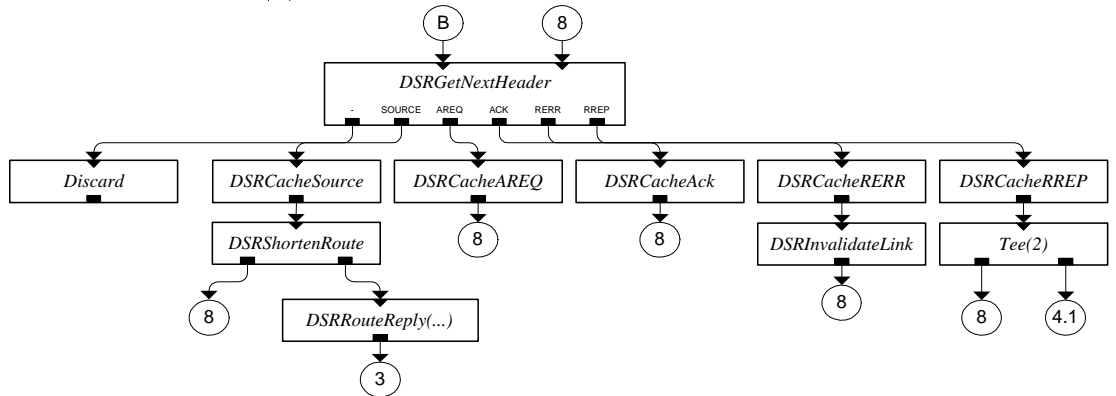
Route maintenance for DSR is significantly more complicated than that for AODV; Figure 2.9 shows the full configuration, split into three subsections for routing packets destined for the local machine, forwarded packets and any packets that may be “snooped” using a promiscuous interface. There are elements in DSR that correspond



(a) Route Maintenance For Packets For Local Machine



(b) Route Maintenance For Forwarded Packets



(c) Route Maintenance For Promiscuous Packets

Figure 2.9: DSR Route Maintenance Elements

almost one-to-one for the AODV elements, such as updating RREQ's and RREP's, updating route caches and the like. DSR caches routes found from incoming packets more aggressively compared to AODV. Overheard packets (i.e. non-local packets that we can see in promiscuous mode) are processed so that additional routes can be cached or invalidated. This is also used to actively help shorten other node's routes to us by sending out RREPs when possible. Routes are cached based on the assumption that the link is bidirectional, which will always be the case when using 802.11b.

2.6.6 'Piggybacking' and diffserv Packet Queuing

My DSR implementation allows header options to be piggy-backed with other options in the same packet. In fact, as an example of the advantages of the modular approach of the Click router configuration graph, it's quite easy to see how this is handled (Fig. 2.9): The **DSRGetNextHeader** element will parse the next DSR header option in the packet and allow that to be processed before the packet again loops back into **DSRGetNextHeader**. This allows for any number of header options to be associated with a packet, although it only allows for one IP header. For instance, it would make sense to piggyback ACKs on packets going back to the source, as traffic is likely to be bidirectional.

For now, only source nodes piggyback RERRs in the subsequent RREQ, which helps propagate news of broken links to other nodes so that they can clean up their route caches [2]. We could also do this for overheard or forwarded RERRs. This optimization is handled by the **DSRRouteRequest** element, and no other elements that create packets piggyback additional headers in the current implementation, but this should be a simple modification if needed. One use for this could be to save up **route errors** or **route replies** (since **route requests** are broadcasted, it makes less sense to piggyback these onto other unicast packets) for a given amount of time and attach them to any outgoing packets with a common (possibly intermediate) destination. This

could again be coupled with a QoS **diffserv** implementation that would decide whether to try to piggyback headers and save power on fewer transmissions vs. packet latency. This could also include delaying **route replies** to see if anyone else replies with a better route, which is already specified in the DSR draft. Specifically, the **Output** element could easily be used to piggyback queued headers on other packets.

2.6.7 Variable DSR Packet Lengths

The last complication in DSR is error reporting due to changes in packet size, resulting in packets larger than the network MTU. Since packets routed with DSR need to have the full source route appended, the final, outbound packet length may vary according to the path for one destination over time, and for every different destination. In addition, the router can add a varying number of packet headers, such as route errors or acknowledgements. These additional headers decrease the space available to the packet payload. As a result, the actual MTU available to underlying protocols will also (unpredictably) vary. In the middle of a packet flow, the selected route to a given destination may increase the required number of hops, decreasing the current MTU. DSR cannot fragment packets and has to drop anything that exceeds the MTU. Normally, this only affects the packet source, since the source specifies the full route. However, DSR allows packets to be “salvaged” **en route** if a node determines that a link is broken but it knows of an alternate route. This added process increases the complexity of handling the ICMP responses for MTU discovery. In this implementation, an ICMP error 3, code 7 is generated by the output component and returned to the kernel, so that underlying protocols have the opportunity to decrease packet length. How much impact having to send this ICMP message to change the MTU whenever the hop count increases will have on performance, is not yet clear. In addition, if the hop count should decrease and leave more room for additional packet data, there is currently no way of notifying underlying protocols to take advantage of this and start sending

larger packets.

2.6.8 Link Layer Encapsulation and Route Expiration

To learn link layer addresses when running a DSR router, we have to somewhat modify the approach used with AODV. It's now only possible to learn the corresponding MAC address of an IP address if a packet is forwarded to us by looking at the recorded source route, from the source field in RREP packets which is updated by each intermediate hop or when other control packets arrive from hosts one hop away. The DSR draft does not specify whether nodes should also incorporate **hello** messages to continuously detect any changes in network topology to facilitate route maintenance. Instead, route errors are only detected when forwarding packets. This being the case, however, and if routes are not time out, it should suffice to learn the MAC address of the next hop from a route reply, as we cannot send anything until a route request and a subsequent reply is received for any given destination anyway. The source route path will remain valid after a route has been established during route discovery until a route error is received or an acknowledgement request times out. If the routing table is configured to time out source routes, we can refresh each node along the path as long as there is traffic, using the RefreshSource element.

2.6.9 Modular Inefficiency

Processing and adding all packet headers and data in one, monolithic element is the common approach to network programming and is usually more efficient than multiple, independent operations (e.g. allowing for a single memcpy). To avoid multiple memcpy's and letting Click allocate packet data space with only one Packet::make, the following elements allow for optional parameters, that (with care) can be used to achieve the efficiency of a single element: **DSRAddHeader(SOURCE, TAILROOM)** will calculate and allocate enough space for a complete source route if SOURCE is true,

while adding TAILROOM bytes to the end of the packet. **DSRRouteError(...)** and **DSRRouteReply(...)** work similarly. **DSRACKRequest** can then rely on the configured amount of bytes specified by TAILROOM to already have been allocated behind the other DSR header options. This does require the configuration script to set up the elements correctly, but in return, greatly reduces overhead. It does however, also require the elements to calculate the exact amount of packet space needed for any additional headers, which in return translates to an additional table look-up to find the needed number of hops.

2.7 Router Properties

The following sections discusses some of the issues that arose while implementing AODV and DSR, but are problems that are generally independent of the actual routing protocol used.

2.7.1 Detecting Changes in Network Topology

Though also available at the 802.11b link-layer, I use two different approaches to updating link connectivity information and detecting changes in (neighborhood) network topology. The AODV draft allows for hello messages to be broadcast from every node at a given interval. These are in fact just **route reply** messages with a time to live of hop count 1 that are used by each node to refresh the routing table and associate that next hop with a link layer address (i.e. the interface MAC address). We will later see that the **AODVHello** element can also be replaced by an **IARPHello** element that proactively floods a predefined zone radius with neighborhood link status information. The DSR implementation uses a header option called an ACK request to ask for an ACK message back from the next hop. This has the advantage of not continuously flooding the network with hello broadcasts, but is more processor-intensive as every new packet has to have this header added and each forwarded packet will have to be updated. In

addition, once discovered, routes should not time out since there is no way of updating them when there is no traffic. This leads to a much higher level of stale cache entries in the DSR implementation. The DSRACKRequest element times out any next hop destinations that do not return an ACK within the configured amount of time, but it does not try to retransmit packets as this is already handled by the link layer. This also means that we do not need to keep track of the ACK identities, as specified by the draft. Again our modular implementation proves useful, if nothing else to experiment with these approaches. Unfortunately, since the DSR approach requires a full set of DSR headers to work, AODV cannot simply use the DSR elements instead of **hello** messages (AODV could send out separate DSR ACK request messages, but that would be terribly inefficient). However, as AODV and DSR share the same routing table, DSR can be configured without the ACK Requests and instead use **hello** message broadcasts in a hybrid router configuration (see ??). Figure ?? show the AODV **hello** element sequence on the right.

2.7.2 ICMP

Click already have elements to handle ICMP, so generating these errors becomes quite straight forward. The only question is when to actually generate these errors. Application-layer software should most likely not be notified of (possibly temporary) network partitioning/link breaks, because unlike in a non-ad hoc network, unreachable nodes may become reachable again in a very short amount of time. As soon as a link is broken, route discovery is again activated and a new route (if available) will be found, or the unreachable node in question might have moved back into transmission range. The AODV and DSR routing mechanisms will learn of route breaks from Route Error messages and do not need the additional ICMP messages to work. Some applications, however, could benefit from this additional information since ICMP does have interesting properties. As an example, ping would report a "destination unreachable" for a

destination address for which an ICMP error is generated along with the **route error**. Without the ICMP element, ping would simply report the increased round-trip time (if the unreachable node becomes reachable again before ping times out) reflected by the new route discovery. Ping would then not report a “destination’ unreachable” until the router decides to give up on route requests. In this second case, the router has to generate an ICMP error as shown in Figure ??, which is absolutely necessary to give applications required feedback on network problems.

2.7.3 IP Protocol Numbers

My implementation uses IP protocol number 200 and 210 for DSR and AODV respectively, protocol number 220 for additional BRP headers and protocol number 230 for IARP link state announcements. There are no official protocol numbers defined for these protocols as of yet, so these are just temporary. If the kernel receives a copy of the packet, as is the case in user-level use of the Click FromDevice element, the kernel will generate an ICMP error 3, type 2 for unknown protocol and try to return it to the IP source. These messages are filtered out by the current implementation of the DSR and AODV configuration scripts since they are not really relevant, generate lots of noise and do not pose a problem when the routers are run in the kernel. If these ICMP errors are generated as a response to a different, unknown protocol, however, they should be forwarded, which is currently not the case.

2.7.4 Multiple Interfaces

The ad hoc toolkit elements all support nodes with multiple interfaces. This is, probably, most useful in a test bed-situation where a single machine may simulate several nodes with several, wired interfaces or a wireless node testing out different wireless cards and/or communication ranges. It is also useful when a node has both a wired and a wireless connection. As both AODV and DSR require MAC addresses to

correctly associate with the interface actually receiving a packet, a node with multiple interfaces of which it uses only one interface to communicate with a particular node has to associate the MAC address of the interface used to transmit for all the node's IP addresses. E.g. node A is both 192.168.1.100 00:00:00:00:00:01 and 192.168.1.101 00:00:00:00:00:02. When node B pings IP address 192.168.1.100, it may receive back a reply from IP source 192.168.1.100 but with MAC source address 00:00:00:00:00:02. This is done so that interfaces are not **required** to operate in promiscuous mode and can drop any traffic not bound for that interface, but can still act as a gateway to all the node's IP addresses.

Chapter 3

Extending the Base Protocols

In the previous chapter I decomposed a number of **ad hoc** routing protocols into distinct components with the primary goal of providing a toolkit with elements that could then easily be modified and combined to experiment with different routing algorithms. This chapter takes a closer look at how we can experiment with these components and build new protocols, illustrating the flexibility of using the Click modular framework for **ad hoc** algorithms. I first extend the design of the AODV routing implementation to support the DSDV routing algorithm.

I then show how the routing components can be combined with additional elements to build an implementation of the Zone Routing Protocol (ZRP), specifically a proactive routing protocol for Interzone Routing (IARP) [9], as well as a variant of the Bordercast Routing Protocol (BRP) [7].

3.1 Implementing New Protocols

When implementing a new protocol in Click, it is again useful to look at how we can categorize **ad hoc** routing protocols. AODV and DSR both implement general elements that could easily be used in other routing protocols. The RoutingTable, GetEtherAddress, LookupIPAddress, GetNextHop, Log, MarkAdhocHeader, RefreshSource, DelayReply and Output elements all do generic operations on the shared routing table without any code specific to AODV or DSR. The remaining **ad hoc** elements

do route discovery and process packets based on specific fields in the AODV or DSR headers but are useful as blueprints for new protocols that implement different route discovery packet headers, but similar functionality. This section describes a brief overview of how one would convert the AODV implementation to build a DSDV router, what components can be reused and what changes need to be made.

3.1.1 Protocol Properties

Looking at the DSR and AODV configurations it's quite obvious that they share a fair amount of similar elements. Input processing is identical, after which packets are classified according to the **ad hoc** headers following the required IP header. At this point, the two protocols handle their own specific headers differently, with different implementations of our 5 main components (**input**, **output**, **forwarding**, **route maintenance** and **route discovery**).

It's also useful to categorize a protocol by looking at what elements give it certain properties. Again, a protocol that wants a next hop approach to routes will be able to utilize the same, identical forwarding sequence. A reactive protocol will use a variation of the **AODVRouteRequest** or **DSRRouteRequest** elements to queue packets as they arrive from the kernel and initiate route requests on demand. A protocol's ability to detect changes in network topology is usually implemented with elements similar to **AODVHello**, but translates to a proactive protocol in the case of **IARPHello** (see 3.1.5).

3.1.2 Generic Ad Hoc Elements

The AODV router uses a larger number of generic elements (i.e. elements that are not specific to AODV but may depend on generic information elements such as the routing table and are therefore reusable for other **ad hoc** protocols), simply because proprietary packet headers are needed only when doing route discovery and not when

actually forwarding packets. DSR could also be configured to use more of these elements; for instance, the **GetNextHop** code looks up a packet's destination annotation in the routing table, finds the next hop and replaces the destination annotation with this hop. This will also work with DSR since the source route associated with this packet should have already been cached during the corresponding route discovery process. However, according to the DSR draft, a node should not depend on the cache but instead use the actual source route entry found in the packet header (which might be different from the current route cached at this node). In addition, a node should check to see if the number of remaining hops recorded in the source route has reached zero and decrease the current number of segments left. In other words, a separate **DSRGetNextHop** is needed to comply with the draft specifications, but not necessarily required.

3.1.3 DSDV

The destination sequence distance-vector (DSDV) routing algorithm is an earlier variant of the AODV algorithm [16]. Like AODV, it is a distance vector algorithm, but it is a **proactive** routing algorithm that provides a constant, timed distribution of the routing table of each node to neighboring nodes. Routing updates either use a “full update” or a “partial update”; the assumption is that full updates should be infrequent and that most node mobility will be captured by partial updates.

AODV and DSDV basically differ in only one area: AODV is reactive while DSDV is proactive, broadcasting route announcements to neighboring nodes. With DSDV we no longer want to send out route requests if a route is unknown - packets should now simply be dropped. While additional protocol control messages such as route errors could certainly be used, DSDV is assuming that this is unnecessary since proactive route advertisements will give each node a complete view of the network topology. The route discovery mechanism in AODV is then replaced with the use of these advertisements, and we can disable the **AODVRouteRequest** element to remove

the reactive properties. We still need to add the proactive functionality, however. There are several pre-existing elements in our AODV configuration that we can modify and use for this purpose; the **AODVHello** element periodically broadcasts Route Replies while **AODVCacheRREP** updates the routing table based on the routes found in the incoming packet. The DSDV algorithm specifies that every node should flood the network with their **entire** routing table and also use more frequent, incremental changes when topology changes are noticed. To do this, we needed to modify AODV to have a header denoting multiple route entries in a single packet in place of the AODV Route Reply header. I modified **AODVCacheRREP** to handle multiple routes and modified the **AODVHello** element to let us specify how many routes to announce in each packet. Other route maintenance elements in the original AODV configuration are no longer needed.

In DSDV, nodes should send a route update message when they receive a better route advertisement; however, protocol designers have suggested delaying the output of route updates to allow other route advertisements to arrive, reducing the total number of route updates. We implemented this by using two existing Click modules: **FrontDropQueue** \rightarrow **DelayUnqueue**. The **FrontDropQueue** element queues packets until the maximum queue size is reached; following that, the front of the queue is dropped and the new incoming packet is added to the queue. The **DelayUnqueue** element uses Click timers to wake up and “pull” an element out of the **FrontDropQueue** and push it through the rest of the router. This results in periodic router updates without our having to explicitly add that feature to the route announcement code. These changes complete our DSDV router.

Overall, the implementation of DSDV is simply to: modify the Click configuration graph to add and remove elements from the AODV configuration and to modify the **AODVHello** and **AODVCacheRREP** to support multiple routing advertisements in a single message. I used the **FrontDropQueue** and **DelayUnqueue** elements from

the existing palette of Click elements as a performance enhancement as suggested in the literature. If we were interested in improving DSDV, it is easy to add more enhancements. For example, Perkins [16] suggests that routing advertisements should be broadcast more often when neighbors move out of range while being careful to avoid storms of broadcasts and overly rapid changes in routing data. We could use a traffic shaper element to queue all routing advertisements, thus effectively limiting the route advertisement rate. Most importantly, **I can configure and deploy these changes in minutes**, reducing the time it takes to experiment with different routing enhancements.

3.1.4 Implementing a Variant of The Zone Routing Protocol

To show how the Click framework and our initial element building blocks can greatly ease the task of implementing new ad hoc protocols, this section describes how the Intrazone Routing Protocol (IARP) is combined with existing ad hoc elements to create a whole new router. IARP is the proactive part of the Zone Routing Protocol (ZRP), which also requires an Interzone Routing Protocol (IERP) [8], that is reactive and used outside our zone of known nodes (ZRP is a good example since its approach is already quite modular and easily configurable). ZRP allows for the Bordercast Routing Protocol (BRP) to replace the existing broadcast-flooding used to send **route requests** in existing reactive protocols. Since we already have two reactive protocols running, we can simply use IERP with either DSR, AODV or a combination of the two and add additional BRP elements to the configurations. We could use DSDV as the proactive router, but instead implement the simple protocol described by the IARP draft, since we only need the limited functionality of proactive link announcements within our zone (note that AODV also has a similar proactive part with effectively a R-hop radius (or TTL) of 1, as set by the **AODVHello** route reply messages). The following are the basic features of the combined ZRP routing protocol:

- Limit proactive routing table broadcasts to a radius of R hops (IARP)
- Direct route requests (IERP) outward from our zone by assigning a bordercast router (BRP)
- Use knowledge of our zone topology to "repair" link failures and shorten routes (IERP)

The first requirement is covered by our IARP routing protocol, the second and third requirements forces us to modify our existing reactive protocol(s). For the remainder of this discussion, I will use our DSR implementation as the reactive element of ZRP, since BRP requires source routing to efficiently build bordercast (multicast) trees to guide route requests outwards.

3.1.5 Building IARP

Going back to viewing our **ad hoc** router as compromising 5 different subgraphs at the configuration level, we already have the input and output components of IARP. Since IARP is proactive, we no longer need the route request component, leaving us to implement route maintenance and forwarding. Forwarding is, effectively, covered by the IERP router, which in our case is DSR. We do, however, have to add route maintenance elements to process and cache incoming link state announcements from neighboring nodes within our zone and send out our own announcements.

IARP requires each node to periodically broadcast its set of current neighbors but not the complete routing tale as is the case with DSDV. This routing information is then forwarded within the zone by intermediate nodes until the announcement's TTL expires. To create these announcements we can easily modify the **AODVHello** element by adding a new packet header type that complies with the IARP specification. Arguments to IARPHello now specify the R-hop radius (i.e. the packet's TTL field) as well as the announcement interval. The **IARPHello** element is also used to keep track

of the neighborhood topology as is the case with AODV, since this information is not usually available from the link-layer drivers.

Looking at the DSR input configuration (see Fig. 2.6), we need to add a third classification of incoming packets to handle IARP announcements. The new element sequence following the Ethernet broadcast classification is shown in Figure 3.1. **IARPCacheHello** stores the announced destinations in the routing table, while **IARPUpdateHello** modifies the IARP header before forwarding the packet.

In conclusion, all information is already maintained in the shared routing table, and only two new elements (**IARPHello** and **IARPCache**) were needed to implement this protocol.

3.1.6 Modifying AODV/DSR to comply with IERP

Following the IERP draft guidelines, we need to:

- Disable hello messages (now handled by IARP) by simply removing the AODVHello element from the configuration (if in use)
- Encapsulate **route request** packets with the BRP service; **AODVValidateRREQ** and **DSRValidateRREQ** can be disabled.
- Calculate one additional alternate route for each destination added to the routing table so that a link failure within our zone can be bypassed instantaneously by switching to the second route.

All other features are already supported by AODV/DSR and/or the shared routing table element. When routing a packet, we examine the shared routing cache and select a route to the destination. For the DSR implementation, we had **IARPCacheHello** store source routes in the cache. Both intra-zone and inter-zone packets are encapsulated using the DSR packet format and then explicitly source routed. According to the ZRP specification, the inter-zone routing protocol should incorporate information from

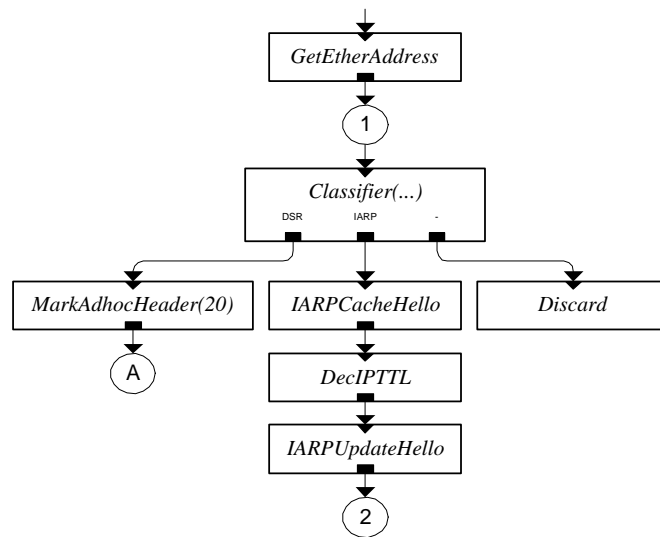


Figure 3.1: IARP Input Elements

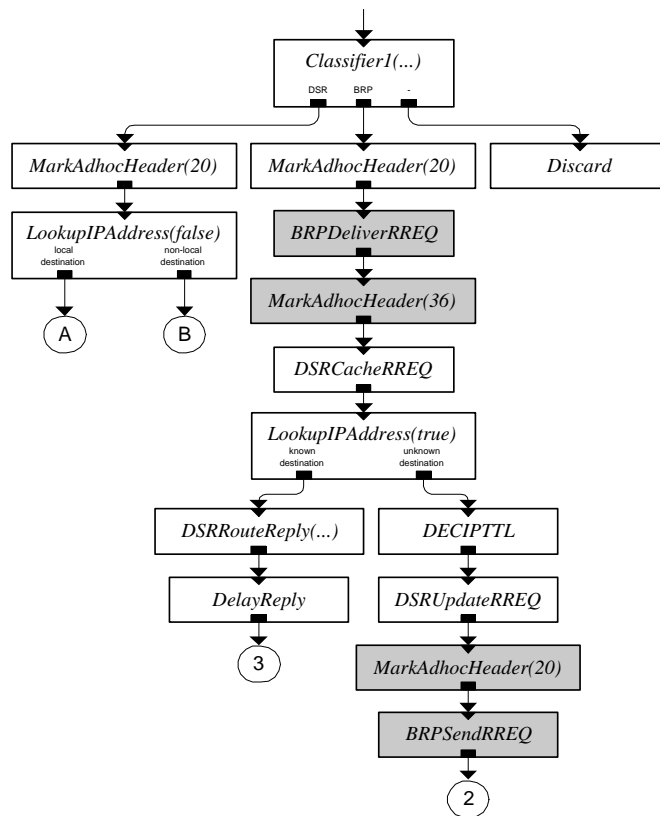


Figure 3.2: BRP Input Elements - Elements Added to DSR Configuration are Shaded

the IARP at each routing step. This is effectively done using our shared routing cache - if no route is known, route requests are broadcast, and nodes use information cached from the IARP route advertisements to assemble routes.

A similar, but different, modification is needed when using AODV for the inter-zone algorithm. Here, the **IARPCacheHello** element simply populates the shared route table with the next hop routing information. Intra-zone packets are forwarded to the appropriate intermediate node. Again, inter-zone packets may require route discovery, but that information will then be added to the same shared route table.

3.1.7 Adding BRP

The idea behind BRP is to encapsulate route requests originating from the IERP layer with BRP headers so that the requests can be directed in a multicast-tree fashion throughout the network. **Route replies** and additional route maintenance functionality, however, is entirely handled by IERP (e.g. DSR). We can encapsulate our DSR requests by adding a **BRPAddHeader** element before route requests are sent to the output device, or by modifying **DSRRouteRequest** to create BRP specific packets right away. My approach uses the **BRPAddHeader** element, which leaves all DSR elements as is, but incurs some additional memory copy overhead. We also need to be able to classify BRP unicast packets by adding a classifier for a BRP protocol number to **Classifier** handling local Ethernet packets (see Fig. 3.2). Packet destination annotations that are found by **LookupIPRoute** have the **ad hoc** header annotation set to the DSR header and are sent directly to the **DSRRouteReply** element. Otherwise, the BRP packet should be forwarded to our peripheral router nodes if we decide they have not yet seen the request (see the BRP draft for more details). **BRPDeliverRREQ** calculates which peripheral nodes have already been covered by this route request, before handing the packet over to DSR. So that global coverage data, along with a cache of already detected queries, is available to all elements in our configuration, the **BRPTable**

information element builds a graph of our network topology, which is again based on the routes found in the routing table. The BRPTable should be configured to handle the same zone radius as the proactive link state flooding protocol (IARP), if also in use, since exceeding that hop count would allow us to make false assumptions about what nodes have already been covered by the route request. The actual packet forwarding logic is contained in the **BRPSendRREQ** element, which delivers the packet to our output configuration. Since it's not clear how peripheral nodes would be assigned multicast addresses, my implementation does not currently support multicast. Instead, **BRPSendRREQ** will create additional copies of the route request to each of the neighbors in our calculated multicast tree. This somewhat unfair overhead may greatly limit the advantages gained by using BRP in the first place.

3.2 Adaptive Hybrid Router

We can experiment with **ad hoc** protocols by combining existing elements to get the desired properties of different routing algorithms. Conceptually, there are several possible approaches:

- Create a new set of proprietary headers with an algorithm that encompasses all the required features of each protocol (e.g. have a router that have all the properties of both DSR and AODV, but use DSR headers since they contain more information).
- Switch router by only allowing a node to operate in one "mode" at a time. E.g. a router that switches to AODV will drop any incoming DSR packets and "shut down" the DSR elements.
- Simultaneous routers - incoming packets are classified according to the protocol used (e.g. AODV or DSR) and handled accordingly, with both routing protocols residing in the kernel at the same time with a shared routing table.

3.2.1 Hybrid Router Configuration

This thesis implements the last solution, where all routing algorithms are combined together and continue to operate as usual (e.g. AODV keeps timing out expired routes as well as sending **hello** messages, even if all incoming traffic is DSR). Each protocol is left unchanged - there is no added functionality. We can do this by combining the AODV and DSR input components to handle both protocols in the initial classification. Outgoing packets are subject to a policy (the **ProtocolSwitch** element - see 3.2.4) that decides which router to use for packets originating from the kernel. Packets that use generic next hop forwarding, without a protocol-specific source route header, will be forwarded to the "best route" next hop, even if a route for that packet was discovered with a different protocol. This applies to DSR and other source route routing algorithms as well only if they use the same packet header format. Also note that all route maintenance messages affect all protocols using the shared routing table; e.g. an AODV **route error** message could invalidate a link originally found with DSR. With AODV, a **route error** telling us that node *A* can no longer reach destination *B* will invalidate our route for destination *B* if our next hop is *A*. We can also invalidate any source routes that have *B* as the next hop from *A*, but we cannot determine from this that any other routes including *A* and *B* (in that order) are no longer valid (since the path used by AODV routing might be different from the source route we have cached and there is no way of knowing this). **AODVInvalidateLink** handles all this, but can also be configured to not bother with source routes, since this presents some additional overhead. For **DSRInvalidateLink**, invalidating next hop links does not result in increased complexity, however.

3.2.2 Shared Routing Table

To allow routes found with one protocol to be used with any other currently running protocol, the routing table is shared between all elements in the configuration. Elements do a lookup into the routing table to acquire and lock route entries before getting an access pointer to that destination. Each destination again has a list of alternate and valid routes, with a pointer to the currently “best” source route and next-hop route. In addition, a lifetime value can be associated with each route.

The routing table data structures should allow for the above properties to be calculated as fast as possible so that we have:

- Speedy access to a given destination’s routing data (e.g. a hashmap indexed by the IP address value)
- Route entries sorted by lifetime, simplifying finding the next route to expire and schedule for timeout (e.g. a heap with all routes based on current lifetime value)
- Route entries sorted by hop length, or some other metric to measure “goodness” (e.g. each destination keeps a sorted list of its routes based on hop length)

Routes are either considered to be complete source routes (e.g. DSR) or next-hop/gateway routes (e.g. AODV) and are indicated as such. This also affects how routes are timed out as the routing table can be configured to have separate timeout or lifetime values for source and next-hop routes. Our second requirement above, then, becomes all the more important; route lifetimes may vary a great deal. I originally did “bulk” timeouts of multiple routes at a constant interval, but experienced improved performance when timing out a route once its lifetime reached exactly zero. Stale cache entries certainly degrade performance, so keeping route entries much beyond their designated lifetime value (assuming that value is proper for the current state of our

network), is obviously damaging. DSR provides an example of this as, by default, it does not expire route entries at all and rely on entries to be invalidated by missing acknowledgments or incoming route errors. The table can however, be configured to handle source routes and next-hop routes differently:

- Source route timeout can be set to any value in milliseconds; default is -1, meaning no timeout will occur. This is applied to any route table entries that have the source route flag set
- Table driven timeout can be set to any value in milliseconds; default is 2000 (specified by the AODV draft). This is applied to any route that only has an entry for the next hop for the given destination

User-level applications can change these values by calling the routing table handler functions. This also raises additional questions like: should packets without a source route (e.g. a packet forwarded using AODV) be allowed to use the gateway/next hop node of a route set up by DSR route discovery even if this route **would** have expired had it not been classified as a source route?

3.2.3 AODV Revisited

An important aspect of AODV is its use of sequence numbers to avoid routing loops and stale cache entries. Now that more than one route to a single destination may be cached in the shared routing table, we have to take care to only increment a table entry's sequence number when a next-hop route is invalidated, since alternate source routes may fail even when the next-hop route is still valid. If the sequence number is unnecessarily incremented, it may give a later AODV **route request** less chance of learning a valid route from intermediate nodes prior to arriving at the requested destination. In fact, keeping a shared routing table complicates the use of sequence numbers. Consider what should happen if a route found with AODV **route discovery**

is timed out and invalidated. Usually the sequence number for that destination is incremented by one so that the following route discovery will return a more recent route. However, what if an alternate route previously found with DSR is still available? In this case I currently keep the sequence number associated with the invalidated route and switch to the alternate route. In addition, the sequence numbers do not reflect the current state of source routes kept in the routing table and should only be used with next-hop routes.

3.2.4 Choosing a Protocol

According to the results presented by [19], DSR does a better job of routing packets as long as loads are kept low with a small number of nodes. I.e. DSR show better performance with low mobility and introduces overall less routing load (though a higher total of actual MAC transmissions), in terms of number of packets at each node. As the size of the network increases, AODV show better scalability. From this, a very simple policy regarding the choice of protocol to use emerges: use DSR until the routing table contains a certain number of entries, as this is an indication of the total number of nodes in network. It does not, however, indicate load in the sense of how much data is forwarded by the router. To add this criteria, a simple packet counter can be inserted into the Click configuration and the protocol selection can take the number of packets forwarded in the last time interval into account when deciding if and when to change routing protocol. A second criteria is to depend on certain properties of a protocol when deciding how to route a packet. If the packet is a multicast packet, the choice of protocol is easy; only AODV currently supports multicast. Should we require unidirectional links, AODV does not support this, while DSR does. **Route requests** are responsible for a large part of the routing load overhead caused by AODV, so we can use the BRP elements to handle this instead.

We can also use mobility as a metric to decide what protocol to use. Simulations

(see Chapter 4) show that AODV provides better throughput during periods of high mobility, while DSR does better when mobility is low. The **ProtocolSwitch** element decides what protocol to choose for packets originating from the kernel by retrieving a current mobility value from the **NetMonitor** element and comparing this with a mobility threshold.

While this policy may adequately predict the most favourable protocol to use at any specific moment, there are other factors to take into account. Most notably, changing from AODV to DSR will leave any routes discovered with AODV unusable for DSR as only the next hop is known and not the complete source route, thus resulting in new route discovery messages for route entries in use (i.e. that are used by the source or destination to initiate new packets, intermediate nodes will still be able to forward DSR packets). So while changing from DSR to AODV does not incur any overhead, changing from AODV to DSR does and it's therefore intuitive to be somewhat 'slower' in changing back to DSR even if the above mentioned policy does dictate a change back. As nodes will start out with no routing table entries and gradually build up the table as route discovery messages are sent out, it makes sense to start out with DSR and immediately change to AODV when the routing table hits a certain number of entries or load becomes large enough to validate a shift. As this will most likely be the most common direction of change of protocols, the penalty associated with going from AODV to DSR will hopefully not apply very often and can more or less be disregarded.

3.2.5 Analyzing Network Activity

To analyze network properties, such as changes to neighborhood topology and level of network mobility given the amount of recent link breaks, the **NetMonitor** information element can be added to a router configuration. The NetMonitor can extract information from both the shared routing table as well as the BRP table, making important information available that is not part of the basic table implementations. A

callback is supplied by the routing table so that the **NetMonitor** can be notified of routes that expire.

One way of deciding mobility is the number of link breaks during a given interval [10]. The routing table increments a **link_breaks** counter when routes time out or as a result of elements invalidating links in the table. It also keeps a **route_breaks** counter, which is the total number of actual routes invalidated (each **link_breaks** increment may be responsible for invalidating several routes). In fact, it would also be interesting to keep track of how many **route_breaks** a single link is responsible for invalidating in order to weight the importance of that link as a measure of the overall network stability, but this has not yet been implemented.

Chapter 4

Evaluation

4.1 Simulation Environment

Our initial simulations give the performance of a 3-node AODV and DSR router on a wired setup. This is then compared with the packet throughput using static Linux routing. Our second simulations use a modified **ns-2** [5] environment that can run any Click configuration script as well as the baseline implementations of DSR and AODV currently available for **ns-2**. This enables us to validate and compare our results with those already published on the performance of **ad hoc** protocols as well as the actual simulation environment. The configuration scripts are easy to read and modify and can be manually set up. To allow for large, simulation-based scenarios, we also provide support for automatically creating multiple configuration files for use with **ns-2**. The simulations have 50 nodes that randomly move about, with a varying number of traffic sources sending 512-byte packets.

4.1.1 Link Layer Feedback

Our AODV implementation use **hello** messages to detect changes in neighborhood topology, while DSR will only detect these changes when failing to transmit a packet. Both [4] and [?] simulate link-layer feedback by notifying the routing layer in **ns-2** when the MAC layer fails to send a packet (i.e. no reply from the next hop node). This is, obviously, very helpful for improving overall efficiency as link failures are discovered

almost immediately (I do not know if there is an implied delay for the sake of the **ns-2** simulation, but for a real implementation, this signal would be sent as soon as the 802.11b hardware stops trying to retransmit the packet without getting back an ACK from the destination). It does not, however, seem to be a feature available with the current set of wireless card drivers, where (at most) only counters reflecting packets sent and received are accessible. This means that the routing layer has to find out when a link goes down on its own, which is much less efficient. In the AODV implementation this translates into waiting for 3 seconds before invalidating a neighbor (**hello** message interval * the number of **hello** messages that are allowed to “disappear”). With a packet rate of 4 packets/s, I could lose as many as 12 packets if the destination moves out of communication range from a source that is transmitting. Our measurements include results from the **ns-2** AODV implementation using both the link-layer detection mechanism as well as **hello** messages, so that the results are comparable with the Click implementation.

4.1.2 Link Layer Collisions

As I started running AODV in the simulator, it quickly became apparent that a lot of packets were getting dropped because of collisions. Unlike a real-world scenario where “random” delays will show up when each node process packets, our ns-2 simulator will run each node at the same exact time with no delays as packets go through the Click router. This caused unnecessary collisions as packets were broadcast simultaneously at each node. To avoid this, I’ve included a **Jitter** element that will delay a packet a random amount of microseconds before forwarding it. This element is also useful in a real-world implementation, although less so. However, as **hello** messages are broadcast at a set interval for AODV, it’s possible that two nodes within communication range could be synchronized so that these hello messages were always lost if we do not insert a jitter. To be able to repeat the exact same simulation, the **Jitter** element can be configured

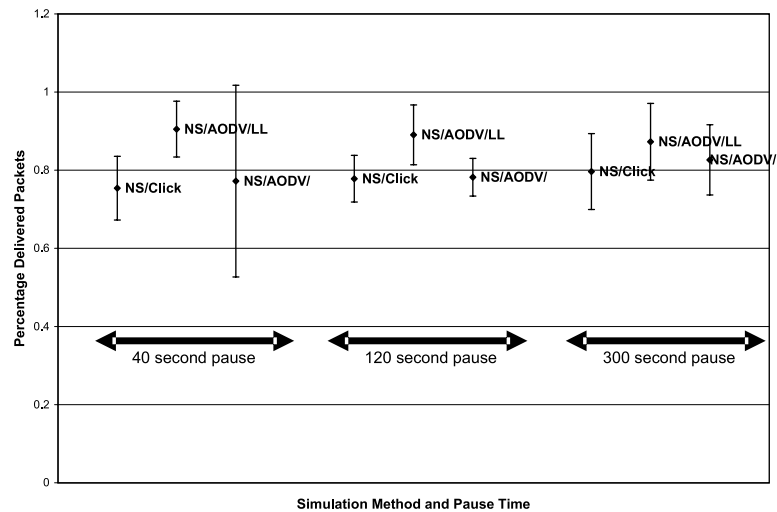


Figure 4.1: Comparison of Click implementation to NS implementation

with a given random seed so that it will give the same random sequence. Also note that if jitter is applied to data packets, it might affect underlying protocols such as TCP since packets may get reordered in the delay queue before leaving the interface.

4.2 Test Bed Simulations

Our preliminary performance testing under Linux shows that our AODV implementation delivers 10 Mbit/s vs. 94Mbit/s for the native Linux routing code using a TCP benchmark. In those tests, AODV was configured as a user-space router rather than a kernel space router, which requires additional buffer copies that slow the routing process.

4.3 ns-2 Simulations

As I don't use a formal method to verify that my AODV implementation is working correctly, I use the simulation results and compare with existing measurements of the **ns-2** AODV protocol to validate my work. Specifically, the throughput measurements indicate the number of packets that get delivered and therefore reflect the performance

of my implementation of AODV. Figure 4.1 show the packet delivery fractions with varying pause time (the time between each node moves; higher pause time gives lower node mobility) compared with the standard implementation distributed with the **ns-2** simulator. The random-waypoint workload is used as the model of mobility. The workloads and network configurations were designed to closely match those used in [19]; in particular, the data in Figure 4.1 is comparable to Figure 1(c) in of the paper by Perkins **et al** comparing the performance of DSR and AODV. We ran three simulation variants. The first, labeled “NS/Click” used our Click implementation of AODV in the **ns-2** simulator. The second, labeled “NS/AODV/LL”, uses the AODV implementation distributed with **ns-2**, which is the same version used in [19]. This version of AODV uses link-layer **hello** packets to detect link breakage. The last variant, “NS/AODV” uses that same implementation, but uses explicit network layer **hello** messages; this version is comparable to our version since we use explicit **hello** messages as well.

To summarize, Fig. 4.1 shows the mean packet delivery rate and the 95% confidence interval for a set of five randomly selected initial workload configurations. It’s hard to compare with the “NS/AODV” implementation, because that specific implementation ended in runtime errors under several scenarios, resulting in much greater confidence level variance. Our implementation, however, show an average throughput within the “NS/AODV” confidence interval and so can be considered to have the same performance at the 95% confidence level. Dropped packets under these conditions result from contention within the wireless media as well as delayed route invalidation when destination and source move out of communication range. Both the Click and NS-2 implementations used the same wireless media model provided by NS-2.

Chapter 5

Conclusions

I have presented a new, modular approach to building complete **ad hoc** routing protocols, such as AODV and DSR. I implemented these protocols both as a proof-of-concept, using the Click modular framework in place of the usual, monolithic kernel implementations, as well as to present a full set of ad hoc elements to provide a toolkit and the building blocks for further research and experimentation. The results presented show that my approach to decompose these protocols into distinct components greatly speeds up development as well as making it simpler to test and experiment with new **ad hoc** protocols. In fact, the Click framework proved to be easily extensible to handle routing algorithms such as DSR and AODV. For instance, I was able to use the DSR implementation as the interzone protocol of ZRP with few modifications. The evaluation results indicate that the Click router configurations work as well as their **ns-2** counterparts and provide the full capabilities of the protocol drafts. Having a shared routing table, however, turned out to be a mixed blessing. At one side, it greatly simplified extending the base protocols when adding IARP and BRP. It probably also made the table overly complex and introduced a few subtle bugs that riddled the early implementation of AODV. The original intent was to have a shared table that would support a hybrid AODV/DSR router, but in hindsight, there is probably too few advantages to sharing routes between the two protocols to warrant the extra amount of bookkeeping needed.

5.0.1 Future Work

To provide a complete routing solution, support for multicast need to be added. AODV already support multicasting, but other multicast schemes are available that are independent of the routing protocol. One such protocol is the The Adaptive Demand-Driven Multicast Routing Protocol for Mobile Ad Hoc Networks (ADMR) [10], which can revert back to simple broadcast flooding, if the conditions for doing multicast degrade below a certain level. This could also be useful for my implementation of BRP, where doing multicast route requests might not always result in better performance than falling back on the request floods sent out by AODV or DSR. It also makes sense to use this in a hybrid router solution that detects changing properties of the network and react accordingly by utilizing the components provided in this toolkit. The AODV/DSR hybrid currently needs more work to take full advantage of the results from the simulation so that the best protocol at any given time can be chosen. I will also look at adding support in AODV and DSR for handling connections between **ad hoc** and infrastructure mode, so that any router node can act as a gateway to the Internet. This will require only minor additions to the existing configurations and should provide some interesting possibilities. Finally, I will complete the BRP implementation so that it complies with the current draft specification and I will also look at implenting additional protocols such as TORA.

Bibliography

- [1] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In Mobile Computing and Networking, pages 85–97, 1998.
- [2] et al. D. Maltz. The effects of on-demand behavior in routing protocols for wireless ad hoc networks. IEEE JSAC, 17(8), August 1999.
- [3] Samir R. Das, Charles E. Perkins, and Elizabeth M. Royer. An implementation study of the aodv routing protocol. In Proceedings of the IEEE Wireless Communications and Networking Conference. ACM/IEEE, 2000.
- [4] Samir Ranjan Das, Charles E. Perkins, and Elizabeth E. Royer. Performance comparison of two on-demand routing protocols for ad hoc networks. In INFOCOMM (1), pages 3–12, 2000.
- [5] K. Fall and K. Varadhan. ns notes and documentation. available from <http://www-mash.cs.berkeley.edu/ns>, Nov 1997.
- [6] Z.J. Haas and M.R. Pearlman. The performamnce of query control schemes for the zone routing protocol. ACM/IEEE Transactions on Networking, August 2001.
- [7] Zygmunt J. Haas, Marc R. Pearlman, and Prince Samar. The bordercast resolution protocol (brp) for ad hoc networks. INTERNET DRAFT 1, IETF MANET group, June 2001.
- [8] Zygmunt J. Haas, Marc R. Pearlman, and Prince Samar. The interzone routing protocol (ierp) for ad hoc networks. INTERNET DRAFT 1, IETF MANET group, June 2001.
- [9] Zygmunt J. Haas, Marc R. Pearlman, and Prince Samar. The intrazone routing protocol (iarp) for ad hoc networks. INTERNET DRAFT 1, IETF MANET group, June 2001.
- [10] J. Jetcheva and D. Johnson. The Adaptive Demand-Driven Multicast Routing Protocol for Mobile Ad Hoc Networks. PhD thesis, Internet-Draft, draft-ietf-manet-admr-00.txt, July 2001. Work in progress.
- [11] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, Mobile Computing, volume 353. Kluwer Academic Publishers, 1996.

- [12] Eddie Kohler. The Click Modular Router. PhD thesis, Massachusetts Institute of Technology, February 2001.
- [13] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. ACM Transactions on Computer Systems, 18(3):263–297, August 2000.
- [14] Fredrik Lillieblad, Oskar Mattsson, Petra Nylund, Dan Ouchterlony, and Anders Roxenhag. Mad-hoc technical documentation. <http://mad-hoc.flyinglinux.net/>.
- [15] D. Maltz, J. Broch, and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed. Technical Report CMU-CS-99-116, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1999.
- [16] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications, pages 234–244, 1994.
- [17] Charles E. Perkins and Elizabeth M. Royer. Adhoc on-demand distance vector routing. In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, pages 90–100, Feb. 1999.
- [18] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. Ad hoc on-demand distance vector (aodv) routing. INTERNET DRAFT 8, IETF MANET group, March 2001.
- [19] Charles E. Perkins, Elizabeth M. Royer, Samir R. Das, and Mahesh K. Markina. Performance comparison of two on-demand routing protocols for ad hoc networks. IEEE Personal Communications, 8:16–28, 2001.

Appendix A

Elements

The following is a complete look at the 56 elements implemented for this thesis. The description specifies what types of ports the elements can connect as, the number of required and optional ports as well as the type of packets expected at those ports. Also listed are the required click annotations in addition to the additional user annotations used by the different **ad hoc** elements.

A.1 Ad Hoc Elements

RoutingTable(ADDR1 [ADDR2 ... ADDR_N] [, SROUTE_LIFETIME, ROUTE_LIFETIME])

Type: Static information element.

This element stores all global routing table data so that it is accessible from all the elements in the configuration. It is usually instantiated at the beginning of a click configuration:

```
// My interface
AddressInfo(me0 192.168.1.115 00:03:47:70:89:14);

// My routing table
RoutingTable(me0);
```

The first argument is a list of any number of previously defined **AddressInfo** en-

tries that specify each of the interface's IP and Ethernet addresses that is to be used by the router for this node. This will usually only be the wireless card. `SROUTE_LIFETIME` is the timeout for complete source routes (defaults to -1 = no timeout), `ROUTE_LIFETIME` is the timeout for other routes. Both values are specified in milliseconds.

The table also provides member functions to:

- Create and add new routes to a given destination
- Invalidate routes
- Queue packets in a send buffer
- Search a destination for the currently shortest valid route and route discovery lifetime
- Expire routes that have not been refreshed or updated within the set lifetime

GetEtherAddress

Type: Agnostic. Input 0: Ethernet. Output 0: Ethernet. Output annotations: Ethernet source address in packet user annotation 1 - 7.

Annotates incoming Ethernet packet with MAC layer source address.

LookupIPAddress([TABLE, SOURCE])

Type: Agnostic. Input 0: IP. Input annotations: Destination. Output 0: IP. Output 1: IP.

`LookupIPAddress` will check to see if the destination annotation matches any of the router's interfaces. If the address is not ours and if `TABLE` is set to true (default), the element will search the routing table to see if a route to the destination is known and valid. If `SOURCE` is false (default), will match any valid route. If set to true, only complete source routes will match the lookup. If the destination annotation matches

one of our interfaces or has a known route, the packet is sent out on output 0. If the lookup fails, output 1 is used.

Output

Type: Push. Input 0: IP. Input annotations: Destination. Output 0: IP for interface 0. Output N (optional): IP for interface N.

Output selects the correct port for unicast packets by sending the packet to the interface stored in the routing table for the destination route. If the destination is a broadcast address, the packet will be copied to every output port.

Every next hop route stored in the routing table is associated with a interface number that is based on the Paint(n) value given to packets as they arrive on a given device. Output allows an arbitrary number of output ports to be connected, but the number of connections should be the same as the number of input devices (or the highest number used to Paint(n) packets from that device). If the output port is not connected, the packet will be dropped.

This element is intended to be used in conjunction with Paint and the RoutingTable. The following is a possible output configuration:

```
// output element
dsr_out :: Output;

// from output
dsr_out[0] -> Print(out_eth0) -> AddEthernet(0) -> Queue -> ToDevice(eth0);
dsr_out[1] -> Print(out_eth1) -> AddEthernet(1) -> Queue -> ToDevice(eth1);
```

GetNextHop([SOURCE])

Type: Agnostic. Input 0: IP. Input annotations: Destination. Output 0: IP. Output 1: IP. Output annotations: Destination.

Expectes IP packets with the destination annotation set - the destination address

is looked up in the routing table. If found, the packet is sent to output 0 with the next hop address set as the new destination annotation. If not, the packet is sent to output 1 (if not connected, the packet is dropped). SOURCE is by default false, meaning that the current shortest route is used. If set to true, the current shortest complete source route (could be same route) will be used instead. Output 1 could be connected to an ICMPError element to generate Unreachable Destination errors.

AddEthernet(INTERFACE)

Type: Agnostic. Input 0: IP. Input annotations: Destination. Output 0: Ethernet.

Element adds an Ethernet header to the packet. Source address is set to that of the output interface we're transmitting on (the INTERFACE argument). Broadcast packets are simply given a broadcast Ethernet destination address, while unicast packets require a lookup into the routing table to get the Ethernet destination address of the next hop destination this packet is bound for.

RefreshRoute([SOURCE ROUTE, SRC, DST])

Type: Agnostic. Input 0: IP. Output 0: IP.

Element refreshes source and/or destination addresses in routing table. SOURCE is by default false, meaning that the current shortest route is refreshed. If set to true, the current shortest complete source route (could be same route) will be refreshed instead. If SRC is true, the IP source address route will be refreshed, and if DST is true, the IP destination address route will be refreshed.

MarkAdhocHeader([OFFSET])

Type: Agnostic. Input 0: IP. Output 0: IP. Output annotations: Ad hoc header.

Sets the ad hoc user annotation. This offset is used by other elements that need to access headers specific to a given ad hoc protocol. These headers usually directly

follow the IP header, but in some cases (e.g. when BRP headers are added), there may be multiple headers.

Jitter([MAX_JITTER, SEED])

Type: Agnostic. Input 0: Packet. Output 0: Packet.

Inserts a randomized jitter delay within MAX_JITTER milliseconds before forwarding the incoming packet. Default MAX_JITTER is 10ms. Should be used to avoid repeated packet collisions when nodes start sending packets in lockstep. Incoming packets are queued in a send buffer so the packet rate should ideally (on average) be less than the MAX_JITTER value so that the queue does not keep growing indefinitely. The SEED value can be specified to seed to random generator so that the delay will always follow the same sequence so that test simulations can be repeated with the same results. If unspecified, the current time of time will be used.

A.2 AODV Elements

AODVCacheRERR

Type: Agnostic. Input 0: IP. Input annotations: Ethernet, Destination, Ad hoc header. Output 0: IP.

Adds the incoming packet's IP and Ethernet annotation addresses to the routing table for the neighboring node we received this packet from.

AODVCacheRREP

Type: Agnostic. Input 0: IP. Input annotations: Ethernet, Destination, Ad hoc header. Output 0: IP.

Caches the incoming packet's IP and Ethernet annotation addresses, adds the forward path and the precursor node of the reverse path of this route reply.

AODVCacheRREQ

Type: Agnostic. Input 0: IP. Input annotations: Ethernet, Destination, Ad hoc header. Output 0: IP.

Caches the incoming packet's IP and Ethernet annotation addresses, adds the reverse path and the precursor node of the forward path of this route request.

AODVHello(SADDR [, INTERVAL])

Type: Push. Input 0: RREP. Output 0: IP. Output annotations: Destination.

Send out route replies to announce our presence to our neighbors every INTERVAL ms so that the protocol does not have to rely on link layer connectivity information. SADDR is the source address of our interface. Default INTERVAL is 2000 ms. Input 0 should receive incoming **hello** messages so that the internal neighbor link table can be updated.

AODVInvalidateLink([SOURCE])

Type: Push. Input 0: AODV RERR. Input annotations: Destination, Ad hoc header. Output 0: AODV RERR. Output 1: AODV RERR.

Looks at the RERR packet and marks any destination(s) in the routing table listed in the RERR message with the next hop matching the IP source as unreachable (by setting the hop count to infinity). Outputs the incoming RERR on output 0 and any new RERRs needed for precursor nodes on output 1 with an updated list of destinations. if SOURCE is true (default is false), the element will also invalidate any source routes that use the IP source and unreachable destination to route packets. This slows things down a little for AODV but will help delete stale cache entries used by DSR in a shared routing table.

AODVOutput

Type: Push. Input 0: IP. Input annotations: Destination. Output 0: IP for interface 0. Output N (optional): IP for interface N.

Same as Output, except route error broadcasts are only sent to output ports for interfaces that have precursor nodes that need to receive this error.

AODVRouteError

Type: Agnostic. Input 0: IP. Input annotations: Destination. Output 0: AODV RERR.

Creates a route error AODV packet for destination set in packet destination annotation, with IP source and destination of that of the incoming IP packet. Drops the incoming IP packet and sends the new route error to output 0.

AODVRouteReply

Type: Push. Input 0: AODV RREQ. Input annotations: Destination, Ad hoc header. Output 0: AODV RREP. Output annotations: Destination, Ad hoc header.

If destination annotation is one of our local interfaces, creates a route reply to the source of the route request with us as the destination. If not, there should be a valid entry in the routing table. The route reply is then created on behalf of that destination. If the **gratious reply** flag is set in the RREQ, an additional reply is sent to the actual destination.

AODVRouteRequest(SADDR)

Type: Push. Input 0: IP. Input 1: AODV RREP. Input 2: AODV RERR. Input annotations: Destination, Ad hoc header. Output 0: IP. Output 1: AODV RREQ. Output 2: AODV RERR. Output 3 (optional): IP. Output annotations: Destination, Ad hoc header.

If a valid route to the requested destination is known, AODVRouteReply will simply output the packet to port 0. If not, the packet is queued and route discovery is initiated. Incoming route replies are processed to see if queued packets can now be forwarded. Route error packets are examined to see if we have queued packets for the

destination recorded in the error and if so, need to initiate new route requests. Route errors can also be forwarded to any nodes that we have listed as precursor nodes to a given destination in our routing table. If a route request is timed out, the packets queued for that destination are sent to output 3. If that port is not connected, the packets are quietly dropped. Argument SADDR should be the local IP address used in the route discovery phase.

AODVSalvage

Type: Push. Input 0: IP. Input annotations: Destination. Output 0: IP. Output 1: IP.

Element will try to locally repair a route by setting the salvage flag in the routing table and initiating the entry's ttl so that AODVRouteRequest can initiate route discovery on behalf of the actual source instead of just returning it a RERR. If the element decides that the packet cannot be saved (happens if the destination entry is deleted from routing table or the last known hop count was larger than the maximum set by the AODV draft), forwards the packet to output port 0. Output 1 should be connected to AODVRouteRequest so that a new route request message can be broadcast if possible.

AODVSetFlags[FLAGS]

Type: Agnostic. Input 0: AODV. Input annotations: Ad hoc header. Output 0: AODV.

Sets the flag AODV header field of incoming AODV packets to FLAGS.

AODVUpdateRREP

Type: Agnostic. Input 0: AODV RREP. Input annotations: Paint, Ad hoc header. Output 0: AODV RREP.

Sets the IP source field to the address of the receiving interface and increments

the RREP header hop count.

AODVUpdateRREQ

Type: Agnostic. Input 0: AODV RREQ. Input annotations: Paint, Destination, Ad hoc header. Output 0: AODV RREQ.

Sets the IP source field to that of the interface the packet was received on and updates the RREQ destination sequence number so that it is the maximum of the RREQ destination sequence number and that of the last known route sequence number. It also increments the RREQ header hop count.

AODValidateRREP

Type: Push. Input 0: AODV RREP. Input annotations: Ad hoc header. Output 0: AODV RREP. Output 1 (optional): AODV RREP.

Checks to see if this route reply contains a greater destination sequence number or a smaller hop count than any replies previously forwarded to the source. If not, the route reply is output on port 1 or dropped - this decreases the amount of replies a source node will receive, while letting it receive the most up to date and shortest path to destination.

AODVValidateRREQ

Type: Push. Input 0: AODV RREQ. Input annotations: Ad hoc header. Output 0: AODV RREQ. Output 1: AODV RREQ. Output 2 (optional): AODV RREQ. Output annotations: Destination.

Checks to see if the broadcast id of the route request packet has recently been processed by this router. If yes, the packet is dropped or sent to output 2. If not, the element looks to see if the routing table has a valid route to the requested destination. If yes, or if we're the actual destination, the packet is sent to output 0. Otherwise the route request should be rebroadcast from output 1.

A.3 DSR Elements

DSRAckReply([TAILROOM])

Type: Push. Input 0: DSR. Input annotations: Paint, Ad hoc offset. Output 0: DSR. Output annotations: Destination.

Creates an ACK based on the incoming request, and outputs the new packet with DSR headers on output 0. Original packet is quietly killed off. The Ad hoc offset should have been set to point to the offset of the AREQ header in the incoming packet. TAILROOM can be set to allocate additional space following the ACK in the new packet.

DSRAckRequest

Type: Push. Input 0: DSR. Input annotations: Paint, Ad hoc offset. Output 0: DSR.

Adds an **acknowledgment request** to the incoming packets and times out route entries that have previously issued a **acknowledgement request** but are still waiting for an acknowledgment back. Expects the incoming packet to have enough allocated space for this new header, starting at the Ad hoc offset.

DSRAddHeader([SOURCE, TAILROOM])

Type: Agnostic. Input 0: IP. Input annotations: Destination. Output 0: DSR. Output annotations: Ad hoc header, Ad hoc offset.

Adds the static DSR header after the incoming packet's IP header, while increasing packet length to accommodate the new header as well as the source route headers if SOURCE is set to true and an additional TAILROOM bytes. The IP header protocol field is set to DSR.

DSRCacheAck

Type: Agnostic. Input 0: DSR. Input annotations: Ad hoc offset. Output 0:

DSR.

Refreshes the source of the ACK in the routing table so the entry's lifetime is extended and the **acknowledgement request** will not time out.

DSRCacheRREP

Type: Agnostic. Input 0: DSR. Input annotations: Ethernet, Ad hoc offset.

Output 0: DSR.

Processes the incoming DSR packet and records the routes found from the source to our node by looking at the source route header. The Ethernet address of the node listed as upstream from us is also recorded in the routing table.

DSRCacheRREQ

Type: Agnostic. Input 0: DSR. Input annotations: Ethernet, Ad hoc offset.

Output 0: DSR.

Caches routes found in the source route header as well as the Ethernet address of the last recorded hop.

DSRCacheRREP

Type: Agnostic. Input 0: DSR. Input annotations: Ethernet, Ad hoc offset.

Output 0: DSR.

Caches routes found in the source route header. Any routes from the source node to our node and (possibly) from us to the destination node are added.

DSRGetNextHeader

Type: Push. Input 0: DSR. Input 1: DSR. Input annotations: Ad hoc header, Ad hoc offset. Output 0: DSR. Output annotations: Ad hoc offset.

DSRGetNextHeader sets user annotation ADHOC_OFFSET_ANNO to point to the next header option in the incoming DSR packet and outputs the packet on the following ports, based on the current DSR option:

- (1) IP; no more DSR options
- (2) Source route option
- (3) Acknowledgement request option
- (4) Acknowledgement option
- (5) Route error option
- (6) Route reply option
- (7) Route request option

Packets coming in on input 0 are not expected to have the ADHOC_OFFSET_ANNOUNCE annotation set, while packets on input 1 is expected to have the annotation set to point to the next DSR header option.

DSRGetNextHop

Type: Push. Input 0: DSR. Input annotations: Ad hoc offset. Output 0: DSR. Output 1 (optional): DSR. Output annotations: Destination.

Searches packet for the DSR source route header and sets the current hop field to the next destination in route and records that address in the packet destination annotation. If no next hop is found (or next hop route entry is no longer valid), packet is output to port 1, if connected (otherwise, the packet is dropped). If the destination is still valid, the packet is sent to output 0.

DSRInvalidateLink

Type: Agnostic. Input 0: DSR. Input annotations: Ad hoc offset. Output 0: DSR.

Removes cached routes containing the broken link listed in the route error header.

DSRRouteError([SOURCE, TAILROOM])

Type: Agnostic. Input 0: IP. Input annotations: Destination. Output 0: DSR.
Output annotations: Ad hoc header, Ad hoc offset.

Creates a **route error** for the address set in the packet destination annotation, with IP source and destination of that of the incoming IP packet. Packet length is set to accommodate the **route error** as well as an additional TAILROOM bytes plus space needed for the source route header (if SOURCE is true). The incoming packet is dropped.

DSRRouteReply([SOURCE, TAILROOM])

Type: Push. Input 0: IP. Input 1: DSR. Input annotations: Paint, Ad hoc offset.
Output 0: DSR. Output annotations: Destination, Ad hoc header, Ad hoc offset.

Similar to AODVRouteReply.

DSRRouteRequest(SADDR)

Type: Push. Input 0: IP. Input 1: DSR RREP. Input 2: DSR RERR. Input annotations: Destination, Ad hoc header. Output 0: IP. Output 1: AODV RREQ. Output 2 (optional): IP. Output annotations: Destination, ad hoc header, ad hoc offset.

Similar to AODVRouteRequest. Additional route errors, however, are not created, so there are only 3 possible outputs ports.

DSRSalvage

Type: Push. Input 0: DSR. Input annotations: Destination. Output 0: DSR. Output 1: DSR. Output annotations: Ad hoc offset.

This element tries to salvage the packet if a valid route to destination is cached and the DSR header salvage count is below DSR_SALVAGE_LIMIT. Packet size is increased or decreased if necessary and annotated with the offset to the new source route so that that DSRSourceRoute can add the new address list. If packet cannot be salvaged, it's sent to output 1 (if connected). If it can, the packet is sent to output 0.

DSRShortenRoute

Type: Push. Input 0: DSR. Input annotations: Ad hoc offset. Output 0: DSR.
Output annotations: Destination.

Will look through the DSR source route in the incoming (overheard) packet to see if the local host is a listed hop and shorten this packet's source route if possible. If successful, the packet is sent to output 1 (usually connected with DSRRouteReply), otherwise the incoming packet is sent to output 0.

DSRSourceRoute

Type: Agnostic. Input 0: DSR. Input annotations: Destination, Ad hoc offset.
Output 0: DSR. Output annotations: Ad hoc offset.

Encapsulates incoming packet with an additional DSR source route header with a complete route to the destination set in the packet's destination annotation. The element expects the packet to have enough room to accommodate the full header.

DSRStripHeaders

Type: Agnostic. Input 0: DSR. Input annotations: Destination, Ad hoc header.
Output 0: IP.

Removes all DSR headers from the front of the packet (which have been added other nodes to do source routing by DSRAddHops and possibly DSRRouteRequest or DSRRouteReply). The Ad hoc header annotation is expected to point to the static DSR header. The original IP packet is returned on output 0.

DSRUpdateAREQ

Type: Agnostic. Input 0: DSR. Input annotations: Ad hoc offset. Output 0: DSR.

Updates source address of **acknowledgement request** to the address of the interface the incoming packet was received on and forwards it on output 0.

DSRUpdateRREQ

Type: Agnostic. Input 0: DSR. Input annotations: Paint, Ad hoc offset. Output 0: DSR. Output annotations: Destination.

Adds the address of the interface the request was received on to the list of hops, updates the **route request** header and forwards the packet on output 0.

DSRValidateRREQ

Type: Push. Input 0: DSR RREQ. Input annotations: Ad hoc offset. Output 0: DSR RREQ. Output 1 (optional): DSR RREQ. Output annotations: Destination.

Checks to see if the broadcast id of the route request packet has recently been processed by this router or if we're the source of the request. If so, the packet is sent to output 1. If this is a valid **route request** that needs further processing, the packet is sent to output 0.

A.4 IARP Elements**IARPCacheHello**

Type: Agnostic. Input 0: IARP. Input annotations: Paint, Ethernet, Ad hoc header. Output 0: IARP.

Adds to the routing table the source of the **hello** message as well as its listed neighbors.

IARPHello(SADDR [, INTERVAL])

Type: Push. Output 0: IP. Output annotations: Destination.

Similar to AODVHello, but replaces the **route replies** with a list of all neighbors. The packet header format is specified by the IARP draft [9].

IARPUpdateHello

Type: Agnostic. Input 0: IARP. Input annotations: Paint, Ad hoc header.

Output 0: IARP.

Changes the IP source address to that of the interface address we received this message on. The element then decrements the IARP TTL field by one before forwarding the announcement.

A.5 BRP Elements

BRPAddHeader

Type: Agnostic. Input 0: IP. Input annotations: Paint, Destination. Output 0: BRP. Output annotations: Ad hoc header.

Adds the static BRP header after the incoming packet's IP header.

BRPDeliverRREQ

Type: Agnostic. Input 0: BRP. Input annotations: Paint, Ad hoc header. Output 0: BRP.

Prepares the incoming RREQ for delivery to a higher layer protocol such as AODV or DSR. To effectively guide the broadcast outwards, the element marks the nodes covered by this **route request** so that it won't flood the network.

BRPSendRREQ

Type: Push. Input 0: BRP. Input annotations: Paint, Ad hoc header. Output 0: BRP. Output annotations: Destination.

Relays or borderdcasts the RREQ towards our peripheral nodes.

BRPTable([RADIUS])

Type: Static information element.

Stores all global BRP table (i.e. coverage) data so that it's accessible from all elements in the configuration. RADIUS is the zone radius, which defaults to 2. If proactive IARP route flooding is used as well, RADIUS should be set to at least the

same hop count as the TTL for **hello** messages (would preferably be equal).