Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: An Implementation of the Zone Routing Protocol in ns-2 (Network Simulator 2)

Author: Prashant Gopal Inamti

Abstract: The ns-2 simulation software provides useful modeling tools to simulate common network protocols. Of special interest are simulating Reconfigurable Wireless Networks, which consist of mobile nodes that are highly-adaptive to rapid changes in network topology.

The deployment of RWNs requires a distributed routing scheme because each node will need to a degree knowledge of network topology to self-adapt to changes in connectivity. The major challenge is in designing an efficient means of distributing this topological information.

Both pro-active and reactive schemes exist for solving the dsitributed routing problem. The Zone Routing Protocol (ZRP) presents a hybrid solution and incorporates aspects of both pro-active and reactive routing protocols. ZRP introduces the concept of a zone, whose size is defined by radius R hops, to which the pro-active IARP solution applies. Requests for routes external to a node's zone cause a reactive route discovery process, to which the IERP (IntErzone Routing Protocol) part of ZRP applies. In addition, the third sub-protocol BRP (Bordercasting Routing Protcol) enhances ZRP's efficiency through query control mechanisms for the external route discovery process.

This implementation of ZRP on ns-2 operates as per the specificiation found in the Internet Engineering Task Force (IETF) Drafts for IERP, IARP and BRP. The latency observed in some initial experiments (on the order of 1-3 milliseconds per hop) was found to be isolated to the lower layer models of ns-2.

The ZRP module will be submitted as an open-source contribution to the ns-2 community to further expand the simulation toolset for investigating RWN technology.

Report Approved by

Project Advisor:____ Date:_______________

AN IMPLEMENTATION OF THE ZONE ROUTING PROTOCOL IN NS-2 (NETWORK SIMULATOR 2)

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

by

Prashant Gopal Inamti

Project Advisor: Zygmunt J. Haas

Degree Date: January 2003

I. Introduction

The goal of this design project was to provide an implementation of the Zone Routing Protocol (ZRP) as a module to ns-2 (Network Simulator 2), a freely available network simulation software suite.

Ns-2 and Wireless Simulations

Ns-2 provides a framework for simulation of wired and wireless networks, including some facility for emulation. The ns-2 simulator is written in C++ with a Tcl shell front-end that uses oTcl (object-oriented Tcl) libraries Scenarios are run by feeding an oTcl script to the ns-2 executable. The output can be read directly or post-processed by an interactive graphics viewer called NAM. NAM does not allow changing parameters on the fly, it is for post-viewing of a simulation dump (a .nam file).

As of this writing the graphics viewer NAM is not advertised to work with wireless simulations, but there are apparently work-arounds[1]. Generally ns-2 has a different architecture for wireless and wired node simulation. Though this report will only examine purely wireless simulations, the code can be adapted for mixed wired/wireless environments.

A community of users and designers has grown around the ns-2 software. The website has pointers to the latest code and documentation, and additional modules submitted by the developer community. There are several mailing lists for users and developers, and these are conveniently archived on the site.

The design section will delineate the specifications for the ZRP protocols, and describe how design decisions were made. The implementation section walks through the functionality of the code. The conclusion should tell what this module looks like, what it can be used for, and the scope of its utility.

Zone Routing Protocol

ZRP is a distributed routing protocol for Ad-Hoc networks that combines and balances the goals of both pro-active and reactive route discovery schemes, and also specifies a query-control mechanism.

The ZRP as implemented by this project refers to the specification given in the IETF Internet Drafts[2]. The Draft specifies three protocols: IARP, BRP and IERP.

The IARP (IntrAzone Routing Protocol) pro-actively and periodically distributes route information among the members of a zone, defined as the nodes that are up to and including radius R hops away. This parameter R singularly defines the behavior of the overall ZRP protocol.

In this project the IARP relies on information from the NDP (Neighbor Discovery Protocol) to track neighbors and link states with neighbors. NDP uses a beacon and acknowledgment packets to obtain link states. New neighbors or expired entries in the neighbor table trigger an immediate update to the entire zone, and periodic link updates keep the entire zone appraised of each other's states.

The prime directive of IARP is to provide a readily available route when a packet needs to be routed to a destination within the zone. But when a packet arrives for which no local route exists, the IERP is in charge of finding the so-called outer route, or external route.

The IERP (IntErzone Routing Protocol) sends out requests that are forwarded, accumulating the route along the way, until a forwarding node, or relay, detects that the destination is within its zone. Then a reply is created with the full route and sent back to the source.

---

[1] Check the ns-2 mailing list archives.

[2] By Haas, Pearlman and Samar. Another helpful reference is chapter 7 from the Ad Hoc Networking text by Charles Perkins.

The flood of packets from this route discovery process can overwhelm a network, which is where the BRP (The Bordercasting Routing Protocol) steps in. BRP sends, or bordercasts, IERP requests only to peripheral nodes, also called bordercasters, which in turn forward to their bordercasters.

BRP further steers outgoing queries by using QD (Query Detection) and Early Termination. Queries destined for a previously queried bordercaster, or a previously queried routing zone, are discarded. Queries are detected as they traverse a network, and this information is used to (ET) terminate redundant trajectories.

II. Design

The code was written according to the IETF draft specification. Some details were specified by the project advisor and gleaned from other sources, such as the Ad Hoc Networks text. The specification is given in detail in the IETF Drafts.

System Requirements

This project was developed on a Linux machine running Mandrake 8.1 with kernel version 2.4.8-26mdk. The ns-2 version was "ns-allinone-2.1b9", which is a single tarball with all the requisite packages that easily installs with one command. A link to the ZRP2002 package is available on the contributed module page of the ns-2. See Appendix I for details on how to obtain, install and use the ZRP code.

The ZRP code should work with the current version of ns-2 on most platforms. Running under future versions of ns-2 might require some modification of the code.

Design of the ZRP Module for ns-2

The ZRP module was designed within the context of the mobile node architecture of the Monarch (Mobile Network Architecture) Project, a joint project between Rice University and CMU. Their software is considered an extension of the ns-2 code-base.

The mobile node differs from wired nodes in that the connectivity and other aspects of network communication depend greatly on the geographical location of the nodes. The model accounts for key aspects of mobile communication, including position and trajectory, propagation and attenuation, media access, power, and a wireless network interface.

The ns-2 mobile node is fundamentally a different kind of object than a wired node. Ns-2 also provides a hybrid node for simulations requiring interfacing between wired and wireless networks. This project does not cover these kinds of simulations, but the module does not necessarily preclude them either. The modular design of the project should allow easy adaptation for hybrid nodes.

The choice of node architecture within between a DSR-like architecture or generic node was straight-forward, as the generic node was less complicated to use. The DSR node on the surface seemed useful, because it forces all packets received by the node to go to the routing agent. The ZRP too would be processing all packets, including data packets, at every hop. The workaround was to encapsulate the upper layer header inside the ZRP header in data packets. The ZRP agent at the destination node could then unwrap the original packet and submit it to the upper layer.

Another implementation of ZRP as an upper layer agent was possible. But this configuration was ruled out as it poses problems in addressing, and is not as elegant as keeping the routing agent where it was designed to be. The mobile node is shown in Diagram 1.

Ns-2 packets mostly contain the headers of all the protocols ns-2 simulates. To save space for massively large simulations, the header space can pruned to include only those needed by the simulation.

Route accumulation which is part of BRP and link updates required using the data portion of the packet. Using the built-in data packet routines was problematic, as there appeared to be a size limitation on how much memory could be

allocated, and were finally abandoned. The workaround was to use pointers embedded in the ZRP header pointing to where the dynamically allocated data was stored.

A test was required at every step of the implementation because so many aspects of the ZRP depend on other parts of the protocol. The ns-2 trace file did not provide the detailed feedback required therefore the testing was accomplished using formatted printfs that went to standard output.

The current state of each node was printed for every event that occurred for each protocol, including the neighbor table, local zone route table, link state table, and the list of peripheral nodes. Each event is described in detail. An event is

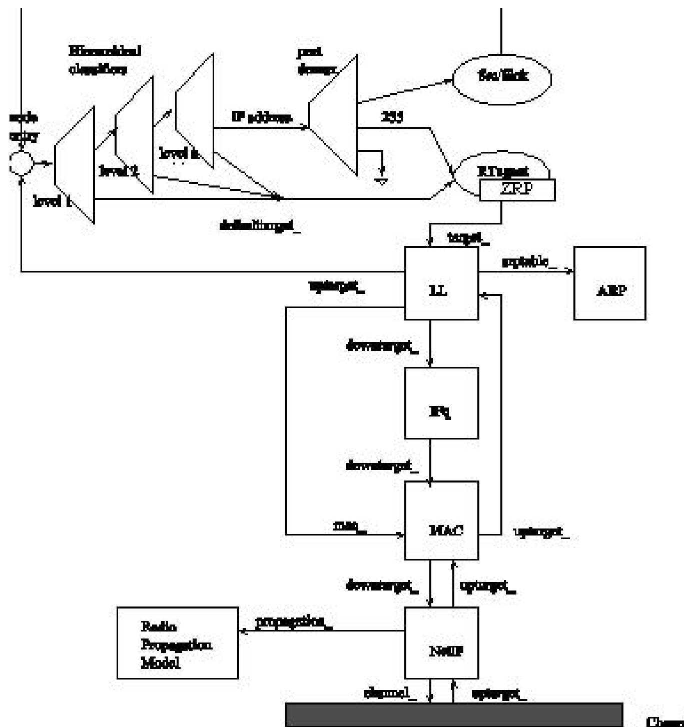

Diagram 1. The ZRP routing agent within the mobile node architecture.

printed with the node id first, then the time, and then aspects of the event delimited by the "|" character. The node tables are printed at the end and are delimited by "!". Here is an example:

```
_ 9_ [12.519476] | Node 9 sent a beacon (seq no. 4). | Node 9 started an
ack-timer to expire at 14.519476 sec. | radius is 3 ! Neighbor Table: 8 !
LinkTable: 17=21 8=9 5=6 18=17 4=5 18=19 7=6 7=21 7=8 4=3 16=17 16=15 !
Routes [8 9 ] [7 8 9 ] [21 7 8 9 ] [6 7 8 9 ] Periph[6 21 ]
```

The design does NOT perform: route repair, route shortening, or the second type of query detection (QD2, i.e., hidden terminal). Accumulated routes are not distributed, but are fully included in each IERP Request and Reply. Link metrics can easily be added to the link state structure as required. The project assumed 802.11 as the MAC protocol.

The dual nature of ns-2 nodes as part-Tcl object and part-C++ object provided many development challenges, but the scripting language Tcl greatly facilitates simulation runs once the C++ aspect of the code is done. The ns-2 code

includes a way of binding variables between the two worlds, via the command() function. For ZRP the most important parameter is the zone radius, but other parameters have been included, including beacon period and the ability to "suspend" nodes temporarily[3]. See Appendix I on how to add new parameters.

III. Implementation and Results

The functionality of the four different protocols NDP, IARP, IERP and BRP is included in the ZRPAgent class and support classes for link state tables and route tables. Key class methods will be treated in detail and mapped to ZRP protocol functions. Refer to Diagram 2.

When a ZRPAgent is instantiated by Tcl, the constructor ZRPAgent(nsaddr_t id) initializes most of the variables. A few variables are initialized when the Tcl code calls the startup() method, and primarily the NeighborScan, BeaconTransmit and PeriodicUpdate timers are started. These are global timers that trigger the scanning of neighbor tables for expired entries, beacon transmission and periodic link updates to the extended zone respectively.

Actions in ns-2 are triggered by either packet or timer events. Packets are events in ns-2, as they inherit from the Event class. The trigger handler for packets is the recv() method and for timers is the handler() method. Packet handling is performed by the ZRPAgent itself by the recv() method. The timers are different classes altogether from the ZRPAgent, and their handlers need to refer to the agent through the agent_ pointer when calling ZRPAgent methods.

The packet manipulation routines initialize, copy or modify, and send packets. The packets can be unicast to the next hop or broadcast to all of them. Routines for adding and reading a link state data structure to and from a packet are pkt_add_link() and pkt_read_update() respectively. Other routines are used to load and extract a route list to and from the packet, add_local_route() for adding a local route and add_outer_route for adding a route to a node external to current zone. The send_next_hop() and send_prev_hop() find the appropriate hop to send to given that packet is at a node indicated by a marker on packet header. The IERP requests go in the outward direction and use the send_next_hop() method and IERP replies go back towards the source and use send_prev_hop().

---

[3] The Suspend command is only included as an illustration, ns-2 provides a better way, see ns-2 documentation.

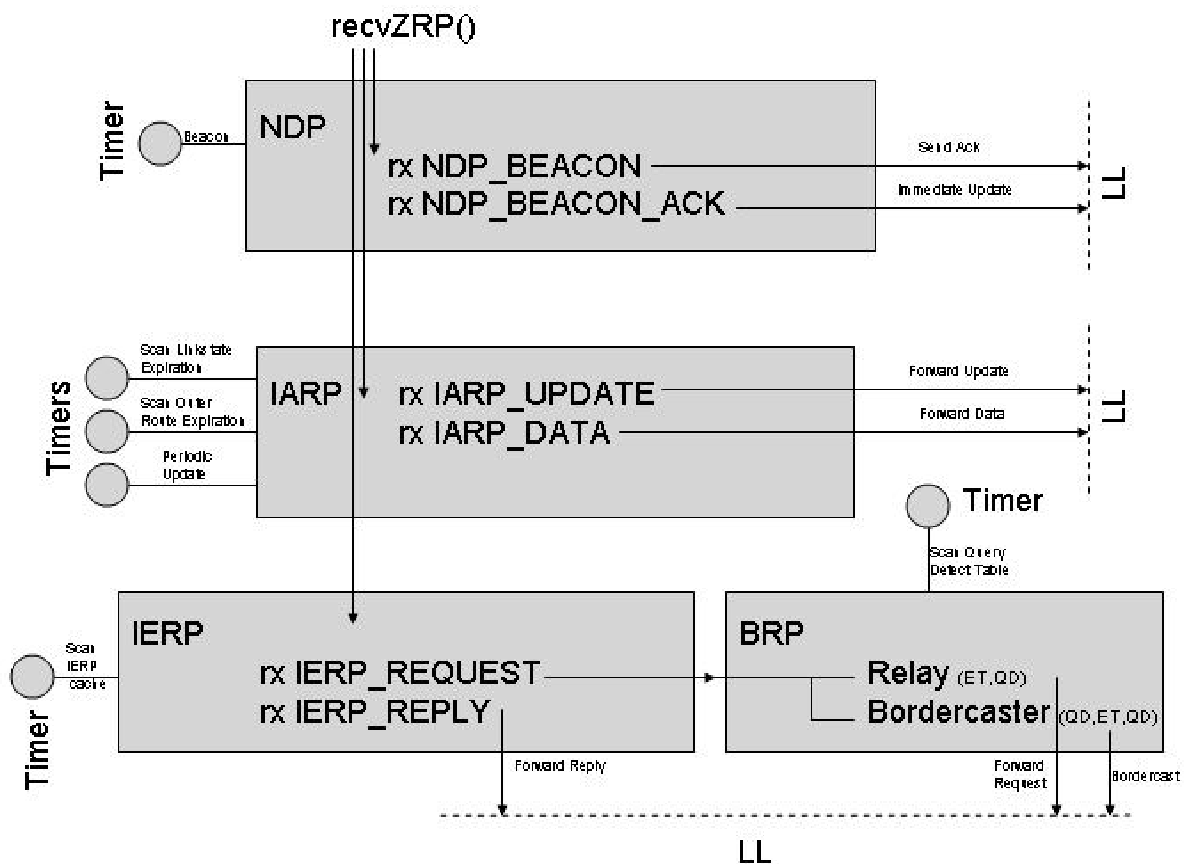

Diagram 2. Packet Flow Diagram.

The heart of the ZRPAgent is the recv() and recvZRP() methods. The recv() function handles packets received from the upper and lower layers. If the packet is from the upper layer, it calls route_pkt(), which is considered IERP, to find out if it can route it locally through IARP by calling add_local_route(). If there is no local route, route_pkt() checks if there is a cached outer route by calling add_outer_route(). If so, the route is appended and sent. Otherwise, as IERP route_pkt() originates a request by calling the originate_request() method, which unicasts a request to each peripheral node.

The recv() method may find that the packet instead comes from the lower layers, ie it has been forwarded, and if the packet has a ZRP-type header, submits the packet to recvZRP(). This latter method is a large switch statement with cases for each type of ZRP packet: NDP_BEACON, NDP_BEACON_ACK, IARP_UPDATE, IARP_DATA, IERP_REQUEST and IERP_REPLY. The prefix of each type indicates the protocol to which it belongs. Each case will either cache the packet for the future, drop it, or re-send it.

For NDP, recvZRP() processes beacons by sending an acknowledgement, re-using the packet object just received, setting TTL to 1. Upon receiving an NDP_BEACON_ACK, NDP through recvZRP() checks if the neighbor is in its neighbor table. If it is not, NDP adds it and also updates the link state table and route table. A new neighbor also triggers an immediate IARP update to be sent.

All IARP_UPDATEs are processed the same way. First the link state table is updated. If IARP as recv() finds there was a change, it re-computes the route table. IARP forwards the update by broadcast if the TTL is not zero.

IERP_REQUEST packets will encounter nodes that will identify themselves as either relays or as a bordercaster. First, the recvZRP() as BRP looks to see if the destination is in its zone. If so, an IERP_REPLY is created, the full route

appended, and the reply is sent to the previous hop via send_prev_hop(). If the destination is not in the current zone, the node decides whether it is a relay or bordercaster by inspecting the route list. The last route entry of a request by definition will name the next bordercaster, so the current node can decide if it is a relay or bordercaster by testing if its id matches with the last entry.

If the node is a relay, the packet is sent by unicast to the next bordercaster if it is not on the covered list. Not sending a packet if it is on the covered list is part of Early Termination. The covered list is found by looking up the query by the unique (query id, origin) pair in the queries_ table. The relay adds the nodes internal to the last bordercaster's zone, not including its peripheral nodes, by calling query_detect1(). Finally, the next bordercaster is added to covered list.

If the node is a bordercaster, the node first marks alll the nodes in the last bordercaster's zone as covered. The node then unicasts a copy of the request to each of its peripherals if that peripheral is not on its covered list, Early Terminating those that are. After the bordercast is done, the node marks the nodes of its own zone as covered.

When recv() gets an IERP_REPLY, it is as IERP, which then relays the packet to the previous hop by send_prev_hop() or hands the packet off to IARP via ierp_notify_iarp() if the node is the originator of the original request. The outer route is cached with an expiry time in a cache to be re-used for outer-routing packets sent in the near future.

The IARP_DATA packet encapsulates the upper layer packet. The IARP_DATA packet is forwarded hop-by-hop, IERP-to-IERP, to the destination ZRPAgent, where it is unwrapped by recvZRP() (ie the original port and packet type are placed in the header) and sent upward to its upper layer.

The timer events trigger actions that will lead to a packet transmission (beacon transmission and IARP updates) or the scanning and purging of expired data entries in tables (scan IERP cache, scan query detect table and scan neighbor table). All these timers re-schedule themselves at the end of the handle() method, with the exception of the acktimer, which is only scheduled by NDP when a beacon is broadcast.

Other classes supporting ZRP are the LinkStateTable and the RoutingTable classes. The most important method for the LinkState class is the update() method. The input is a link_struct which includes source, destination, and the linkstate. If this link is in the table, the state is copied to the linkstate entry stored in the table, the link expiry is updated and the sequence number of the incoming link state data is also copied to the stored linkstate entry. If the link is not found in the table, the link is added and stamped with an expiry time.

The RoutingTable class method compute_routes() uses Prim's algorithm to create a minimum spanning tree starting from the current node. The data structure is an array of linked lists. Each array element is an entry for a destination, and contains the destination's id in the data structure. The next hop from that node can be found by looking at its "next_" pointer, which just points to another array element. All entries, if iterated through as a linked list, end up finally at the first entry, which is the current node's route entry for itself.

Results

Simulations were run by calling a Tcl script from the ns prompt. Each script therefore defines one simulation scenario that defines the location of nodes, protocols used and commands submitted to each individual node.

Simulations show that the route finding ability of the ZRP is accurate and timely, the routes are found with only 2-3 millisecond delay per hop and flooding is minimized thanks to the query-quenching mechanisms of BRP. The data streams (IARP_DATA type) had delays of 1.35ms. The increase in traffic during route discovery is definitely a factor in adding delay in a shared media access environment.

An investigation of the delays in the data stream show it is isolated to the lower layers below the network layer. A special test node was created in C++ devoid of all other functionality except that to broadcast one kind of packet, and receive one kind of packet. Two nodes were created and one was forced to be the transmitter and the other the receiver. The nodes were placed 250 meters apart giving a propagation delay below a microsecond, which is not enough to account for the entire delay observed.

Table 1 was generated by running a scenario where the test nodes were set

to have varying dataRates and varying packet sizes.

	dataRate (Mbps)		
pkt size (bytes)	1 Mbps	5 Mbps	10 Mbps
0	0.001194	0.001015	0.000993
100	0.001994	0.001175	0.001073
200	0.002794	0.001335	0.001153

Table 1. One-hop delay, network-layer to network-layer, in milliseconds for a given packet size (in bytes) and a given MAC dataRate_ (in Mbps).

The delays were calculated by subtracting the time when the routing agent (in this case the dummy network layer that pumps packets) hands the packet to the lower layer from the time the routing agent in the receiving node receives the packet from the lower layer. This delay is accounted for by all the intermediate Mac protocol handshaking/backoff delays and the MAC data transmission delay added together.

The transmission delay for the first column is 100 bytes * 8 bits/byte / 1 Mbps = 800 µsec for every 100 bytes sent. The delta of the values between every row in the first column can be confirmed by inspection to be exactly 800 µsec. Similarly, the theoretical delay for the second and third columns are 160 µsec and 80 µsec, matching the deltas for each column respectively. Therefore by varying the data size one can tease out the delay factor due only to data transmission and see that it relates to data rate inversely, where delay = pkt_length / data_rate.

We can also inspect the trace output file of ns-2 for one of the scenarios (settings are basicRate=1Mbps, dataRate=1Mbps, pkt size = 20 bytes) to look for latency characteristics:

```
s 13.248346074 _1_ MAC  --- 0 RTS 44 [4be 2 1 0]
s 13.253531431 _1_ MAC  --- 0 RTS 44 [4be 2 1 0]
r 13.254199098 _1_ MAC  --- 0 CTS 38 [384 1 0 0]
s 13.254209098 _1_ MAC  --- 101 undefined
r 13.255100764 _1_ MAC  --- 0 ACK 38 [0 1 0 0]
s 16.942309585 _0_ RTR  --- 121 undefined 20
s 16.942384585 _0_ MAC  --- 121 undefined 72 << Mac data
r 16.942961419 _1_ MAC  --- 121 undefined 20 << Mac data delay = ~ 600 µsec
r 16.942986419 _1_ RTR  --- 121 undefined 20
```

With the given settings, we expect a pure MAC data transmission delay (not including back-offs, etc.) of about $20*8/10^6$ sec or 160 µsec, but the trace above indicates a 600 µsec delay. The extra delay could be accounted for in one of two ways, either by extra bytes embedded in the data stream inserted by a lower layer function or by some other aspect of the model in the lower layer (queue model for example), the latter being more likely. As the MAC and LL layers are outside the scope of this project, we shall limit our analysis but this exercise illustrates how to analyze a given characteristic using the models that ns-2 provides.

Conclusions

The simulation shows that ZRP module works for the scenarios presented by the Tcl scripts, but the implementation may not scale well in practice (theoretically, yes) unless the printf output is abbreviated. A large scale scenario would otherwise inundate the user with an unreasonable amount of data. Curtailing information overload can be done simply by wrapping each printf with a conditional that depends on a parameter that can be modified from Tcl. The Tcl code might look like:

```
set r_(0) [$node_(0) set ragent_]
$ns_ at 0.0 "$r_($i) ndp_output off"
```

See the Appendix I for further information on adding Tcl bindings.

The behavior of the nodes in the test simulations were in line with the specifications for the sub-protocols of ZRP, as corroborated by the careful inspection of the detailed ZRP event log output. Having the NAM graphical view of

network simulations would be a welcome addition to the toolset if it can be fixed to work with wireless simulations in the future. The right parsing language at one's disposal (perl and python for instance), however, and the native trace ns-2 output and the ZRP event log output present sufficiently powerful tools by themselves for analyzing protocol behavior.

For the scenarios run so far, route discovery takes 2-3 ms per hop and data streams take nominally 1.35 ms hop for a specified data rate of 1 Mbps, a basic rate of 1 Mbps and packets with a length of 20 bytes. These delay values will vary depending on the settings given to the simulation, by simply editing the Tcl script. Future investigators can thus leverage the power of ns-2 simulation in this manner to gain more insights into ZRP's approach of routing in ad-hoc networks.

Appendix I - How to Use ZRP on ns-2

The ZRP2002 module can be obtained from the contributed modules section of the ns-2 web page at

http://www.isi.edu/nsnam/ns/ns-contributed.html

The all-in-one version 2.1b9 of ns-2 is found at

http://www.isi.edu/nsnam/ns/ns-build.html#allinone

Assumptions:

- We will be installing version ns-2.1b9 allinone in your home directory (not ns-2.1b9a or any other version, these are not guaranteed to work as of this writing)

- If you want to install somewhere else, make sure you untar both ns2 and zrp while in the same directory. The zrp tarball assumes the directory structure of ns-2 is in place. See further note on untarring under installing ZRP2002 below.

I. Install ns-2

- download ns-allinone-2.1b9.tar.gz to your home directory
- type "tar xzvf ns-allinone-2.1b9.tar.gz" to untar archive
- type "cd ns-allinone-2.1b9" to enter the top directory
- type "./install" to install, compiling may take up to twenty minutes
- edit your ~/.bashrc, add following lines

```
#NS2 specifics
export PATH=$HOME/ns-allinone-2.1b9/bin:$HOME/ns-allinone-2.1b9/ns-2.1b9/zrp:$PATH
export LD_LIBRARY_PATH=$HOME/ns-allinone-2.1b9/otcl-1.0a7:/ns2/ns-allinone-2.1b8a/lib
export LD_LIBRARY_PATH=$HOME/ns-allinone-2.1b9/otcl-1.0a8:/ns2/ns-allinone-2.1b9/lib
export TCL_LIBRARY=$HOME/ns-allinone-2.1b9/tcl8.3.2/library
export TCLSH=$HOME/ns-allinone-2.1b9/bin/tclsh8.3
```

- re-login in that terminal window to make sure settings changes are implemented type "set" to see a listing, check if the above paths are in your environment.

II. Install ZRP2002

- Download zrp2002.tar.gz from the contributed module section of the ns-2 site to your home directory.

- Type "tar xzvf zrp2002.tgz" to untar zrp archive, the code is automatically inserted in the right places in the directory tree (if you untarred zrp2002.tgz before ns2, you need to untar zrp2002.tgz again to make sure the zrp archived files over-write the corresponding ns2 code)

- Edit Makefile at ns-allinone-2.1b9/ns-2.1b9/Makefile, and at line 209 (after "dsdv/dsdv.o dsdv/rtable.o queue/rtqueue.o \") insert the zrp code objects:

```
	dsdv/dsdv.o dsdv/rtable.o queue/rtqueue.o \
	zrp/zrp.o zrp/zrp_table.o \
	routing/rttable.o \
```

Make sure there is a tab (the keystroke, not the word "tab") at the beginning of the line.

- Now we need to modify some ns2 code. The files in the directory "zrp/otherfiles" are the files that will be changed. You can change them using the patch method (recommended for any version of ns2 "close" to ns-2.1b9) or just edit these files manually to make the changes. (hint: if you want to find out the difference between two files, use "diff file1 file2")

By patching:
  % cd ~/ns-allinone-2.1b9 # (the patch has to be run from here)
  % patch -p1 < ns-2.1b9/zrp/otherfiles/ns2-zrp.patch

- Now type:

% cd ~/ns-allinone-2.1b9/ns-2.1b9
% touch common/* tcl/lib/* trace/*

To make sure we re-compile newly added code. (for some reason this wasn't always enough, I had to actually either touch a file individually by name or edit the file and add a space anywhere in the file to force the Makefile to re-compile that file, especially the files trace/cmu-trace.*)

- Type "make -k" within the "~/ns-allinone-2.1b9/ns-2.1b9/" directory, the ns-2 code should re-compile

- If no errors (there might be some warnings), then run the demo script:

% ns ~/ns-allinone-2.1b9/ns-2.1b9/zrp/demo/demo3_rad3.tcl

num_nodes is set 23

_ 0_ [0.000000] | Node 0 was created. ! Neighbor Table: empty ! LinkTable: empty ! Routes empty Periph[]

_ 1_ [0.000000] | Node 1 was created. ! Neighbor Table: empty ! LinkTable: empty ! Routes empty Periph[]

_ 2_ [0.000000] | Node 2 was created. ! Neighbor Table: empty ! LinkTable: empty ! Routes empty Periph[]

. . . .

_ 1_ [100.007170] | Got IARP_Data packet orig=10 dest=9 (>>> forwarding to next hop 2 ! Neighbor Table: 0 2 ! LinkTable: 10=11 0=1 5=6 20=10 22#20 0=22 4=5 2=1 3=2 3=4 ! Routes [2 1] [0 1] [3 2 1] [22 0 1] [4 3 2 1] Periph[4]

_ 2_ [100.008868] | Got IARP_Data packet orig=10 dest=9 (>>> forwarding to next hop 3 ! Neighbor Table: 3 1 ! LinkTable: 10=11 5=6 1#2 7=21 20=10 22#20 0=22 7=6 4=5 0=1 3=2 7=8 3=4 ! Routes [3 2] [4 3 2] [5 4 3 2] Periph[5]

III. Adding a new parameter

For a parameter that takes no arguments: For example, the command "suspend" sets a flag "suspend_flag_" to be TRUE so that when pkt_send() is called, this flag is tested before the packet is sent. The packet is dropped if the node is suspended.

1. Edit the zrp.cc file and find the command() method.
2. Look for the "suspend" command
3. Edit the command to perform some action for being called from Tcl.
4. Always make sure the command() method returns a TCL_OK for a normal condition, or Tcl will give an error.

For a parameter that takes one argument, the command name is compared with argv[1], and the argument is given by argv[2]. To convert to an integer, call this way:

abc = atoi(argv[2]);

The beacon_period command is a good example of how to do it.

Now to call the function from a Tcl script, here is a Tcl snippet for our two examples:

```
set r_(2) [$node_(2) set ragent_] # gets a pointer to route agent
$ns_ at 12.0 "$r_(2) suspend"     # send suspend command to ragent at time=12 sec
$ns_ at 0.4 "$r_(2) beacon_period 7.0" # change beacon period to 7 at time=0.4 sec
```

Appendix II - Code Listing

```
// ZRP.CC===========================================================
#include <random.h>
#include <object.h>
#include <route.h>
#include <packet.h>
#include <cmu-trace.h>

#include "zrp.h"

#if defined(WIN32) && !defined(snprintf)
#define snprintf _snprintf
#endif /* WIN32 && !snprintf */

// Nominal settings for some parameters:
//- beacon frequency: 5 seconds
//- ack timeout: 2 seconds
//- neighbor timeout: 15 seconds
//- jitter (and where should I apply it): uniformly distributed 0-2 sec;
// should be applied to the beacon transmission

// main defaults
#define TIME_FACTOR 1
#define DEFAULT_TRANSMIT_JITTER 1
#define DEFAULT_STARTUP_JITTER 3
#define DEFAULT_PROCESS_JITTER 1
#define DEFAULT_BEACON_PERIOD 5*TIME_FACTOR  // period of beacon transmission in sec
#define DEFAULT_BEACON_PERIOD_JITTER 0
#define DEFAULT_NEIGHBOR_TIMEOUT 15  //seconds
#define DEFAULT_NEIGHBOR_ACK_TIMEOUT 2*TIME_FACTOR // sec
#define DEFAULT_NEIGHBORTABLE_SCAN_PERIOD 11*TIME_FACTOR // scan table every n sec
#define DEFAULT_IARP_UPDATE_PERIOD 22*TIME_FACTOR // time between periodic update in sec
#define SUSPEND_FLAG FALSE // all in/outgoing packets are dropped if TRUE

#define MAX_ROUTE_LENGTH 1000
#define IERP_TTL 40

#define MAX_NUM_NEIGHBORS 20
#define MAX_LINKIE 10
#define MAX_OUTBOUND_PENDING 20 // max queue for pkts waiting for ierp requests

#define MAX_DETECTED_QUERIES 20 // max queries that can be detected
#define IERP_REQUEST_TIMEOUT 5 // secs before timeout on an ierp request
#define OUTER_ROUTE_EXPIRATION 20 // secs that a cached route will last

#define HDR_BEACON_LEN 20 // bytes
// zone radius related defines in rtable.h

#define IERPCACHE_SCAN_PERIOD 5
        // scan ierp cache every n number of beacon timeout cycles
#define MAX_ROUTE_LENGTH 20
#define ROUTER_PORT      0xff
#define FOUND 1
#define LOST 0
#define TRUE 1
#define FALSE 0


//==========================================================
// ZRP:
// Returns a random number between 0 and max, used for jitter
//==========================================================
static inline double
jitter (double max, int be_random_) {
  return (be_random_ ? Random::uniform(max) : 0);
}

//==========================================================
// ZRP:
```

```
// prints to STDOUT Route table and Peripheral list
//========================================================
void ZRPAgent::print_routes()
{
  rtable_node *j;
  int i;

  printf("Routes ");
  for (i=1; rtable.route[i] != NULL ; i++) {
    printf("[");
    for (j=rtable.route[i]; j!=NULL; j=j->next_  ) {
      printf("%d ",(int)j->id_,j->numhops_);
    }
    printf("] ");
  }

  if (rtable.route[1] == NULL) {
    printf("empty ");
  }

  printf("Periph[");
  for (j=rtable.periphnodes_; j!=NULL; j=j->next_) {
    printf("%d ",(int)j->id_);
  }
  printf("]");
}

// Prints list of neighbors, list of link states, route and peripheral tables
void ZRPAgent::print_tables()
{
  int flag = TRUE;

  printf("! Neighbor Table: ");
  for (neighbor_struct* np=neighbors_;
       np != NULL; np=np->next_ ){
    printf("%d ",np->addr_);
    flag = FALSE;
  }
  if (flag) {
    printf("empty ");
  }

  printf("! ");
  lstable.print_links();

  printf("! ");
  print_routes();

}

//=====================================================================
// ZRP:
// A supporting function used in finding the ZRP header in a packet
// You will most likely not use this directly, used by TCL
//=====================================================================
int hdr_zrp::offset_;
static class ZRPHeaderClass : public PacketHeaderClass {
public:
        ZRPHeaderClass() : PacketHeaderClass("PacketHeader/ZRP",
                                             sizeof(hdr_zrp)) {
          bind_offset(&hdr_zrp::offset_);
        }
} class_zrp_hdr;

//=====================================================================
// ZRP:
// A binding between ZRP and TCL, you will most likely not change this
//=====================================================================
static class ZRPClass : public TclClass {
public:
       ZRPClass() : TclClass("Agent/ZRP") {}
```

```
        TclObject* create(int argc, const char*const* argv) {

                return(new ZRPAgent((nsaddr_t) atoi(argv[4])));
                // Tcl code will attach me and then pass in addr to me
                // see tcl/lib/ns-lib.tcl, under create-zrp-agent,
                // is done by "set ragent [new Agent/ZRP [$node id]]"
        }
} class_zrp;

//==================================================================
// ZRP:
// Default Constructor, not really used
// The Tcl script should call the other Main constructor
//==================================================================
ZRPAgent::ZRPAgent() : Agent(PT_TCP), NeighborScanTimer_(this),
                      BeaconTransmitTimer_(this),suspend_flag_(0) ,
                      PeriodicUpdateTimer_(this), AckTimer_(this),
                      rtable(this),
                       myaddr_(0), myid_('\0'), seq_(1), qid_(1),
                      radius_(DEFAULT_ZONE_RADIUS) {
  // Prashant's Theory on how this works (could be wrong):
  // initialize my address, both int and string forms.
  // this constructor normally isn't called
  // if you decide to use it, my address has to be obtained from
  // some parent Tcl script, ie in my case
  // command("addr id")
  // normally, the other constructor is usually what is called
  myaddr_= this->addr(); // This might not give anything useful back,
  // I do not know my own address, my node does, have to wait for Tcl
  // to connect us and give my my id
  myid_ = Address::instance().print_nodeaddr(myaddr_);
  bind("radius_", (int*)&radius_);

}

//==================================================================
// ZRP:
// Main Constructor for ZRP Agent
//==================================================================
ZRPAgent::ZRPAgent(nsaddr_t id):Agent(PT_TCP), NeighborScanTimer_(this),
                              BeaconTransmitTimer_(this), AckTimer_(this),
                              num_neighbors_(0), suspend_flag_(0) ,
                              PeriodicUpdateTimer_(this), rtable(this),
                              myaddr_(id), myid_('\0'), seq_(1),qid_(1),
                              radius_(DEFAULT_ZONE_RADIUS),
                              transmit_jitter_(DEFAULT_TRANSMIT_JITTER),
                              startup_jitter_(DEFAULT_STARTUP_JITTER),
                              process_jitter_(DEFAULT_PROCESS_JITTER),
                              beacon_period_(DEFAULT_BEACON_PERIOD),
                              beacon_period_jitter_(DEFAULT_BEACON_PERIOD_JITTER),
                              neighbor_timeout_(DEFAULT_NEIGHBOR_TIMEOUT),
                              neighbor_ack_timeout_(DEFAULT_NEIGHBOR_ACK_TIMEOUT),
                              neighbortable_scan_period_(DEFAULT_NEIGHBORTABLE_SCAN_PERIOD),
                              iarp_update_period_(DEFAULT_IARP_UPDATE_PERIOD)
{

  // initialize my address, both int and string forms
  // my node knows its id, but I (ragent) didn't until now (I think?)
  myid_ = Address::instance().print_nodeaddr(myaddr_);
  Time now = Scheduler::instance().clock(); // get the time

  tx_=0; // initialize counter for transmitted packets
  rx_=0; // initialize counter for received packets

  // inits for tables
  // These are so the table classes know each other and their own IP address
  rtable.my_address_ = id;
  lstable.my_address_ = id;
  rtable.linkstatetable = &lstable;

  // Sequence counter starts at one
```

```
  seq_ = 1;

  // This ZRPAgent was just created.
  printf("\n_%2d_ [%6.6f] | Node %d was created. ", myaddr_, (float)now,
         myaddr_);

  print_tables();

  printf("\n");
}

//=================================================================
// ZRP:
// Here are bindings for parameters you can tweak from TCL.
// The first set take no arguments.
// The second set take one argument.
//=================================================================
int
ZRPAgent::command (int argc, const char *const *argv) {
  Time now = Scheduler::instance().clock(); // get the time

  // First set
  if (argc == 2) { // No argument from TCL
    if (strcmp (argv[1], "start") == 0) { // Init ZRP Agent
      startup();
      return (TCL_OK);
    }
    else if (strcasecmp (argv[1], "ll-queue") == 0) { // who is my ll
      if (!(ll_queue = (PriQueue *) TclObject::lookup (argv[2]))) {
        fprintf (stderr, "ZRP_Agent: ll-queue lookup "
                 "of %s failed\n", argv[2]);
        return TCL_ERROR;
      }
      return TCL_OK;
    } else if (strcmp (argv[1], "suspend") == 0) { // turns off receipt of pkts
      suspend_flag_ = TRUE;
        printf("_%2d_ [%6.6f] | Node %d suspended. ",
               myaddr_,now, myaddr_);
        print_tables();
        printf("\n");
      return (TCL_OK);
    } else if (strcmp (argv[1], "unsuspend") == 0) { // turns it back on
      suspend_flag_ = FALSE;
        printf("_%2d_ [%6.6f] | Node %d un-suspended. ",
               myaddr_,now, myaddr_);
        print_tables();
        printf("\n");
      return (TCL_OK);
    }

    // Second set
  } // if argc == 2
  else if (argc == 3)   { // One argument from TCL
    if (strcasecmp (argv[1], "radius") == 0) {
      // change the radius, takes (int) num hops
      int temp;
      temp = atoi(argv[2]);
      if (temp > 0) { // don't change radius unless input is valid value
        printf("_%2d_ [%6.6f] | Radius change from %d to %d ",myaddr_,now,
               radius_, temp);
        print_tables();
        printf("\n");
        radius_ = temp;
      }
      return TCL_OK;
    }

    // change beacon period, takes (int) number of secs
    if (strcasecmp (argv[1], "beacon_period") == 0) {
      int temp;
      temp = atoi(argv[2]);
```

```
    if (temp > 0) { // don't change unless input is valid value
      printf("_%2d_ [%6.6f] | Beacon period change from %d to %d ",
             myaddr_,now, beacon_period_, temp);
      print_tables();
      printf("\n");
      beacon_period_ = temp;
    }
    return TCL_OK;
  }

  // change beacon period jitter, takes (int) number of secs
  if (strcasecmp (argv[1], "beacon_period_jitter") == 0) {
    int temp;
    temp = atoi(argv[2]);
    if (temp > 0) { // don't change unless input is valid value
      printf("_%2d_ [%6.6f] | Beacon period jitter change from %d to %d ",
             myaddr_,now, beacon_period_jitter_, temp);
      print_tables();
      printf("\n");
      beacon_period_jitter_ = temp;
    }
    return TCL_OK;
  }

  // change neighbor timeout, takes (int) number of secs
  if (strcasecmp (argv[1], "neighbor_timeout") == 0) {
    int temp;
    temp = atoi(argv[2]);
    if (temp > 0) { // don't change unless input is valid value
      printf("_%2d_ [%6.6f] | Neighbor timeout change from %d to %d ",
             myaddr_,now, neighbor_timeout_, temp);
      print_tables();
      printf("\n");
      neighbor_timeout_ = temp;
    }
    return TCL_OK;
  }

  // change neighbor ack timeout, takes (int) number of secs
  if (strcasecmp (argv[1], "neighbor_ack_timeout") == 0) {
    int temp;
    temp = atoi(argv[2]);
    if (temp > 0) { // don't change unless input is valid value
      printf("_%2d_ [%6.6f] | Neighbor timeout change from %d to %d ",
             myaddr_,now, neighbor_ack_timeout_, temp);
      print_tables();
      printf("\n");
      neighbor_ack_timeout_ = temp;
    }
    return TCL_OK;
  }

  // resets myid_ to correct address
  if (strcasecmp (argv[1], "addr") == 0) {
    char str;
    myaddr_ = Address::instance().str2addr(argv[2]);
    delete myid_; // remove old string @ myid_, avoid memory leak
    myid_ = Address::instance().print_nodeaddr(myaddr_);
    return TCL_OK;
  }

  // Error if we cannot find TclObject
  TclObject *obj;
  if ((obj = TclObject::lookup (argv[2])) == 0) {
    fprintf (stderr, "%s: %s lookup of %s failed\n", __FILE__, argv[1],
             argv[2]);
    return TCL_ERROR;
  }

  // Returns pointer to a trace target
  if (strcasecmp (argv[1], "tracetarget") == 0)  {
```

```
      tracetarget = (Trace *) obj;
      return TCL_OK;
    }
    // Returns pointer to our node
    else if (strcasecmp (argv[1], "node") == 0) {
      node_ = (MobileNode*) obj;
      return TCL_OK;
    }
    // Returns pointer to port-demux
    else if (strcasecmp (argv[1], "port-dmux") == 0) {
      port_dmux_ = (NsObject *) obj;
        return TCL_OK;
    }
  }  // if argc == 3

  return (Agent::command (argc, argv));
}

//=====================================================================
// ZRP:
// This is run only when ns-lib.tcl does a "start", see first if-statement
// under command() which calls this
//=====================================================================
void
ZRPAgent::startup() {

  // give to rtable the pointer to lstable, a form of call-back
  rtable.linkstatetable = &lstable;

  // pgi - blank all tables
  for (int i=0; (i< MAX_NODES_PER_ZONE); i++)
    rtable.route[i] = NULL;
  rtable.periphnodes_=NULL;
  rtable.agent_ = this;
  for (int i=0; i<MAX_QUERY_DETECT_TABLE_SIZE; i++)
    queries_[i] = NULL;

  // Start timers
  // The timers start at slightly different times. You can also just remove
  // the "start ragent" under create-zrp-agent in ns-lib.tcl
  // and start agents manually in your simulation Tcl script
  // don't forget do a make depend for any changes to tcl/lib/*
  double ttt;
  ttt =  Random::uniform(startup_jitter_);
  NeighborScanTimer_.start(ttt);

  ttt =  Random::uniform(startup_jitter_);
  BeaconTransmitTimer_.start(ttt);

  ttt =  Random::uniform(startup_jitter_);
  PeriodicUpdateTimer_.start(ttt);

  // erase ierp route cache
  for (int i=0; i<MAX_IERP_CACHE; i++) {
    ierpcache[i] = NULL;
  }

  // Erase Outer route list
  for (int i=0; i<MAX_NODES_PER_ZONE+1; i++)
      rtable.route[i] = NULL;
  for (int i=0; i<MAX_NODES_PER_ZONE+1; i++)
    rtable.outer_route[i] = NULL;

  // init route table to know its own address
  rtable.my_address_=myaddr_;
}
//=========================================================================
// P a c k e t   Ro u t i n e s
//
//=========================================================================
// Packet routines:
```

```
// Creates a fresh new packet and initializes as much as it knows how.
//=======================================================================
Packet* ZRPAgent::pkt_create(ZRPTYPE zrp_type, nsaddr_t addressee, int ttl)
{

  Scheduler & s = Scheduler::instance();  // Useful to send (Schedule) packets
  Packet *p = allocpkt(); // fresh new packet
  hdr_ip *hdrip = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN(p);
  hdr_zrp *hdrz = HDR_ZRP(p);

  hdrc->ptype() = PT_ZRP;   // ZRP pkt type
  hdrc->next_hop() = addressee;
  hdrc->addr_type_ = NS_AF_NONE;
  hdrc->size() = IP_HDR_LEN ;  //  set default packet size
  hdrip->ttl() = ttl;
  hdrip->saddr() = myaddr_;        // source address
  hdrip->sport() = ROUTER_PORT;    // source port
  hdrip->daddr() = addressee;      // dest address
  hdrip->dport() = ROUTER_PORT;    // dest port
  hdrz->zrptype_ = zrp_type;       // which zrp pkt am I?
  hdrz->numentries_ = 0;           // no entries in data space
  hdrz->seq_ = seq_;               // copy from gobal sequence counter
  hdrz->forwarded_ = 0;            // have not been forwarded before
  hdrz->originator_ = myaddr_;     // I am originator, this is used by NDP/IARP
  hdrz->routeindex_=0;             // where in the route list am I sending ?
  // init to first hop

  // thought I needed this?! but I don't
  // Looks like free() does take care of an already freed space ok, but I leave
  // this here in case your compiler acts differently
  //hdrz->route_ = (nsaddr_t *)malloc(sizeof(nsaddr_t));
  //hdrz->links_ = (linkie *)malloc(sizeof(linkie));

  inc_seq(); // increments global sequence counter
  return(p);
}

//==========================================================
// Packet Routines:
// Dynamically allocates memory for route data in a packet
// given size is number of nodes in route list
//==========================================================
void ZRPAgent::pkt_create_route_data_space(Packet *p, int size) {
  // size is in number of nsaddr_t
  hdr_zrp *hdrz = HDR_ZRP(p);

  free(hdrz->route_); // clear out if there is data already
  hdrz->route_ = (nsaddr_t *)malloc(size*sizeof(nsaddr_t) ) ;
}

//==========================================================
// Packet Routines:
// Frees allocated memory for route data in a packet
//==========================================================
void ZRPAgent::pkt_free_route_data_space(Packet *p) {
  hdr_zrp *hdrz = HDR_ZRP(p);
  free(hdrz->route_);
}

//==========================================================
// Packet Routines:
// Dynamically allocates memory for link state data
// in a packet, given size is number of links
//==========================================================
void ZRPAgent::pkt_create_link_data_space(Packet *p, int size) {
  // size is in number of linkie
  hdr_zrp *hdrz = HDR_ZRP(p);
  free(hdrz->links_); // clear out if there is data already
  hdrz->links_ = (linkie *)malloc(size*sizeof(linkie) ) ;
}
```

```
//=========================================================
// Packet Routines:
// Frees allocated memory for link state data in a packet
//=========================================================
void ZRPAgent::pkt_free_link_data_space(Packet *p) {
  hdr_zrp *hdrz = HDR_ZRP(p);
  free(hdrz->links_);
}

//=========================================================================
// Packet Routines:
// Copy an entire packet to another.
//=========================================================================
void ZRPAgent::pkt_copy(Packet *pfrom, Packet *pto) {
  hdr_cmn *hdrcfrom = HDR_CMN(pfrom);
  hdr_ip *hdripfrom = HDR_IP(pfrom);
  hdr_zrp *hdrzfrom = HDR_ZRP(pfrom);
  hdr_cmn *hdrcto = HDR_CMN(pto);
  hdr_ip *hdripto = HDR_IP(pto);
  hdr_zrp *hdrzto = HDR_ZRP(pto);
  int size;
  nsaddr_t *ptrto;
  nsaddr_t *ptrfrom;

  size = hdrzfrom->numentries_; //
  hdrcto->direction() = hdrcfrom->direction();
  hdrcto->ptype() = hdrcfrom->ptype();
  hdrcto->next_hop() = hdrcfrom->next_hop();
  hdrcto->addr_type_ = hdrcfrom->addr_type_;
  hdrcto->size() = hdrcfrom->size() ;

  hdripto->ttl() = hdripfrom->ttl();
  hdripto->saddr() = hdripfrom->saddr();
  hdripto->sport() = hdripfrom->sport();
  hdripto->daddr() =  hdripfrom->daddr();
  hdripto->dport() = hdripfrom->dport();

  hdrzto->zrptype_ = hdrzfrom->zrptype_;
  hdrzto->seq_ = hdrzfrom->seq_;
  hdrzto->numentries_ = hdrzfrom->numentries_;
  hdrzto->originator_ = hdrzfrom->originator_;
  hdrzto->ierpsource_ = hdrzfrom->ierpsource_;
  hdrzto->ierpdest_ = hdrzfrom->ierpdest_;
  hdrzto->iarpsource_ = hdrzfrom->iarpsource_;
  hdrzto->iarpdest_ = hdrzfrom->iarpdest_ ;
  hdrzto->lastbordercaster_ = hdrzfrom->lastbordercaster_;
  hdrzto->pktsent_ = hdrzfrom->pktsent_;
  hdrzto->enc_dport_ = hdrzfrom->enc_dport_;
  hdrzto->enc_daddr_ = hdrzfrom->enc_daddr_;
  hdrzto->enc_ptype_ = hdrzfrom->enc_ptype_;
  hdrzto->qid_ = hdrzfrom->qid_;
  hdrzto->forwarded_ = hdrzfrom->forwarded_;
  hdrzto->routeindex_ = hdrzfrom->routeindex_;

  //free(ptrto);
  pkt_create_route_data_space(pto,size);

  ptrto = hdrzto->route_;
  ptrfrom = hdrzfrom->route_;

  // copy route entries
  for (int i=0; i<size; i++) {
    ptrto[i] = ptrfrom[i];
  }
  // we don't need to copy link data, but if you need to,
  // place code here
}

//=========================================================================
// Packet routines:
```

```
// Adds a link state entry to a packet data space
//========================================================================
void ZRPAgent::pkt_add_link(Packet* p, nsaddr_t lnksrc, nsaddr_t lnkdest,
                            int isup) {
  linkie* ls;
  neighbor_struct* ns;
  int numentries;
  int* seq_stamp;
  hdr_zrp *hdrz = HDR_ZRP(p);

  // current number of entries in our data space
  numentries = hdrz->numentries_;

  // this assumes space has ALREADY been allocated
  // pointer to head of link state entries
  ls = hdrz->links_;

  // add at end of entry list
  ls[numentries].src_ = lnksrc;
  ls[numentries].dest_ = lnkdest;
  ls[numentries].isup_ = isup;
  hdrz->numentries_++; // we just added one
}

//========================================================================
// Packet routines:
// Add all neighbors link states to this packet p
//========================================================================
void ZRPAgent::pkt_add_all_links(Packet* p) {
  int* num_entries;
  linkie* ls;
  int neighbor_cnt;
  int seq_stamp;
  hdr_zrp *hdrz = HDR_ZRP(p);
  neighbor_struct* ns;

  // ptr to data area of pkt
  ls = hdrz->links_;

  // place link states consecutively into space
  neighbor_cnt = 0;
  for (ns=neighbors_; ns != NULL; ns=ns->next_) {
    ls[neighbor_cnt].src_ = myaddr_;
    ls[neighbor_cnt].dest_ = ns->addr_;
    ls[neighbor_cnt].isup_ = TRUE; // all on my neighbors are "up"
    neighbor_cnt++;
  }
  hdrz->numentries_ = neighbor_cnt; // will be zero if no neighbors
}

//========================================================================
// Packet routines:
// Send packet, ie schedule it for sometime in future to target_ of ZRP
// which is handled by LL. This is unicast only to "addressee"
//========================================================================
void ZRPAgent::pkt_send(Packet* p, nsaddr_t addressee)
{
  Scheduler & s = Scheduler::instance();  // Useful to send (Schedule) packets
  hdr_ip *hdrip = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN(p);
  hdr_zrp *hdrz = HDR_ZRP(p);

  // if this node is suspended, drop the packet
  if (suspend_flag_ == TRUE) {
    zdrop(p, DROP_RTR_IARP);
    return;
  }

  // set next hop and dest to addressee
  hdrc->next_hop() = addressee;
  hdrip->daddr() = addressee;
```

```
  // transmit, no delay
  s.schedule(target_, p, 0);

  tx_++; // increment pkt transmitted counter

}

//======================================================================
// Packet routines:
// Loads link update information indexed by "index" from pkt data
// into link struct
//======================================================================
void ZRPAgent::pkt_read_update(Packet* p, link_struct *ls, int index ) {
  hdr_zrp *hdrz = HDR_ZRP(p);
  linkie* ptr;
  link_struct* lnew;

  ptr = hdrz->links_;  // ptr to data area of pkt

  ls->src_ = ptr[index].src_ ;
  ls->dest_ = ptr[index].dest_ ;
  ls->isup_ = ptr[index].isup_ ;
}

//======================================================================
// Packet routines:
// Broadcast packet to all neighbors
//======================================================================
void ZRPAgent::pkt_broadcast(Packet* p) {
 Scheduler & s = Scheduler::instance();  // Useful to send (Schedule) packets

  hdr_ip *hdrip = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN(p);
  hdr_zrp *hdrz = HDR_ZRP(p);

  // if this node is suspended, drop the packet
  if (suspend_flag_ == TRUE) {
    zdrop(p, DROP_RTR_IARP);
    return;
  }
  hdrc->next_hop() = IP_BROADCAST;
  hdrip->daddr() = IP_BROADCAST;
  // transmit with jitter
  s.schedule(target_, p, Random::uniform(transmit_jitter_) );
  tx_++;
}

//======================================================================
// Packet routines:
// Add entry (indexed by "entry") for "node" to route list in packet
// The caller HAS to update hdrz->numentries_ .
//======================================================================
void ZRPAgent::pkt_add_node(Packet* p, nsaddr_t node, int entry)
{
  int num_entries;
  nsaddr_t *ptr;
  hdr_zrp *hdrz = HDR_ZRP(p);
  hdrz->route_[entry] = node;
}

//======================================================================
// Packet routines:
// returns the address of the next hop on the route list of a packet
// If error, returns the first node on list
//======================================================================
nsaddr_t ZRPAgent::find_next_hop(Packet* p)
{
  hdr_zrp *hdrz = HDR_ZRP(p);
  nsaddr_t *ptr;
  nsaddr_t junk;
```

```
  int junkie ;
  int i;
  i=hdrz->routeindex_;
  return(hdrz->route_[i+1]);

  junkie = hdrz->numentries_;

  if (hdrz->numentries_ > 0) {

    ptr = hdrz->route_;

    for (int i = 0; i<hdrz->numentries_-1; i++) {

      if (ptr[i] == myaddr_) {
        junk = ptr[i+1];

        return(junk);
      }
    }
  } else {
    return(-1);
  }

  // default, just return the node first on list
  return(hdrz->route_[0]);
  // ie no next hop
}

//======================================================================
// Packet routines:
// Sends packet to next hop on the route list of a packet
//======================================================================
void ZRPAgent::send_next_hop(Packet *p) {
  hdr_zrp *hdrz = HDR_ZRP(p);
  int i;
  hdrz->routeindex_++;
  i=hdrz->routeindex_;
  pkt_send(p,hdrz->route_[i]);
}

//======================================================================
// Packet routines:
// Sends packet to previous hop on the route list of a packet
//======================================================================
void ZRPAgent::send_prev_hop(Packet *p) {
  hdr_zrp *hdrz = HDR_ZRP(p);
  int i;
  hdrz->routeindex_--;
  i=hdrz->routeindex_;
  pkt_send(p,hdrz->route_[i]);
}

//======================================================================
// Packet routines:
// Returns the address of the previous hop on the route list of a packet.
// If error, returns the current node's address
//======================================================================
nsaddr_t ZRPAgent::find_prev_hop(Packet* p) {
  hdr_zrp *hdrz = HDR_ZRP(p);
  nsaddr_t *ptr;
  int i;
  i=hdrz->routeindex_;
  return(hdrz->route_[i-1]);


  if (hdrz->numentries_ < 2)
    return(myaddr_);  // need more than 1 entry
  ptr = hdrz->route_;

  for (int i = 1; i<hdrz->numentries_; i++) {
```

```
    if (ptr[i] == myaddr_) {
      return(ptr[i-1]);
    }
  }
  return(myaddr_); // ie no prev hop
}

//======================================================================
// Packet routines:
// Wrapper for drop().
// Drops packet after properly free dynamic memory allocation.
//======================================================================
void ZRPAgent::zdrop(Packet *p, const char *s) {
  hdr_zrp *hdrz = HDR_ZRP(p);

  drop(p,s);
  return;

  if (hdrz->links_ != NULL)
    free((void *)hdrz->links_); // extra: free links_ data
  if (hdrz->route_ != NULL)
    free((void *)hdrz->route_); // extra: free route_ data

}

//======================================================================
// Packet routines:
// Adds an outer route to destination "daddr" to a route list in a packet
//======================================================================
int ZRPAgent::add_outer_route(Packet* p, nsaddr_t daddr) {
  nsaddr_t tmp[100];
  int flag = FALSE;
  hdr_zrp *hdrz = HDR_ZRP(p);
  int cntr = 0;
  int gg;

  for ( int i = 0; i < MAX_NODES_PER_ZONE && !flag ; i++) {

    if (rtable.outer_route[i] != NULL && rtable.outer_route[i]->id_ == daddr) {
      // found outer route
      flag = TRUE;
      printf("| Added outer route ");

      // by definition: numhops+1 = numentries
      gg = rtable.outer_route[i]->numhops_+1 ;
      pkt_create_route_data_space(p, gg);

      nsaddr_t *zroute = hdrz->route_;

      for (rtable_node *ptr=rtable.outer_route[i]->next_; ptr != NULL;
          ptr=ptr->next_) {
       zroute[cntr] = ptr->id_;
       printf("%d[%d] ",cntr,zroute[cntr]);
       cntr++;
      } // end for ptr
      printf("\n");
      hdrz->numentries_ = cntr;
    } // end if
  } // end for i
  return(flag);
}

//===================================================================
// Timers: NeighborScanTimer
// Starts NeighborScanTimer, called by startup(), delayed by thistime
// Protocols: NDP
//===================================================================
void ZRPNeighborScanTimer::start(double thistime) {
  Scheduler::instance().schedule(this, &intr, thistime );
}
```

```
//=================================================================
// Timers: ZRPBeaconTransmitTimer
// Starts BeaconTransmitTimer, called by startup(), delayed by this time
//=================================================================
void ZRPBeaconTransmitTimer::start(double thistime) {
  Scheduler::instance().schedule(this, &intr, thistime );
}

//=================================================================
// Timers: PeriodicUpdateTimer
// Starts PeriodicUpdateTimer, called by startup(), delayed by thistime
// Protocols: IARP
//=================================================================
void ZRPPeriodicUpdateTimer::start(double thistime) {
  Scheduler::instance().schedule(this, &intr, thistime );
}

//=================================================================
// Timers: BeaconTransmitTimer
// Triggered when it is time to Transmit a Beacon
//=================================================================
void ZRPBeaconTransmitTimer::handle(Event* e) {
  Time now = Scheduler::instance().clock(); // get the time
  char* address;
  Packet* p;
  hdr_zrp *hdrz;

  // Note: we need to specify agent-> to access ZRPAgent methods
  // for all timer classes

  // transmit beacon
  p = agent->pkt_create(NDP_BEACON, IP_BROADCAST, 1); // last arg is TTL
  hdrz = HDR_ZRP(p); // access ZRP part of pkt header
  hdrz->originator_ = agent->myaddr_; // I am originator of pkt
  agent->pkt_broadcast(p); // broadcast pkt
  printf("\n_%2d_ [%6.6f] | Node %d sent a beacon (seq no. %d). "
         "| Node %d started an ack-timer to expire at %6.6f sec. ",
         agent->myaddr_, now, agent->myaddr_, hdrz->seq_,
         agent->myaddr_, (float)now + agent->neighbor_ack_timeout_);

  printf("| radius is %d ",agent->radius_);

  agent->print_tables();

  printf("\n");

  agent->AckTimer_.start();   // start ack timer for the beacon sent

  // check outer route requests for expirations
  agent->outer_route_expiration();
  // check ierp cache for expirations
  agent->scan_ierp_cache();
  // check query detect for expirations
  agent->scan_query_detections();
  // check linkstate expiries, mark for purging later
  agent->scan_linkstate_table();

  // schedule next scan in BEACON_PERIOD+jitter() sec
  Scheduler::instance().schedule(this, &intr, agent->beacon_period_ +
                              Random::uniform(agent->beacon_period_jitter_)
                              );
}

//===================================================================
// Timers: Scanning Query Detect Table
// Scan query detect table, delete expired entries.
//===================================================================
void ZRPAgent::scan_linkstate_table() {
 Time now = Scheduler::instance().clock(); // get the time
 link_struct *ls;
```

```
 // loop through whole link state table
 for (ls=lstable.lshead_ ; ls->next_ != NULL; ls=ls->next_) {
   if (ls->next_->expiry_ < now) // if past expiration stamp
     ls->next_->isup_ == FALSE; // mark for purging later
 }

}

//==================================================================
// Timers: Scanning Query Detect Table
// Scan query detect table, delete expired entries.
//==================================================================
void ZRPAgent::scan_query_detections() {
  QueryDetectEntry *entr;
  int i;
  rtable_node *tmp;
  Time now = Scheduler::instance().clock(); // get the time

  for (i=0; i<MAX_QUERY_DETECT_TABLE_SIZE; i++) {
    // if entry is not NULL and has expired, delete
    if ( (queries_[i] != NULL) && (queries_[i]->expiry_ < now) ) {

      // delete entire list of covered nodes
      for (rtable_node *p=queries_[i]->next_; p!=NULL; ) {
        tmp = p;
        p=p->next_;
        delete tmp;
      }

      // delete entry
      delete queries_[i];
      queries_[i] = NULL;
    } // endif
  }
}

//==================================================================
// IERP:
// Checks all outer routes to see if they have expired, deletes those
// that have. Triggered by timer.
//==================================================================
void ZRPAgent::outer_route_expiration() {
  Time now = Scheduler::instance().clock(); // get the time
  rtable_node *tmp;
  int i;
  int flag;
  flag = FALSE;

  printf("\n_%2d_ [%6.6f] | Checking outer route expirations. ",myaddr_,now);

  // Check table
  for (  i = 0; i < MAX_OUTER_ROUTES ; i++) {

    if (rtable.outer_route[i] != NULL && rtable.outer_route[i]->id_ != -99 &&
        rtable.outer_route[i]->expiry_ < now) { // if expired
      flag=TRUE;

      printf("| Outer route for node %d expired at %6.3f. "
             "It has been DELETED. Route = ",
             rtable.outer_route[i]->id_, rtable.outer_route[i]->expiry_);

      // delete the outer route entry - it is a linked list
      for (rtable_node *ptr=rtable.outer_route[i]->next_; ptr!=NULL; ) {
        tmp = ptr;
        printf("%d ",tmp->id_);
        ptr=ptr->next_;
      }

      // mark deleted
      rtable.outer_route[i]->id_ = -99;
```

```
    }
  }
  if (!flag)
    printf("| None expired.");

  printf("\n");

}
//=============================================================
// Timers: NeighborScanTimer
// Triggered by alarm for scanning NDP table. If there is a new
// neighbor, update IARP and re-compute route table.
// Protocols: NDP, NDP notifies IARP
//=============================================================
void ZRPNeighborScanTimer::handle(Event* e) {
  Time now = Scheduler::instance().clock(); // get the time
  neighbor_struct* ns,nstemp;
  neighbor_struct* tmp;
  Packet* p;
  nsaddr_t na;
  Time tout;
  int flag = FALSE; // have we deleted a neighbor ?
  int ls;
  int seq_match;
  int* seq_stamp, num_entries;
  hdr_zrp *hdrz;

  // no neighbors yet, nothing to tell IARP
  if (agent->neighbors_ == NULL) {
    printf("\n_%2d_ [%6.6f] | Node %d neighborscan-timer timed out, "
           "but neighbor table is empty. Next scan at %6.6f sec.",
           agent->myaddr_, now, agent->myaddr_,
           now + agent->neighbortable_scan_period_);

    goto EOFF;
  }

  // insert a dummy struct at head to make it work
  neighbor_struct dummy;
  dummy.next_ = agent->neighbors_;
  agent->neighbors_ = &dummy;

  printf("\n_%2d_ [%6.6f] | Node %d neighborscan-timer timed out. "
         "Next scan at %6.6f sec.  ",
         agent->myaddr_, now, agent->myaddr_,
         now + agent->neighbortable_scan_period_);

  // drop timed-out neighbors who didn't ack
  for (ns=agent->neighbors_; ns->next_ != NULL; ) {

    if (ns->next_->expiry_ < now) { // timed out
      na = ns->next_->addr_;
      tout = ns->next_->expiry_;

      printf("| Node %d detected that the entry for Node %d has expired. "
             "Expiry was at %6.6f sec. ",
             agent->myaddr_, na, tout);

      // delete node
      neighbor_struct *temp = ns->next_;
      ns->next_ = ns->next_->next_;
      delete temp;

      if (flag == FALSE) { // if first time we detect a timeout in this for-loop
        p = agent->pkt_create(IARP_UPDATE, IP_BROADCAST, agent->radius_*2-1);
        // allocate area for our update data in packet, we set it to max
        // current number of neighbors, but we only will use number of
        // neighbors that timed out
        agent->pkt_create_link_data_space(p,MAX_LINKIE);
        hdrz = HDR_ZRP(p);
      }
```

```
      // NDP TO IARP: update lstable
      ls = agent->lstable.update(na,agent->myaddr_,FALSE,0,&seq_match) ;

      agent->rtable.compute_routes(agent->myaddr_,agent->radius_);
      // Don't care if my table was updated or not, let's
      // tell all other nodes regardless
      agent->pkt_add_link(p,agent->myaddr_, na, FALSE);
      flag=TRUE; //indicates we have at least one drop and subsequent update
    } else {
      printf("| Node %d detected that the entry for Node %d has not expired. "
             "Expiry will be at %6.6f sec.", agent->myaddr_, ns->next_->addr_,
             ns->next_->expiry_);
      ns=ns->next_;
    }
  }

  // reset neighbors_ to head of list
  agent->neighbors_ = dummy.next_;

  if (flag) { // we had at least one update, increment sequence
    agent->pkt_broadcast(p);
    printf("| Node %d sent update (seq no. %d) about neighbors "
           "dropped ( ", agent->myaddr_, hdrz->seq_);
    for (int i=0; i<hdrz->numentries_; i++) {
      link_struct curr;
      agent->pkt_read_update(p, &curr, i);
      printf("%d ",curr.dest_);
    }
    printf(") due to expiry timeout. ");
  }

 EOFF:
  agent->print_tables();
  printf("\n");
  // re-schedule timer
  Scheduler::instance().schedule(this, &intr, agent->neighbortable_scan_period_);
}


//=====================================================================
// Timers: PeriodicUpdateTimer
// Triggered when it is time to do a link update
// Protocols: IARP
//=====================================================================
void ZRPPeriodicUpdateTimer::handle(Event* e) {
  Scheduler & s = Scheduler::instance();  // Useful to send (Schedule) packets

  agent->do_update();
  Scheduler::instance().schedule(this, &intr, agent->iarp_update_period_ );

}

//=====================================================================
// IARP:
// This actually does the work for doing link update,
// re-computes route table. Called by ZRPPeriodicUpdateTimer::handle().
//=====================================================================
void ZRPAgent::do_update() {
  Time now = Scheduler::instance().clock(); // get the time
  Packet* p;
  hdr_zrp *hdrz = HDR_ZRP(p);
  link_struct templink;

  // clean_link_table_of_down_and_expired_links
  lstable.purge_links();
  // re-compute route table
  rtable.compute_routes(myaddr_, radius_);

  // only send an update if we have neighbors
  if (neighbors_ != NULL) {
```

```
    printf("\n_%2d_ [%6.6f] | Node %d sent periodic update (seq no. %d) "
           "containing ( ", myaddr_, (float)now , myaddr_, seq_);

    p = pkt_create(IARP_UPDATE, IP_BROADCAST, radius_*2-1);

    pkt_create_link_data_space(p,MAX_LINKIE); // make space in packet
    pkt_add_all_links(p); // add entire link state table to packet data space

    // Print link states in pkt data, redundant but for debug purposes
    for (int i=0; i<num_neighbors_; i++) {
      pkt_read_update(p, &templink, i);
      if (templink.isup_ == TRUE)
       printf("%d=%d ",templink.src_,templink.dest_ );
      else
       printf("%d#%d ",templink.src_,templink.dest_ );
    }

    printf("). ");

    pkt_broadcast(p); // broadcast packet

  } else {
    printf("\n_%2d_ [%6.6f] | Node %d has no neighbors to put in this"
           "periodic update.", myaddr_, now, myaddr_);
  }
  print_tables();
  printf("\n");
}

//======================================================================
// Timers:  AckTimer
// Schedule timeout at neighbor_ack_timeout_, just sent a Beacon.
// Protocol: NDP
//======================================================================
void ZRPAckTimer::start() {
  Scheduler::instance().schedule(this, &intr, agent->neighbor_ack_timeout_);
}

//======================================================================
// Timers:  AckTimer
// Handle a scheduled timeout neighbor_ack_timeout_ secs since we sent
// original Beacon.
// Remove neighbor if it is in our list. If any were removed, update IARP &
// send an immediate update ("triggered update").
// Protocols: NDP, a call to IARP
//======================================================================
void ZRPAckTimer::handle(Event* e) {
  Time now = Scheduler::instance().clock(); // get the time
  neighbor_struct* ns,nstemp;
  neighbor_struct* tmp;
  Packet* p;
  nsaddr_t na;
  Time tout;
  Time lastack;
  int flag = FALSE; // have we deleted a neighbor ?
  int ls;
  int seq_match;
  int* seq_stamp, num_entries;
  hdr_zrp *hdrz;

  // no neighbors yet, nothing to tell IARP
  if (agent->neighbors_ == NULL) {
    printf("\n_%2d_ [%6.6f] | Node %d ack-timer timed out, but neighbor"
           "table is empty. ", agent->myaddr_, now, agent->myaddr_);
    agent->print_tables();
    printf("\n");
    return;
  }

  // insert a dummy struct at head
  neighbor_struct dummy;
```

```
dummy.next_ = agent->neighbors_;
agent->neighbors_ = &dummy;

printf("\n_%2d_ [%6.6f] | Node %d ack-timer timed out. ",
       agent->myaddr_, now, agent->myaddr_);

// drop timed-out neighbors who didn't ack
for (ns=agent->neighbors_; ns->next_ != NULL; ) {

  if ( (now - ns->next_->lastack_) > agent->neighbor_ack_timeout_ ) {
    printf("| Node %d did not receive ack from Node %d before timeout. "
           "Lastack expiry was at %6.6f sec. ",
           agent->myaddr_, ns->next_->addr_,
           ns->next_->lastack_+ agent->neighbor_ack_timeout_);

    na = ns->next_->addr_;
    tout = ns->next_->expiry_;
    lastack =  ns->next_->lastack_;

    // delete node
    neighbor_struct *temp = ns->next_;
    ns->next_ = ns->next_->next_;
    delete temp;

    // total number of neighbors
    agent->num_neighbors_ --;

    if (flag == FALSE) { // if occurring for first time

      p = agent->pkt_create(IARP_UPDATE, IP_BROADCAST, agent->radius_*2-1);
      // allocate area for our update data in packet, we set it to max
      // current number of neighbors, but we only will use number of
      // neighbors that timed out
      agent->pkt_create_link_data_space(p,MAX_LINKIE);

      hdrz = HDR_ZRP(p);
    }

    // NDP TO IARP: update lstable
    ls = agent->lstable.update(na,agent->myaddr_,FALSE,0,&seq_match) ;
    agent->rtable.compute_routes(agent->myaddr_,agent->radius_);
    //      if (ls) { Don't care if my table was updated or not, let's
    // tell all other nodes regardless
    agent->pkt_add_link(p,agent->myaddr_, na, FALSE);
    flag=TRUE; //indicates we have at least one drop and subsequent update
    //      }
  } else {
    printf("| Node %d received ack from Node %d before time-out. "
           "Lastack expiry was at %6.6f sec.", agent->myaddr_, ns->next_->addr_,
           ns->next_->lastack_ + agent->neighbor_ack_timeout_);
    /// ns->next_->lastack_ = now; // update neighbor lastack WRONG
    // the acktimer and the neighborscantimer routines DO NOT
    // alter the last/expiry field of the neighbor
    // they only read it, and decide to delete the neighbor from the
    // table or not
    ns=ns->next_;
  }
}

// reset neighbors_ to head of list, see hack above
// FUTURE can do this a different way without changing neighbors_
// in for loop, say ns=&dummy ?
agent->neighbors_ = dummy.next_;
//  delete dptr doesn't work!

if (flag) { // we had at least one update, send
  agent->pkt_broadcast(p);
  printf("| Node %d sent update (seq no. %d) about neighbors "
         "dropped ( ", agent->myaddr_, hdrz->seq_);
  for (int i=0; i<hdrz->numentries_; i++) {
    link_struct curr;
```

```
      agent->pkt_read_update(p, &curr, i);
      printf("%d ",curr.dest_);
    }
    printf(") due to ack timeout. ");

  }

 EOFF:
  agent->print_tables();
  printf("\n");
}

//=========================================================================
// Timers:  Scanning IERP Cache
// If any IERP requests time out, delete them.
// Protocol: IERP
//=========================================================================
void ZRPAgent::scan_ierp_cache() {
  IERPCacheEntry *entr;
  int i;
  Time now = Scheduler::instance().clock(); // get the time

  for (i=0; i<MAX_IERP_CACHE; i++) {
    if ( (ierpcache[i] != NULL) && (ierpcache[i]->expiry_ < now) ) {
      zdrop(ierpcache[i]->pkt_ ,DROP_RTR_NO_ROUTE);
      delete ierpcache[i];  // the request is dead
      ierpcache[i] = NULL;
    }
  }
}

//=========================================================================
// IERP:
// Pass route in proute to iarp to add to cached packet and send,
// and also save route in outer_route table
// Protocol: IERP, notifies IARP
//=========================================================================
void ZRPAgent::ierp_notify_iarp(Packet *proute) {
  hdr_zrp *hdrzroute = HDR_ZRP(proute);
  nsaddr_t junk,daddr;
  Time now = Scheduler::instance().clock(); // get the time
  int dst, st,cntr;
  int found,flag;
  rtable_node *ptr;
  nsaddr_t *zroute;
  int numentries,qid;

  zroute = hdrzroute->route_;
  numentries = hdrzroute->numentries_;
  qid = hdrzroute->qid_;

  // 1. copy new route to outer_route table
  // 2. find cached packet
  // 3. if not found, return "drop cached packet, delete cache entry" is WRONG
  // 4. if found, add_outer_route to it and send it

  // 1. copy new outer_route to outer_route table

  // flag indicates true if we find a free spot
  flag = FALSE;

  // find a free spot in outer_route table
  for ( int i = 0; i < MAX_OUTER_ROUTES && !flag ; i++) {
    if (rtable.outer_route[i] == NULL) {
      flag = TRUE;
      dst = i;

      // If we happen to find the route in table, keep the shorter route
    } else if (rtable.outer_route[i]->id_ == hdrzroute->ierpdest_) {
      dst=i;
```

```
    printf("| Outer route already in ierp cache, ");

    // keep the shorter route
    // by definition: numhops+1 = numentries
    // if length of store route < length of newly received accum route
    if (rtable.outer_route[dst]->numhops_+1 <= numentries) {
      printf("newly received route is longer or the same, keeping "
             "stored route. ");
      return;
    }

    // the printf explains the next section
    printf("new route is shorter, replacing stored with new route [");
    // delete old route
    rtable_node *tmp;
    for (ptr=rtable.outer_route[dst]->next_; ptr!=NULL; ){
      tmp = ptr;
      ptr=ptr->next_;
      delete tmp;
    }

    rtable_node * nptr = NULL;

    // create list backward
    for (int i=numentries-1; i >=0; i--) {
      //printf("%d ",zroute[i]);
      tmp = new rtable_node;
      tmp->id_ = zroute[i];
      tmp->next_ = nptr;
      nptr=tmp;
    }

    // put new route list into route entry
    rtable.outer_route[dst]->next_ = nptr;
    for (rtable_node *ptr=rtable.outer_route[dst]->next_; ptr!=NULL;
       ptr=ptr->next_)
    printf("%d ",ptr->id_);

    printf("].\n");

    return; // pkt deleted upon return
  }

}

// Our outer route table is full

if (!flag) {
  printf("No more room in outer_route table for new routes!\n");
  zdrop(proute,DROP_RTR_IARP); // drop route reply
  return;
}

assert(flag); // if no more room in outer_route table, need to increase

// create new head for entry
rtable.outer_route[dst] = new rtable_entry;

// copy the dest id to the header node, stamp with expiry, add numhops
rtable.outer_route[dst]->id_ = zroute[numentries - 1];
rtable.outer_route[dst]->expiry_ = now + OUTER_ROUTE_EXPIRATION;
rtable.outer_route[dst]->numhops_ = numentries - 1;

// copy first node address of route list
ptr = new rtable_node;
rtable.outer_route[dst]->next_ = ptr;

// copy rest of route to the linked list in outer_route entry
for (int j=0; j <= rtable.outer_route[dst]->numhops_ ; j++) {
  ptr->next_ = new rtable_node;
  ptr = ptr->next_;
```

```
  ptr->id_ = zroute[j];
  ptr->numhops_ = j;
}
ptr = rtable.outer_route[dst]->next_;
rtable.outer_route[dst]->next_ = ptr->next_;
delete(ptr);

printf("copied route ");
cntr=0;

for (ptr=rtable.outer_route[dst]->next_; ptr !=NULL; ptr=ptr->next_) {
  printf("%d(%d) ",cntr++,ptr->id_);
} printf("\n");

// 2. find cached packet

assert(numentries > 3); // the route list must at minimum
// include myself + 1 more + 1 more for outer routes at radius=1

// finding cached request, flag = TRUE if found, $found = index
flag = FALSE;
for (int i=0; i<MAX_IERP_CACHE && !flag; i++) {

  if (ierpcache[i] != NULL && qid == ierpcache[i]->qid_) {
      flag = TRUE;
      found = i;
  }
}

if (!flag) {
  // 3. if not found, return
  // no cached route, must have expired
  // should we try again, another request?
  printf("cached ierp request not found, must have expired."
         "Dropping ierp reply.\n");

} else {

  // 4. if found, add_outer_route to it and send it
  Packet *p = ierpcache[found]->pkt_;
  hdr_zrp *hdrz = HDR_ZRP(p);
  hdr_ip *hdrip = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN(p);

  // add new outer route entry to packet
  st = add_outer_route(p, zroute[numentries - 1] );

  //encapsulate upper layer info
  hdrz->enc_daddr_ = hdrip->daddr() ;
  hdrz->enc_dport_ = hdrip->dport() ;
  hdrz->enc_ptype_ =  hdrc->ptype();
  // send to next hop
  hdrc->direction() = hdr_cmn::DOWN; // this is default, but just in case
  hdrc->ptype() = PT_ZRP;
  //hdrc->next_hop() = nto[0];
  hdrc->addr_type_ = NS_AF_NONE;
  hdrc->size() = IP_HDR_LEN ;  //  set default packet size
  hdrip->ttl() = 100;
  //  hdrip->daddr() = find_next_hop ;
  hdrip->dport() = ROUTER_PORT;
  hdrz->zrptype_ = IARP_DATA;
  hdrz->numentries_ = numentries ;
  hdrz->forwarded_ = 0;
  hdrz->originator_ = myaddr_;
  hdrz->pktsent_ = now;

  hdrz->zrptype_ = IARP_DATA;
  hdrc->direction() = hdr_cmn::DOWN;

  junk = find_next_hop(p);
```

```
    printf("| Sending data packet to next hop %d )\n",junk);
    send_next_hop(p);
  }

}

//=========================================================================
// ZRP:
// Receives incoming packet, determines if it is a ZRP packet or not
// If it is, it is processed by ZRP (IERP) at recvZRP.
// If not, that means the packet is coming from the upper layer, and the
// packet needs to be routed using route_pkt.
// Protocols: IERP and IARP
//=========================================================================
void
ZRPAgent::recv(Packet * p, Handler *)  {
  hdr_ip *hdrip = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN(p);
  hdr_zrp *hdrz = HDR_ZRP(p);
  int src = Address::instance().get_nodeaddr(hdrip->saddr());
  int dst = hdrc->next_hop();
  int jem;
  Time now = Scheduler::instance().clock(); // get the time

  rx_++; // just received a new packet

  // check if we are in suspend mode
  if (suspend_flag_ == TRUE) {
        zdrop(p, DROP_RTR_IARP);
       return;
  }

  if (hdrc->ptype() == PT_ZRP) {
    jem =  hdrip->ttl();
    hdrip->ttl() -= 1;
    assert(jem == (hdrip->ttl() + 1) );
    if (hdrip->ttl() < 0) {
      zdrop(p, DROP_RTR_TTL);
      return;
    }

    // it is a proper ZRP packet, let ZRP process it
    recvZRP(p);
    return;
  }

  // if other type
  if(src == myaddr_ && hdrc->num_forwards() == 0) {
    // A packet from higher layer
    hdrc->size() += IP_HDR_LEN;     //  Add the IP Header size
    //   hdrip->ttl_ = IP_DEF_TTL;
    hdrip->ttl() = radius_ ;
  } else if (src == myaddr_) { // rcving a pkt I sent, prob a routing loop
    zdrop(p, DROP_RTR_ROUTE_LOOP);
    return;
  } else {

    // forwarding a non-ZRP packet from another node
    // Check the TTL.  If it is zero, then discard.
    if(--hdrip->ttl() == 0) {
      zdrop(p, DROP_RTR_TTL);
      return;
    }

    // WHAT do I do with non-ZRP packets rcvd from another node?
    // just route_pkt it like any other? then this else clause is superfluous
  }

  printf("\n_%2d_ [%6.3f] | Got non-ZRP packet type %d from upper layer "
         "seq no. %d (daddr=%d dport=%d) ",
         // "(%2d,PT_ZRP=%2d,PT_TCP=%d) ",
```

```
        myaddr_, now, hdrip->daddr(),
        seq_, hdrip->daddr(), hdrip->dport() );

    print_tables();
  printf("\n");

  // This packet is not a ZRP packet, it needs to be routed.
  route_pkt(p, hdrip->daddr() );
}

//========================================================================
// IERP:
// Process received ZRP type packet.
// Protocols: IERP, IERP calling IARP, IARP, NDP, NDP notifying IARP
// IARP calling IERP
//========================================================================
void
ZRPAgent::recvZRP(Packet* p)
{
  hdr_ip *hdrip = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN(p);
  hdr_zrp *hdrz = HDR_ZRP(p);
  int tmp[MAX_ROUTE_LENGTH];
  int src = Address::instance().get_nodeaddr(hdrip->saddr());
  int dst = hdrc->next_hop();
  int st;
  int ne,ma;
  int aaa,bbb;
  int cntr,qqqid;
  link_struct templink;
  nsaddr_t *nodeptr;
  nsaddr_t *nptr;
  nsaddr_t *mptr;
  nsaddr_t* ptr;
  nsaddr_t prev_hop;
  nsaddr_t nextbc,isrc;
  int flag,ii,found;
  int seq_match;
  char *walk;
  link_struct* joe;
  link_struct* currlink;
  Packet *pnew;
  Time now = Scheduler::instance().clock(); // get the time

  Scheduler & s = Scheduler::instance();  // Useful to send (Schedule) packets

  // if I get my own transmission (excluding ierp_replies)
  if (hdrip->saddr() == myaddr_ && hdrz->ierpsource_ != myaddr_) {
    zdrop(p,DROP_RTR_HIYA);
    return;
  }

  //assert we're talking from network layer(rtagent) to network layer(rtagent)
  assert(hdrip->sport() == RT_PORT);   assert(hdrip->dport() == RT_PORT);
  assert(hdrc->ptype() == PT_ZRP);

  switch (hdrz->zrptype_)
    {

    case NDP_BEACON: // getting a beacon, send ack

      // re-use packet to create beacon ack, no need to drop
      hdrz->zrptype_ = NDP_BEACON_ACK;
      hdrc->next_hop_ = hdrz->originator_; // send directly to originator
      hdrc->direction() = hdr_cmn::DOWN; // this is default, but just in case
      hdrip->ttl() = 1;
      hdrip->saddr() = myaddr_;

      pkt_send(p,hdrz->originator_);  // send ack

      printf("\n_%2d_ [%6.6f] | Node %d received a beacon from Node %d "
```

```
        "(seq no. %d) | Node %d sent ack to Node %d (seq no. %d). ",
        myaddr_, (float)now, myaddr_,hdrz->originator_ , hdrz->seq_ ,
        myaddr_, hdrz->originator_, hdrz->seq_);

  print_tables();

  printf("\n");
  break;

case NDP_BEACON_ACK: // getting an ack, update neighbor table and
  // send update, and update link state table.

  printf("\n_%2d_ [%6.6f] "
        "| Node %d received an ack (seq no. %d) from Node %d. ",
        myaddr_, now, myaddr_, hdrz->seq_, hdrip->saddr());

  st = FALSE; // assume no change in N table

  // check if I am getting an ack for a beacon I sent
  if (hdrz->originator_ == myaddr_) {

    printf("");

    // add, or update if already in table
    st = neighbor_add(hdrip->saddr() );

  } else {
    zdrop(p,DROP_RTR_IARP);
    printf("| Node %d dropped an ack (seq no. %d) originating from a"
          " different Node %d", myaddr_, hdrz->seq_, hdrz->originator_ );
    print_tables();
    printf("\n");
    return;
  }
  aaa = st;
  bbb = hdrip->saddr();

  if (st) { // @@if (st)  change in NDP table, notify IARP

    // seq=0 means update from NDP
    st = lstable.update(hdrip->saddr(),myaddr_,TRUE,0,&seq_match);
    aaa= st;

    if (st) {
      // re-compute if there is a change in link state table
      //      rtable.compute_routes();
      rtable.compute_routes(myaddr_,radius_);
    }

    // always send update if neighbor has been added

    // send out immediate update (event-driven)
    pnew = pkt_create(IARP_UPDATE, IP_BROADCAST, radius_*2-1);

    hdr_zrp *hdrz_new = HDR_ZRP(pnew);

    pkt_create_link_data_space(pnew,MAX_LINKIE);
    pkt_add_link(pnew,myaddr_, hdrip->saddr(), TRUE);

    pkt_broadcast(pnew);

    printf("| Node %d sent update (seq no. %d) about new "
          "neighbor Node %d. ",myaddr_, hdrz_new->seq_,hdrip->saddr());

  }

  print_tables();
  printf("\n");

  zdrop(p,DROP_RTR_TTL);  // packet expired
  break;
```

```
case IARP_UPDATE:  // received update, forward if appropriate.

  char msg[100] ;

  sprintf(msg,"");

  if (hdrz->forwarded_ == 1 ) {
   sprintf(msg,"forwarded");
  }

  printf("\n_%2d_ [%6.6f] | Node %d received %s update (seq no. %d) "
         "sent from Node %d originating from Node %d (contains: ",
         myaddr_, (float)now, myaddr_, msg, hdrz->seq_, hdrip->saddr(),
         hdrz->originator_);

  for (int i=0; i<hdrz->numentries_; i++) {

   pkt_read_update(p, &templink, i);
   if (templink.isup_ == TRUE)
     printf("%d=%d ",templink.src_,templink.dest_ );
   else
     printf("%d#%d ",templink.src_,templink.dest_ );
  }

  printf("). ");

  flag = FALSE; // assume no updates to our table
  // go through list of updates in pkt
  for (int i=0; i<hdrz->numentries_; i++) {

   // load link[i] from update
   pkt_read_update(p, &templink, i);

   // update link state table
   st = lstable.update(templink.src_,templink.dest_,templink.isup_,
                      hdrz->seq_,&seq_match);

   // if change in link state table, recompute routes
   if (st) {
     flag = TRUE; // link table has been changed
     //      rtable.compute_routes();
     rtable.compute_routes(myaddr_,radius_);
   }

   if ( seq_match == TRUE) { // update was received before
     zdrop(p,DROP_RTR_IARP);
     printf("| Node %d dropped packet because seq id is same or older as "
            "id in table. ", myaddr_);
     // return;
     goto EOFF;
   }

  }

  // check ttl
  if ( (hdrip->ttl() < 1) ) {  // TTL expired, don't forward
   printf("| Node %d dropped packet due to TTL. ",myaddr_);
   zdrop(p,DROP_RTR_TTL);
   goto EOFF;
  }

  // let's forward by broadcast
  // re-use packet, no need to drop
  printf("| Node %d forwarded update (seq no. %d) "
         "sent from Node %d originating from Node %d (contains: ",
         myaddr_, hdrz->seq_, hdrip->saddr(), hdrz->originator_);
  for (int i=0; i<hdrz->numentries_; i++) {
   pkt_read_update(p, &templink, i);
   if (templink.isup_ == TRUE)
     printf("%d=%d ",templink.src_,templink.dest_ );
```

```
    else
      printf("%d#%d ",templink.src_,templink.dest_ );
  }

  printf("). ttl=%d",hdrip->ttl());
  hdrc->direction() = hdr_cmn::DOWN;
  hdrz->forwarded_ = 1;

  pkt_send(p,IP_BROADCAST);

EOFF:

  print_tables();

  printf("\n");

  // pgi - this was changed for the Haas demo in June! Do I need this?
  // quenching IARP updates
  // check if either lnksrc or lnkdest is me, don't forward, drop
  // this is my neighborhood.
  //      if ((hdrz->lnksrc_ == myaddr_) || (hdrz->lnkdest_ == myaddr_)){
  //drop(p,DROP_RTR_IARP);
  //return;
  //}
  return;

  break;

case IARP_DATA: // got a data packet, forward if I am relay,
  // or send up if I am destination.

  nodeptr = hdrz->route_;  // ptr to route list

  printf("\n_%2d_ [%6.6f] | Got IARP_Data packet orig=%d dest=%d ",myaddr_,
         now, hdrz->originator_,nodeptr[hdrz->numentries_ - 1]);

  printf("(");
  for (ii=0; ii<MAX_IERP_CACHE ; ii++)
    if (ierpcache[ii] != NULL)
      printf("%d.%d.%d ",ii,HDR_ZRP(ierpcache[ii]->pkt_)->enc_daddr_,
           HDR_ZRP(ierpcache[ii]->pkt_)->enc_daddr_ );

  // if pkt has arrived at dest (me)
  if (nodeptr[hdrz->numentries_ - 1] == myaddr_) {

    printf(" >>> we are at destination "
           "(delay=%3.2f ms seq=%d daddr=%d dport=%d ptype=%d) ",
           1000* ((float)now - (float)hdrz->pktsent_), hdrz->seq_,
           hdrz->enc_daddr_, hdrz->enc_dport_, hdrz->enc_ptype_);

    // copy ecnapsulated data back to this packet
    hdrip->daddr() = hdrz->enc_daddr_;
    hdrip->dport() = hdrz->enc_dport_;
    hdrc->ptype() = hdrz->enc_ptype_;
    hdrc->next_hop() = myaddr_;
    hdrc->addr_type_ = NS_AF_NONE;
    hdrc->size() = IP_HDR_LEN ;  //  set default packet size
    hdrip->ttl() = 1;
    hdrip->daddr() = myaddr_;
    hdrc->direction() = hdr_cmn::UP; // and is sent up

    // unicast
    pkt_send(p,myaddr_);

  } else {

    // forward onto next hop on list
    int nexthop = find_next_hop(p);
    if (nexthop == myaddr_) {
      // no next hop error
    }
```

```
    printf(" >>> forwarding to next hop %d ",nexthop);
    hdrc->next_hop() = nexthop;

    hdrc->direction() = hdr_cmn::DOWN;
    send_next_hop(p);
   }

   print_tables();
   printf("\n");

   break;

case IERP_REPLY: // got a reply, process as a relay or destination.

   ne = hdrz->numentries_;
   ma = myaddr_;

   printf("\n_%2d_ [%6.6f] | Got an ierp_reply, ",
         myaddr_,now,ne,hdrz->numentries_,ma,myaddr_);

   // if I am source, wake up IARP
   if (hdrz->ierpsource_ == ma) {
    printf("I am originator. \n");
     // wake up IARP, pass packet to it
     ierp_notify_iarp(p);
    zdrop(p,DROP_RTR_IERP); // finished with reply packet
     return;

   } else { // if I am not source
    // I am a relay node
    printf("forwarding route ");
     ptr = hdrz->route_;
    printf("[%d ",ptr[0] );
    // find prev hop, forward reply to it
    for (int i = 1; i<ne; i++) {
      printf("%d ",ptr[i] );
      if (ptr[i] == ma) {
        prev_hop = ptr[i-1]; printf("<");

      }
    }
    printf("]");

    printf(" to %d. \n",find_prev_hop(p));

    hdrc->direction() = hdr_cmn::DOWN; // send back down to prev node
    hdrip->ttl() = 3;
    hdrc->next_hop() = find_prev_hop(p);
    send_prev_hop(p);
   }
   break;

case IERP_REQUEST: // outgoing request. QD,ET as a relay or bordercaster.
  printf("\n_%2d_ [%6.6f] | Got an ierp_request, ",myaddr_,now);
  printf("origin of request is %d, queryid is %d. ",
         hdrz->ierpsource_,hdrz->qid_);

  printf("Last bordercaster was %d. Last hop was %d.",
         hdrz->lastbordercaster_,find_prev_hop(p) );
  ne = hdrz->numentries_;
  nextbc = hdrz->route_[ne-1];
  qqqid = hdrz->qid_;
  isrc = hdrz->ierpsource_;

  ptr = hdrz->route_;

  // if dest is in my zone, originate reply and send back route
  if (rtable.route_exists(hdrz->ierpdest_)) {

    printf("\n\t| ierpdest (%d) is in my zone, sending IERP Reply "
           "with accumulated route", hdrz->ierpdest_);
```

```
  // where am I on route list? store in cntr
  cntr=999;
  for (int i=0; i<hdrz->numentries_; i++) {
    if (ptr[i] == myaddr_ ) {
      cntr=i+1;
    }
  }
  assert(cntr != 999); // I have to be on list!
  // clip route after me
  //memcp data to tmp
  bcopy(hdrz->route_,tmp,cntr*sizeof(nsaddr_t));
  // allocate a possibly smaller space, enough for accum route up to me
  pkt_create_route_data_space(p,100); //cntr+1);
  //memcp tmp back to data
  bcopy(tmp,hdrz->route_,cntr*sizeof(nsaddr_t));
  hdrz->numentries_ = cntr;
  add_local_route(p,hdrz->ierpdest_);

  //reuse packet, change to ierp reply
  hdrz->zrptype_ = IERP_REPLY;
  hdrc->direction() = hdr_cmn::DOWN; // send back down protocol stack
  send_prev_hop(p);
  printf("\n");
  return;
}

if (ptr[hdrz->numentries_-1] == myaddr_) { // if last entry is me
  // I am a periph node
  printf("\n\t| I am a peripheral node. My peripherals are [");
  for (rtable_node *mptr = rtable.periphnodes_; mptr != NULL;
       mptr=mptr->next_) {
    printf("%d ",mptr->id_);
  }
  printf("].");

  // QD - cover last bordercaster's zone
  found = query_detect1(hdrz->qid_, hdrz->ierpsource_,
                        hdrz->lastbordercaster_, radius_);

  // unicast to each peripheral node
  for (rtable_node *mptr = rtable.periphnodes_; mptr != NULL;
       mptr=mptr->next_) {

    // check if node is covered, otherwise bordercast
    if (node_is_covered(mptr->id_, hdrz->qid_, hdrz->ierpsource_)) {
      // Early Termination
      printf("\n\t| ET - Query detected peripheral node %d is in zone"
             " of previous bordercaster %d or is just on covered list",
             mptr->id_, hdrz->lastbordercaster_);
    } else {

      Packet *pnext = pkt_create(IERP_REQUEST, IP_BROADCAST, 1);
      // parameters don't matter, will be overwritten by pkt_copy
      pkt_copy(p,pnext);
      // add myself as first route entry

      add_local_route(pnext,mptr->id_); // add rest

      hdr_cmn *hdrcnext = HDR_CMN(pnext);
      hdr_zrp *hdrznext = HDR_ZRP(pnext);
      hdrcnext->direction() = hdr_cmn::DOWN; // send back down
      hdrznext->lastbordercaster_ = myaddr_;
      printf("\n\t| Sending ierp request to next bordercaster %d, "
             "next hop is %d. ", mptr->id_, find_next_hop(pnext));
      printf("Route list [");
      for (int i=0; i<hdrznext->numentries_; i++)
        printf("%d ",hdrznext->route_[i]);
      printf("] ");
      send_next_hop(pnext);
```

```
      }
    } // for mptr
    zdrop(p,DROP_RTR_IERP);

    // QD - cover my zone
    query_detect1(qqqid,isrc,myaddr_, radius_);

  } else { // I am a "relay"
    printf("| I am a relay node. ");
    printf("Route list [");
    for (int i=0; i<ne; i++) printf("%d ",hdrz->route_[i]);
    printf("]");

    hdr_zrp *hdrz = HDR_ZRP(p);

    // send if next bordercaster is not covered
    if (node_is_covered(nextbc, hdrz->qid_, hdrz->ierpsource_)){
      printf("\n\t| ET - Query detected next bordercaster %d is in "
             "covered list. ", nextbc);
      zdrop(p,DROP_RTR_IERP);
    } else {
      hdrc->direction() = hdr_cmn::DOWN;
      printf("| Sending to next hop %d, next bordercaster is %d.",
             find_next_hop(p), nextbc);
      send_next_hop(p);
    }

    // QD - mark last bordercaster's zone's inner nodes as covered
    found = query_detect1(hdrz->qid_, hdrz->ierpsource_,
                          hdrz->lastbordercaster_, radius_-1);

    // add next bordercaster to covered list
    // check node against list in query entry
    flag = FALSE;
    for (rtable_node *ptr=queries_[found]->next_;
         ptr!=NULL; ptr=ptr->next_) {
      // if node in query entry  == next bordercaster
      if (ptr->id_ == nextbc) {
        flag=TRUE; //=found, don't need to add again
        break;
      }
    }
    if (!flag) { // if not in list, add at head
      rtable_node *tmp;
      tmp = new rtable_node;
      tmp->id_ = nextbc;
      tmp->next_ = queries_[found]->next_;
      queries_[found]->next_ = tmp;
      printf("\n\t| QD - Next bordercaster [%d] added to covered list.",
             (int)tmp->id_ );
    } else {
      printf("\n\t| QD - Next bordercaster [%d] already added to covered "
             "list.", nextbc);
    }
  }


  printf("\n\t| Covered list for query=%d origin=%d [",
         qqqid,isrc);
  for (rtable_node *p=queries_[found]->next_; p!=NULL; p=p->next_) {
    printf("%d ",p->id_);
  }

  printf("]\n");

  break;

default:
  // error, unknown zrptype, drop
  // debug - printf("Uknown ZRPTYPE !\n");
  fprintf(stderr, "Invalid ZRP type (%x)\n", hdrz->zrptype_);
```

```
      exit(1);
    }
}
//=========================================================================
// IARP:
// Called by IARP to update list of outer routes for any changes to
// local routes.
//=========================================================================
void ZRPAgent::IARP_update_outer_route(int route_index) {
  rtable_node *nptr;
  rtable_node *mptr;
  rtable_node *newnode;
  rtable_node *newhead;
  rtable_node *deleteme;
  int index;

  // if entry is empty or route list is empty (latter is unlikely)
  if (rtable.outer_route[route_index] == NULL ||
      rtable.outer_route[route_index]->next_ == NULL)
    return;

  // finds all the nodes in outer route that exist in current node's zone
  // and returns pointer to the furthest in hops from current node (should be
  // a peripheral), and guarantees it is the last inner node before end of outer route list
  for ( nptr=rtable.outer_route[route_index]->next_; nptr != NULL;
        nptr=nptr->next_) {
    index = rtable.node_exists(nptr->id_ );

    if (index != -1 ) {
      mptr = nptr; // found peripheral
    }

  } // end for

  if (index == -1)
    return; // nothing to do, in future return error so we can delete route
  // outer route must contain an inner route as a subset!

  // mark where we are going to start deleting on old clipped list
  deleteme = rtable.outer_route[route_index]->next_;

  // start creating new local route and pre-pending to where mptr left off
  // list looks like this:
  // rtable.outer_route[route_index]->next_->O -> -> -> mptr-> last
  // inner node-> -> -> dest ->NULL
  newhead = new rtable_node;
  newhead->id_ = rtable.route[index]->id_;
  newhead->next_ = mptr->next_;

  //for each route entry list node, copy to newnode and insert newnode
  // at head of newhead
  for (nptr=rtable.route[index]->next_; nptr != NULL; nptr=nptr->next_) {
    newnode = new rtable_node;
    newnode->id_ = nptr->id_;
    newnode->next_ = newhead;
    newhead = newnode;
  }

  // attach newhead to outer route entry
  rtable.outer_route[route_index]->next_ = newhead;

  //delete old inner route
  // cut at last inner route node
  mptr->next_ = NULL;
  for (nptr=deleteme; nptr != NULL; ) {
    nptr=nptr->next_;
    delete deleteme;
    deleteme = nptr;
  }
}
```

```
//==================================================================
// IERP:
// Call by IERP to start a request from source node.
// Protocols: IERP, get local route from IARP
//==================================================================
void ZRPAgent::originate_request(nsaddr_t final, int qid) {
  rtable_node* ptr;
  nsaddr_t *nptr;
  int ind;
  Scheduler & s = Scheduler::instance();  // Useful to send (Schedule) packets
  Packet* p;

  printf("\n_%2d_ | Bordercast to ", myaddr_ );
  // unicast to each peripheral node
  for (ptr = rtable.periphnodes_; ptr != NULL; ptr=ptr->next_) {
    printf("[%d - route ",ptr->id_);
    p = pkt_create(IERP_REQUEST, IP_BROADCAST, MAX_ROUTE_LENGTH*2);
    hdr_zrp *hdrz = HDR_ZRP(p);
    hdr_cmn *hdrc = HDR_CMN(p);
    hdr_ip *hdrip = HDR_IP(p);

    hdrz->lastbordercaster_ = myaddr_;
    hdrz->ierpsource_ = myaddr_;
    hdrz->ierpdest_ = final;
    hdrz->qid_ = qid;
    pkt_create_route_data_space(p,1);

    // add myself in source route list
    pkt_add_node(p, myaddr_, 0);
    hdrz->numentries_ = 1;
    hdrz->routeindex_ = 0;

    // add the rest of the route to the peripheral node
    add_local_route(p,ptr->id_);

    nptr = hdrz->route_;
    for (int i=0; i<hdrz->numentries_ ; i++) {
      printf("%d ",nptr[i]);
    }
    printf("(nexthop=%d) ]",find_next_hop(p) );
    hdrc->direction() = hdr_cmn::DOWN;

    // since we are in request mode, we move route index forward
    hdrz->routeindex_++;
    ind=hdrz->routeindex_;
    // then send it
    pkt_send(p,hdrz->route_[ind]);
  }

  // pseudo-code:
  // for each periph node
  // create packet
  // add to hdrzrp ORIGINATOR, REQUESTED, NEXTPERIPH, LASTPERIPH
  // add src route to that node, attach to pkt
  // saddr() = me
  // daddr() = periph node
  // nexthop = next on src lst
  // send
}

//==================================================================
// IERP:
// Route Packet, first check if there is a local route. If so, add
// route to packet and send.
// Otherwise check if there is an outer route. If so, add and send.
// Otherwise, originat request.
// Protocols: IERP, IERP calls IARP
//==================================================================
void ZRPAgent::route_pkt(Packet* p, nsaddr_t daddr) {
  hdr_cmn *hdrc = HDR_CMN(p);
  hdr_ip *hdrip = HDR_IP(p);
```

```
hdr_zrp *hdrz = HDR_ZRP(p);
nsaddr_t *nexthop;
int st;
int ierpd;
Time now = Scheduler::instance().clock(); // get the time

hdrz->routeindex_=0;
hdrz->numentries_ =0;

st = add_local_route(p,daddr);
if (!st) {
  st = add_outer_route(p,daddr); //printf("the route saved is now:");
  nsaddr_t *rrr = hdrz->route_;
}

if (st) { // if a local or outer route exists, it was added, start sending...
  //send to first addr on list
  //nexthop = (nsaddr_t *)p->accessdata();
  nexthop = hdrz->route_;

  // this packet will now be encapsulated into an IARP_DATA packet

  // save upper layer info in encapsulated area
  hdrz->enc_daddr_ = hdrip->daddr() ;
  hdrz->enc_dport_ = hdrip->dport() ;
  hdrz->enc_ptype_ =  hdrc->ptype();

  hdrc->direction() = hdr_cmn::DOWN; // this is default, but just in case
  hdrc->ptype() = PT_ZRP;
  hdrc->next_hop() = nexthop[0];
  hdrc->addr_type_ = NS_AF_NONE;
  hdrc->size() = IP_HDR_LEN ;  //  set default packet size
  hdrip->ttl() = IERP_TTL;
  hdrip->daddr() = nexthop[0] ;
  hdrip->dport() = ROUTER_PORT;
  hdrz->zrptype_ = IARP_DATA;
  hdrz->originator_ = myaddr_;
  hdrz->pktsent_ = Scheduler::instance().clock(); // get the time
  hdrz->seq_ = seq_ ;
  inc_seq();
  send_next_hop(p);
  return;
}

// no local or cached route, call IERP
// push onto ierp request queue
int flag = FALSE;
ierpd = hdrz->ierpdest_;

for (int i=0; i<MAX_IERP_CACHE; i++) {
  if (ierpcache[i] == NULL) {
    flag = TRUE;
    st = i;
  } else if (HDR_ZRP(ierpcache[i]->pkt_)->ierpdest_ == ierpd) {
    // ierp request pending
    // drop packet
    printf("\n_%2d_ [%6.6f] | A request for destination already in "
           "ierpcache, dropping packet\n",myaddr_,now);
    zdrop(p,DROP_RTR_IERP);
    // or add buffer data structure
    return;

  }

}
assert(flag == TRUE); // increase MAX_IERP_CACHE otherwise
if (flag == FALSE) { // don't like assertion? comment out and keep this
  zdrop(p,DROP_RTR_HIYA);
  return;
}
```

```
  // new entry for ierp request cache
  ierpcache[st] = new IERPCacheEntry;
  ierpcache[st]->qid_ = qid_;
  ierpcache[st]->expiry_ = Scheduler::instance().clock() +
    IERP_REQUEST_TIMEOUT;
  ierpcache[st]->pkt_ = p;
  originate_request(daddr, qid_ ) ;
  qid_++;
  printf("\n\t| New ierpcache entry: qid=%d expiry=%3.3f (daddr=%d)\n",
         ierpcache[st]->qid_,
         ierpcache[st]->expiry_,daddr);

}

//==================================================================
// IARP:
// Add route to packet if local route to daddr exists. Return true if so.
// Otherwise return false.
//==================================================================
int ZRPAgent::add_local_route(Packet* p, nsaddr_t daddr) {
  rtable_node *nptr;
  int totalhops; int cntr;
  nsaddr_t *ptr;
  hdr_zrp *hdrz = HDR_ZRP(p);
  int tmp[MAX_ROUTE_LENGTH];

  // assuming there could be node entries (hdrz->numentries_)
  // already in pkt
  // should work even if hdrz->numentries_ = 0

  for (int i=0; i< MAX_NODES_PER_ZONE; i++) {

    // if not found return FALSE
    if (rtable.route[i] == NULL) {
      return FALSE; // end of table, not found
    } else { // still looking
      if (rtable.route[i]->id_ == daddr) { // if found
        totalhops = hdrz->numentries_ + rtable.route[i]->numhops_ ;

        //memcp data to tmp
        bcopy(hdrz->route_,tmp,hdrz->numentries_*sizeof(nsaddr_t));

        // allocate a BIGGER space, enough for prev entries + new entries
        pkt_create_route_data_space(p,totalhops);

        //memcp tmp back to data
        bcopy(tmp,hdrz->route_,hdrz->numentries_*sizeof(nsaddr_t));
        //
        cntr = totalhops-1;

        // copy route list to data portion of packet (reverse order)
        for (nptr = rtable.route[i]; nptr->id_ != myaddr_;  nptr=nptr->next_) {
          pkt_add_node(p, nptr->id_, cntr ) ;
          cntr--;
        } // end for nptr

        hdrz->numentries_ = totalhops;
        return TRUE;
      } // end if (rtable.route[i]->id_ == daddr)

    }  //end if (rtable.route[i] == NULL)

  // find daddr in lookup
  // else
  // create data area of size (num hops is stored in table)
  // step through table, adding each node to data area
  //
  } // end for i
}

//============================================================
```

```
// NDP:
// Add neigbor with address "addr" to NDP table if it is not
// already there. If already there, return false. Otherwise true.
//==============================================================
int ZRPAgent::neighbor_add(nsaddr_t neighbor_addr) {
  neighbor_struct *ns;
  Time now = Scheduler::instance().clock(); // get the time

  if (neighbors_ == NULL) {

    num_neighbors_ ++;

    neighbors_ = new neighbor_struct;
    neighbors_->addr_ = neighbor_addr;
    neighbors_->expiry_ = now+neighbor_timeout_;
    neighbors_->lastack_ = now;
    neighbors_->next_ = NULL;

    printf("| Node %d added new neighbor Node %d, will expire at %6.6f sec, "
         "lastack timer will expire at %6.6f sec.",
           myaddr_, neighbor_addr , neighbors_->expiry_,
           neighbors_->lastack_ + neighbor_ack_timeout_);
        return TRUE;
  }

  for (ns=neighbors_; ns != NULL; ns=ns->next_ ){
    if (ns->addr_ == neighbor_addr){
      // update neighbor timeout

      printf("| Node %d updated neighbor table, changed neighbor Node %d "
             "expiry from %6.6f to %6.6f sec and lastack timer expiry "
             "from %6.6f to %6.6f. ",
             myaddr_, neighbor_addr,
             (float)neighbors_->expiry_, (float)now + neighbor_timeout_,
             ns->lastack_, now + neighbor_ack_timeout_ );

      ns->expiry_ = now+neighbor_timeout_;
      ns->lastack_ = now;

      return FALSE; // no change, we have that node in our table
    }
  }

  // if we end up here, we have at least one neighbor on list but no match
  // so add our new neighbor to top of list
  ns = new neighbor_struct;
  ns->addr_ = neighbor_addr;
  ns->expiry_ = now+neighbor_timeout_;
  ns->lastack_ = now;
  ns->next_ = neighbors_;
  neighbors_ = ns;
  num_neighbors_ ++;
  printf("| Node %d added new neighbor Node %d, entry will expire at "
        "%6.6f sec, lastack timer will expire at %6.6f sec.",
        myaddr_,neighbor_addr, neighbors_->expiry_,
        neighbors_->lastack_ + neighbor_ack_timeout_);

  return TRUE;

}

//======================================================================
// BRP:
// Mark "node" as covered for given query (qid,origin).
//======================================================================
int ZRPAgent::node_is_covered(nsaddr_t node, int queryid, nsaddr_t originid) {
  int flag = FALSE;
  rtable_node *ptr;
  Time now = Scheduler::instance().clock(); // get the time
  int found = 999;
```

```
  for (int i=0; i<MAX_QUERY_DETECT_TABLE_SIZE; i++) {
    if (!flag && queries_[i] != NULL && queries_[i]->originid_ == originid &&
        queries_[i]->qid_ == queryid)  {
      //  refresh expiry unconditionally
      queries_[i]->expiry_ = now +  QUERY_TABLE_EXPIRATION;
      found = i;
      for (rtable_node *ptr=queries_[i]->next_; ptr!=NULL; ptr=ptr->next_) {
        if (ptr->id_ == node) {
          flag=TRUE;
          break;
        }
      }
      break;
    }
  }

  // return FALSE if we do not find query entry at all
  // return TRUE only if we find query entry AND the node is in its list
  // all else is FALSE!!
  return(flag);
}

//===================================================================
// BRP:
// Query detect given (qid,origin). Will add nodes of covered area (nodes
// that are "radius" hops away from "lastbordercaster") to a cached
// query detect entry. If this query has not been received before, will
// create a new entry and add covered nodes to entry.
// Returns TRUE if query was received before, else FALSE.
//===================================================================
int ZRPAgent::query_detect1(int queryid, nsaddr_t originid,
                            nsaddr_t lastbordercaster, int radius) {
  RoutingTable prevtable(this);
  int flag = FALSE;
  int found;
  Time now = Scheduler::instance().clock(); // get the time
  int iii;

  // 1. if query exists in our cache, =queries[found]
  // 2. if not exist, find an empty slot, queries[found],
  //    add new entry at queries[found]
  // 3. create route table for prev bordercaster
  // 4. add nodes in table to queries[found], ie
  //    for each node in table:
  //       if node_is_covered is false (node is NOT in queries),
  //          add to queries[found]
  // 5.  look for query id in queries_[]

  // 1. if exist in queries_[], queries[found] is the one
  for (int i=0; i<MAX_QUERY_DETECT_TABLE_SIZE; i++) {
    if (!flag && queries_[i] != NULL && queries_[i]->originid_ == originid &&
        queries_[i]->qid_ == queryid)  {
      flag = TRUE;
      found = i;

      // refresh expiration
      if (now > queries_[i]->expiry_) {
        queries_[i]->expiry_ = now +  QUERY_TABLE_EXPIRATION;
      }
      break;
    }
  }

  // 2. if not exist, find an empty slot, queries[found],
  //    add new entry at queries[found]
  if (!flag) { // if no entry already, create one
    for (int i=0; i<MAX_QUERY_DETECT_TABLE_SIZE; i++) {
      if (!flag && queries_[i] == NULL) {
        flag = TRUE;
        found = i;
        queries_[i] = new QueryDetectEntry;
```

```
        queries_[i]->originid_ = originid;
        queries_[i]->qid_ = queryid;
        queries_[i]->expiry_ = now +QUERY_TABLE_EXPIRATION;
       queries_[i]->next_ = NULL;
       break;
      }
    }
  }

  // 3. create+compute route table for prev bordercaster
  prevtable.my_address_ = lastbordercaster; //  get route table for last bc
  prevtable.linkstatetable = &lstable;

  for (int i=0; (i< MAX_NODES_PER_ZONE); i++) {
    prevtable.route[i] = NULL;
  }
  prevtable.periphnodes_=NULL;
  prevtable.agent_ = this;
  prevtable.compute_routes(lastbordercaster, radius);

  // 4. add nodes in table to queries[found], ie
  //    for each node in table:
  //       if node_is_covered is false (node is NOT in queries),
  //          add to queries[found]
  printf("\n\t| QD - Calculated Route Table of bordercaster %d: ",
         lastbordercaster);
  for (int i=1; prevtable.route[i] != NULL ; i++) {
    printf("[");
    for (rtable_node *j=prevtable.route[i]; j != NULL; j=j->next_  ) {
      printf("%d ",j->id_);
    }
    printf("] ");
  }

  printf("\n\t| QD - added [");
  rtable_node *tmp;
  for (iii=0; prevtable.route[iii] != NULL &&
         prevtable.route[iii]->numhops_<=radius ; iii++) {
    flag = FALSE; // =not found

    // check node against list in query entry
    for (rtable_node *ptr=queries_[found]->next_; (!flag) && (ptr != NULL);
         ptr=ptr->next_) {
      if (ptr->id_ == prevtable.route[iii]->id_) {
       flag=TRUE; //=found
       printf("");
      }
    }

    // debug only
    if (flag) { // just to illustrate that this flag tells us if
      //the query was in our cache or not
      //printf("FOUND");
    } else {
      //printf("NOTFOUND");
    }
    if (!flag) { // if not in list, add at head
      // want to add to a separate list, then join afterwards. more efficient
      tmp = new rtable_node;
      tmp->id_ = prevtable.route[iii]->id_;
      tmp->next_ = queries_[found]->next_;
      queries_[found]->next_ = tmp;
      printf("%d ", queries_[found]->next_->id_ );
    }
  }
  printf("] to covered list. ");
  return(found) ;
}
```
// End of ZRP.CC===

// ZRP.H===

```
#ifndef _zrp_h_
#define _zrp_h_

#include <config.h>
#include <assert.h>
#include <agent.h>
#include <packet.h>
#include <ip.h>
#include <delay.h>
#include <scheduler.h>
#include <queue.h>
#include <trace.h>
#include <arp.h>
#include <ll.h>
#include <mac.h>
#include <priqueue.h>
#include <delay.h>
#include "zrp_table.h"

#if defined(WIN32) && !defined(snprintf)
#define snprintf _snprintf
#endif /* WIN32 && !snprintf */

#define ROUTER_PORT      0xff
#define FOUND 1
#define LOST 0
#define TRUE 1
#define FALSE 0
#define NULL 0

#define MAX_OUTBOUND_PENDING 20 // max queue for pkts waiting for ierp requests
#define DEFAULT_ZONE_RADIUS 3

#define MAX_DETECTED_QUERIES 20 // max queries that can be detected
#define IERP_REQUEST_TIMEOUT 30 // secs before timeout on an ierp request
#define ZRP_STARTUP_JITTER 2.0       // secs to jitter start of periodic activity
#define MAX_SEQUENCE_ID 1000000000
#define QUERY_TABLE_EXPIRATION 1000
#define MAX_QUERY_DETECT_TABLE_SIZE 50

#define NEIGHBOR_ACK_TIMEOUT 10 // sec
#define TIMER_JITTER 3

#define BEACON_PERIOD 1  // period of beacon transmission in sec
#define NDP_SCAN_PERIOD 5
        // scan NDP table every number of beacon timeout cycles
#define MAX_NODE_ID_CHAR_SIZE 10 // longest name string for my node address
                             // (char *)myid_ is string equiv of (int)myaddr_
#define MAX_IERP_CACHE    50  // max ierp requests cache size
#define MAX_QID           50  // max size query detect table
#define FOUND 1
#define LOST 0
#define TRUE 1
#define FALSE 0

typedef double Time;
typedef int32_t Query_ID;

class ZRPAgent;

enum ZRPTYPE // Types of ZRP packets
{
  NDP_BEACON, NDP_BEACON_ACK, IARP_UPDATE, IARP_DATA, IERP_REPLY, IERP_REQUEST
};

// link state data used in updates
struct linkie {

  int src_;   // 32 bits
```

```
  int dest_;  // 32 bits
  int isup_;
};

// ZRP header structure
struct hdr_zrp {
  int zrptype_;
  //  int safe_;
  int routeindex_;

  // IARP hdr
  int seq_;
  int numentries_; // number of link entries in this update
  // or length of direct routing list
  // info is in data portion of packet, use accessdata()

  nsaddr_t originator_; // for ndp, iarp or ierp
  nsaddr_t ierpsource_;
  nsaddr_t ierpdest_;
  nsaddr_t iarpsource_;
  nsaddr_t iarpdest_;
  nsaddr_t lastbordercaster_;
  Time pktsent_;

  // this is where the original data for upper layer pkts
  // is saved while ZRP routes pkt, at dest this is placed
  // back into hdrip->dport(), ie this is part of encapsulated data
  int enc_dport_;
  int enc_daddr_;
  packet_t enc_ptype_;

  int qid_;  // query id counter

  nsaddr_t *route_; // pointer to route list data
  linkie *links_;  // pointer to link state list

  int forwarded_;  // TRUE if forwarded before

  // Packet header access functions
  static int offset_;
  inline static int& offset() { return offset_; }
  inline static hdr_zrp* access(const Packet* p) {
    return (hdr_zrp*) p->access(offset_);
  }

};

class ZRPBeaconTransmitTimer : public Handler {
 public:

  ZRPBeaconTransmitTimer(ZRPAgent* a)  { agent = a; }
  // pgi- set zrp_agent to 'a', ie our ZRP_Agent
  void handle(Event*);  // function handling the event
  void start(double thistime);

  //  private:
  ZRPAgent *agent;
  Event intr;

};

class ZRPNeighborScanTimer : public Handler {
 public:

  ZRPNeighborScanTimer(ZRPAgent* a)  { agent = a; }
  // pgi- set zrp_agent to 'a', ie our ZRP_Agent
  void handle(Event*);  // function handling the event
  void start(double thistime);

  //  private:
  ZRPAgent *agent;
```

```
  Event intr;

};

class ZRPPeriodicUpdateTimer : public Handler {
 public:

  ZRPPeriodicUpdateTimer(ZRPAgent* a)  { agent = a; }
  // pgi- set zrp_agent to 'a', ie our ZRP_Agent
  void handle(Event*);  // function handling the event
  void start(double thistime);

  //  private:
  ZRPAgent *agent;
  Event intr;

};

class ZRPAckTimer : public Handler {
 public:

  ZRPAckTimer(ZRPAgent* a)  { agent = a; }
  // pgi- set zrp_agent to 'a', ie our ZRP_Agent
  void handle(Event*);  // function handling the event
  void start();

  //  private:
  ZRPAgent *agent;
  Event intr;

};


class ZRPAgent : public Agent {

  friend class ZRPBeaconTimer;
 public:
  ZRPAgent(); // not generally used
  ZRPAgent(nsaddr_t id);

  void recv(Packet * p, Handler *);
  int  initialized() { return 1 && target_; }

  //== Tcl Related ======================================
  char*  myid_;   // (char *)myid_ is string equiv of (int)myaddr_
  PriQueue* ll_queue;
  Trace* tracetarget;
  MobileNode* node_;
  NsObject* port_dmux_;

  //==Timer Related ==========================================

  ZRPNeighborScanTimer NeighborScanTimer_;
  ZRPBeaconTransmitTimer BeaconTransmitTimer_;
  ZRPPeriodicUpdateTimer PeriodicUpdateTimer_;
  ZRPAckTimer AckTimer_;

  //==Methods ================================================

  void startup();
  int  command (int argc, const char*const* argv);
  void scan_ierp_cache();
  void ierp_notify_iarp(Packet *p);
  void recvZRP(Packet* p);
  void forward_update(Packet* p);
  void route_pkt(Packet* p, nsaddr_t dest);
  void print_tables();

  int  neighbor_add(nsaddr_t newneighbor);
  void send_periodic_update();
  void sendp(nsaddr_t ns);
```

```
  void originate_request(nsaddr_t final, int qid);
  void scan_query_detections();
  void scan_linkstate_table();
  int query_detect1(int qid, nsaddr_t originid, nsaddr_t lastbordercaster,
                   int radius);
  int  node_is_covered(int queryid, nsaddr_t node);
  void inc_seq() { seq_++; if (seq_ > MAX_SEQUENCE_ID) seq_ = 1;}
  void dec_seq() { seq_--; if (seq_ < 1)  seq_ = MAX_SEQUENCE_ID;}
  void pkt_add_all_links(Packet *p);
  void pkt_send(Packet *p, nsaddr_t addressee);
  void send_next_hop(Packet *p);
  void send_prev_hop(Packet *p);
  void pkt_broadcast(Packet *p);
  Packet* pkt_create(ZRPTYPE zrp_type, nsaddr_t addressee, int ttl);
  void pkt_add_node(Packet* p, nsaddr_t node, int total);
  void pkt_add_link(Packet* p, nsaddr_t lnksrc, nsaddr_t lnkdest, int isup);
  void pkt_read_update(Packet* p, link_struct *ls, int index) ;
  nsaddr_t find_next_hop(Packet* p);
  nsaddr_t find_prev_hop(Packet* p);
  int add_local_route(Packet* p, nsaddr_t node);
  int add_outer_route(Packet* p, nsaddr_t node);
  void do_update();
  void print_routes();
  void pkt_create_route_data_space(Packet *p, int size);
  void pkt_free_route_data_space(Packet *p);
  void pkt_create_link_data_space(Packet *p, int size);
  void pkt_free_link_data_space(Packet *p);
  void pkt_copy(Packet *pfrom, Packet *pto);
  void zdrop(Packet *p, const char *s);
  int  node_is_covered(nsaddr_t node, int queryid, nsaddr_t originid);
  void outer_route_expiration();
  void IARP_update_outer_route(int index);

  //==Data ===================================================

  int myaddr_;
  int radius_;   // Can be set from Tcl script, see ZRPAgent::command
  int transmit_jitter_;
  int startup_jitter_;
  int process_jitter_;
  int beacon_period_; // Can be set from Tcl script, see ZRPAgent::command
  int beacon_period_jitter_;
  int neighbor_timeout_;
  int neighbor_ack_timeout_;
  int iarp_update_period_;
  int neighbortable_scan_period_;

  int num_neighbors_;

  int tx_; // total pkts transmitted by agent
  int rx_; // total pkts received by agent
  int seq_; // current sequence id for outgoing IARP updates
  int qid_; // global query id counter, updated for every query sent by node

  int suspend_flag_; // Can be set from Tcl script, see ZRPAgent::command
  // "suspends" node, there are Tcl commands that work better "start","stop"

  //==Tables ===================================================
  neighbor_struct *neighbors_; // linked list of my neighbors
  LinkStateTable lstable;     // class, linked list
  RoutingTable rtable;        // class, array of pointers
  IERPCacheEntry *ierpcache[MAX_IERP_CACHE]; // array of pointers
  QueryDetectEntry *queries_[MAX_QUERY_DETECT_TABLE_SIZE];
};

#endif
```

// End ofZRP.H==

// ZRP_TABLE.CC===

```
#include <config.h>
#include <agent.h>
#include <packet.h>
#include <ip.h>
#include <delay.h>
#include <scheduler.h>
#include <queue.h>
#include <trace.h>
#include <arp.h>
#include <ll.h>
#include <mac.h>
#include <priqueue.h>
#include <delay.h>
#include <random.h>
#include <object.h>
#include <route.h>
#include "zrp.h"
#include "zrp_table.h"

#define UP    1
#define DOWN  0

//=============================================================================
// Linkstate Table:
// Perform an update to table given a link state, Return seq_match=TRUE
// if there was a match. Return update=TRUE if any change in table.
//=============================================================================
BOOLEAN
LinkStateTable::update(int src, int dest, int lstate, int seq, int *seq_match)
{
  Time now = Scheduler::instance().clock(); // get the time
  int flag; // TRUE indicates there has been a change in table
  link_struct* i;
  int found;

  *seq_match = FALSE;
  found = FALSE;
  flag = FALSE;

  if  (lshead_ == NULL) { // check if empty
    if (lstate == UP) { // add only if up
      // add to table,
      lshead_ = new link_struct;
      lshead_->src_ = src;
      lshead_->dest_ = dest;
      lshead_->isup_ = lstate;
      lshead_->seq_ = seq;
      lshead_->expiry_ = now + LINKSTATE_EXPIRATION;
      lshead_->next_ = NULL;
      return TRUE;  // change in table
    }
    return FALSE;
  }

  for (i=lshead_; i != NULL; i=i->next_) {
    if ( ( (i->src_ == src)  && (i->dest_ == dest) ) || ( (i->src_ == dest) && (i->dest_ == src)  )
) {  // src/dest match with update

      if ( (seq != 0) && (i->seq_ >= seq) ) {  // got this update before
        *seq_match = TRUE;
        return FALSE;
      }

      found = TRUE; // found a match (a,b) or (b,a)

      i->expiry_ = now + LINKSTATE_EXPIRATION; // update expiry

      if (seq != 0 )  // only update if new seq is non-zero (
        i->seq_ = seq; // update to new seq no.
```

```
      if ( i->isup_ != lstate) { //inequality=we need to change state to latest
        i->isup_ = lstate;
        flag = TRUE;
      }
    }
  }

  if (found) // return flag if there was change
    return flag;

  // we have found nothing up until this point, so add link

  if (lstate == UP) {  // don't want to add a downed link
    // add to table,
    i = new link_struct;
    i->src_ = src;
    i->dest_ = dest;
    i->isup_ = lstate;
    i->seq_ = seq;
    i->next_ = lshead_;
    i->expiry_ = now + LINKSTATE_EXPIRATION;
    lshead_ = i;
    return TRUE;  // change in table
  }
  return FALSE;
}

//=========================================================================
// Linkstate Table:
// Insert Link state entry into Link state table
//=========================================================================
void
LinkStateTable::add(link_struct* link)
{
  // insert link at head
  if (lshead_ != NULL) {  // if there is at least one link in list
    link->next_ = lshead_;       ;
    lshead_ = link;
  } else { // if empty
    lshead_ = link;
    lshead_->next_ = NULL;
  }

}

//==========================================================================
// Route Table:
// Erases route and peripheral tables.
//==========================================================================
void RoutingTable::erase() {
  // would be better if we saved fallow data structures
  // on a free list - for version 2.0 ;)
  int i;
  rtable_node *tmp;

  //erase route table
  for (i=0; (i< MAX_NODES_PER_ZONE) && (route[i] != NULL) ; i++) {
    delete(route[i]);
    route[i] = NULL;
  } //end for

  // erase periph node list
  while (periphnodes_ != NULL ) {
    tmp = periphnodes_;
    periphnodes_ = periphnodes_->next_;
    delete(tmp);
  }
  periphnodes_ = NULL;
}

//===========================================================================
```

```
// Linkstate Table:
// Clean table of expired links.
//============================================================================
void
LinkStateTable::purge_links()
{
  Time now = Scheduler::instance().clock(); // get the time
  link_struct *ls;

  // deletes down or expired links

  if (lshead_ == NULL)
    return;

  // hack, insert a dummy node at head
  link_struct dummy;
  dummy.next_ = lshead_;
  lshead_ = &dummy;

  for (ls=lshead_ ; ls->next_ != NULL; ) {

    if ( (ls->next_->expiry_ < now) || (ls->next_->isup_ == FALSE ) ) {
      link_struct *temp = ls->next_;
      ls->next_ = ls->next_->next_;
      delete temp;
      // other actions to take when you delete a node here
    } else {
      ls=ls->next_ ;
    }

  }
  lshead_ = dummy.next_;

}

//=============================================================================
// Linkstate Table:
// Print table to STDOUT.
//=============================================================================
void
LinkStateTable::print_links()
{
  link_struct *ls;

  if (lshead_ == NULL) {
    printf("LinkTable: empty ");
  } else {
    printf("LinkTable: ");

    for (ls = lshead_;
         (ls != NULL);
       ls=ls->next_) {   // If link list has one or more link

      if (ls->isup_) {
       printf("%d=%d ",ls->src_,ls->dest_);
      } else {
       printf("%d#%d ",ls->src_,ls->dest_);
      }
    } // for link_stuct
  }
}

//=============================================================================
// Route Table:
// Print entries.
//=============================================================================
void
RoutingTable::print() {
  rtable_node *j;
  int i;
  Time now = Scheduler::instance().clock(); // get the time
```

```
  printf("_%2d_ [%6.6f] | Route Table ",my_address_,now);

  for (i=0; route[i] != NULL && i< MAX_NODES_PER_ZONE ; i++) {

    printf("| Entry %d : ",i);

    if (route[i]==NULL) {
      printf("empty");
      return;
    }

    for (j=route[i]; j!=NULL; j=j->next_  ) {
      printf("%d(%d)->",(int)j->id_,j->numhops_);
    }
  }
  if (i==0) {
    printf("empty ");
  }

}

//===========================================================================
// Route Table:
// Compute minimum hop routes to all nodes that are "radius" hops or less
// with respect to node myadd, given link state table.
//===========================================================================
void
RoutingTable::compute_routes(nsaddr_t myadd, int radius) {
  link_struct *lptr; // a temp ptr for iterative loops
  link_struct *lst;  // a copy of link table
  link_struct* lnk;  // a ptr to newly created links
  link_struct* ls;   // a temp ptr for iterative loops
  int i;             // counter for iterative loop over route table
  int flag;
  Time now = Scheduler::instance().clock(); // get the time
  rtable_node *tmp;

  flag = 0;
  lst = NULL;
  erase(); // route table

  if (linkstatetable->lshead_ == NULL) {    // If link list is empty
    return; // nothing broken, just ran out of link states
  }

  // copy valid links in linkstatetable list to lst
  for (lptr=linkstatetable->lshead_; lptr!=NULL; lptr=lptr->next_) {
    flag = 1;

    if (lptr->isup_) {  // if link is valid, add to lst
      lnk = new link_struct(lptr->src_,lptr->dest_,TRUE,lst);
      lnk->src_ = lptr->src_;
      lnk->dest_ = lptr->dest_;
      lnk->isup_ = TRUE;
      // insert link at head of lst
      if (lst != NULL) {
        lnk->next_ = lst;
        lst = lnk;
      } else { // else if lst is empty
        lst = lnk;
        lst->next_ = NULL;
      } // end else
    }  // end if

  } // end while
  if (!flag)  { }

  route[0] = new rtable_node();
  route[0]->id_= myadd;
  route[0]->next_ = NULL;
  route[0]->numhops_ = 0;
```

```
  route[1] = NULL;

  num_entries_=1;

  // rtable[i] is "current" src we are looking at
  // search for all its neighbors dest(i), which gives us
  // a number of (current src,dest(i) pairs
  // for each pair found, check if dest is in route table
  // if not, add to route table. delete link from lst (set isup_=FALSE)
  // after you check the dest.
  // add-drop will add/drop entries in route table
  for ( i=0; (i < MAX_NODES_PER_ZONE) && (route[i] != NULL); i++) {

    for (ls=lst;  // lst is a copy of linkstatetable, from above
         (ls != NULL);
         ls=ls->next_)
      {
       if (ls->isup_ == TRUE) { // make sure it hasn't been "dropped"
         add_drop(ls,i);
       }
      } // end for ls
  } // end for i

  // calculate peripheral nodes
  for (i=0; (i < MAX_NODES_PER_ZONE) && (route[i] != NULL); i++) {
    if (route[i]->numhops_ == radius ){
      rtable_node *temp = new rtable_node;
      temp->id_ = route[i]->id_;
      temp->next_ = periphnodes_;
      periphnodes_ = temp;
    }
  }
  empty_links(lst);
  return;
}

//=============================================================================
// Route Table:
// Looks up node with address "id" in route table, returns index into table
// if it exists, otherwise sets to NotInTable.
//=============================================================================
int
RoutingTable::get_entry(nsaddr_t id)
{
  int i;
  for (i=0;((route[i] != NULL) && (i<MAX_NODES_PER_ZONE+1)); i++)
    if (route[i]->id_ == id)
      return i;
  return (MAX_NODES_PER_ZONE+1);
}

//=============================================================================
// Route Table:
// Looks up node with address "id" in route table, returns TRUE if in table,
// otherwise false.
//=============================================================================
int
RoutingTable::route_exists(nsaddr_t id)
{
 int i;
  for (i=0;((route[i] != NULL) && (i<MAX_NODES_PER_ZONE+1)); i++)
    if (route[i]->id_ == id)
      return TRUE;
  return FALSE;
}

//=============================================================================
// Linkstate Table:
// Delete out lstates in link state table.
//=============================================================================
```

```
void
RoutingTable::empty_links(link_struct *lst)
{
  link_struct *lnk;
  while (lst != NULL) {
    lnk = lst;
    lst = lst->next_;
    delete lnk;
  }
}

//=============================================================================
// Route Table:
// Looks up node with address "nodeid" in route table,
// returns index if in table, otherwise -1.
//=============================================================================
int
RoutingTable::node_exists(nsaddr_t nodeid)
{
  int i;
  for (i=0;((route[i] != NULL) && (i<MAX_NODES_PER_ZONE+1)); i++)
    if (route[i]->id_ == nodeid)
      return i; // found
  return (-1); // not found
}

//==============================================================================
// Route Table:
// Used by compute_routes() to calculate minimum hop tree.
// Add one more entry to route tree if this node "entree" is found to be
// one hop from an entry already in route tree, ie if node is equal to
// src or dest of this link, "lnk". If a match, this link is marked
// as covered for next pass. If no match, return, caller will continue
// iterating through link state table.
//==============================================================================
void
RoutingTable::add_drop(link_struct *lnk, int entree)
{

  // if link is not up, nothing left to do
  if (lnk->isup_ == FALSE) {
    return;
  }

  // if src is in route table
  if ( lnk->src_ == route[entree]->id_ ) {
    // found a link with same src

    // see if dest exists in route table
    if (get_entry(lnk->dest_) != (MAX_NODES_PER_ZONE+1)) {

      // yes route already exists, don't add
      lnk->isup_ = FALSE; // in effect dropping link
      //return TRUE; // but do delete current link_struct
      return;

    } else if ( route[ get_entry(lnk->src_) ]->numhops_ < agent_->radius_ ) {
      // add node to RouteTable if within radius

      if (num_entries_ == MAX_NODES_PER_ZONE) {
        // error, overflow
        printf("Need to increase MAX_NODES_PER_ZONE in zrp.h to greater"
               "than %d\n",MAX_NODES_PER_ZONE);
      }
      // add a new route to route table, id=dest,
      // next is route entry with id=src
      route[num_entries_] = new rtable_node;
      route[num_entries_]->id_ = lnk->dest_;
      route[num_entries_]->next_ = route[ get_entry(lnk->src_) ];
      if (route[num_entries_]->next_) {
        route[num_entries_]->numhops_ = route[num_entries_]->next_->numhops_+1;
```

```
      } else {
        route[num_entries_]->numhops_ = 0;
      }
      num_entries_++;
      lnk->isup_ = FALSE; // in effect dropping link
    }
    //return TRUE; // tell the caller to delete the current link_struct
    return;

  }  else if ( lnk->dest_ == route[entree]->id_ ) {
    // found a link with same dest

    // see if src exists in route table
    if (get_entry(lnk->src_) != (MAX_NODES_PER_ZONE+1)) {

      // yes route already exists, don't add
      lnk->isup_ = FALSE; // in effect dropping link
      //return TRUE; // but do delete current link_struct
      return;

      // if dest is in route table
    } else if ( route[ get_entry(lnk->dest_) ]->numhops_ < agent_->radius_ ) {

      if (num_entries_ == MAX_NODES_PER_ZONE) {
        // error, overflow
        printf("Need to increase MAX_NODES_PER_ZONE in zrp.h to greater"
               "than %d\n",MAX_NODES_PER_ZONE);
      }
      // add a new route to route table, id=dest,
      //next is route entry with id=src
      route[num_entries_] = new rtable_node;
      route[num_entries_]->id_ = lnk->src_;
      route[num_entries_]->next_ = route[ get_entry(lnk->dest_) ];
      if (route[num_entries_]->next_) {
        route[num_entries_]->numhops_ = route[num_entries_]->next_->numhops_+1;
      } else {
        route[num_entries_]->numhops_ = 0;
      }
      num_entries_++;
      lnk->isup_ = FALSE; // in effect dropping link
    }
    return;
  }
  return;
}
```
// End of ZRP_TABLE.CC===

// ZRP_TABLE.H===

```
#ifndef _zrp_table_h_
#define _zrp_table_h_

#include <config.h>
#include <agent.h>
#include <packet.h>
#include <ip.h>
#include <delay.h>
#include <scheduler.h>
#include <queue.h>
#include <trace.h>
#include <arp.h>
#include <ll.h>
#include <mac.h>
#include <priqueue.h>
#include <delay.h>
#include <random.h>
#include <object.h>
#include <route.h>
#include "zrp.h"

//typedef int32_t Query_ID;
```

```
typedef int Query_ID;
typedef int BOOLEAN;
typedef double Time;

#define LINKSTATE_EXPIRATION 25

#define MAX_NODES_PER_ZONE 50
#define MAX_OUTER_ROUTES 50

class ZRPAgent; // pre-declaration

//============================================================================
// IERP Cache Table:
// A single entry in the outgoing IERP requests table, has an expiry.
//============================================================================
struct IERPCacheEntry {

  Query_ID qid_;
  Packet *pkt_;
  Time expiry_;
};

struct rtable_node; // pre-declaration needed for next struct

//============================================================================
// Query Detect Table:
// A single entry in the QD table, has an expiry.
//============================================================================
struct QueryDetectEntry {

  nsaddr_t originid_;
  Query_ID qid_;
  Time expiry_;
  rtable_node *next_; // list of covered nodes

};

//============================================================================
// Neighbor Table:
// Neighbor table is a linked list of these.
//============================================================================
struct neighbor_struct
{
  nsaddr_t addr_;
  neighbor_struct* next_;
  Time expiry_;
  Time lastack_;

};

//============================================================================
// Linkstate Table:
// Link state atomic data, contains state in isup_ (up or down) ans seq.
//============================================================================
class link_struct {
public:

  link_struct() : seq_(0), isup_(FALSE) {//, next_(NULL) {
    bzero(this, sizeof(link_struct));
  }

  link_struct(nsaddr_t srcid, nsaddr_t destid, BOOLEAN valid,link_struct* next)
    {
      src_=(srcid);
      dest_=(destid);
      isup_=(valid);
      //      next_=(next);
      seq_=0;
    }

  nsaddr_t src_;   // 32 bits
```

```
  nsaddr_t dest_;  // 32 bits

  int flag_; // flag used for Dijstkra's algorithm
  int isup_;
  int seq_;  // sequence number
  Time expiry_;

  //  Time timestamp;

  link_struct* next_;
};

//==========================================================================
// Linkstate Table:
// Class for storing Link States in node.
//==========================================================================
class LinkStateTable {
 public:

  LinkStateTable() : lshead_(NULL) { }

  nsaddr_t  my_address_; // debug

  link_struct* lshead_;

  link_struct* last() ;

  void add(link_struct *link);  // quietly ignore errors ;)

  int remove(link_struct *link);

  BOOLEAN update(nsaddr_t src, nsaddr_t dest,
                int state, int seq, int *seq_match);

  int exist(link_struct *link);

  void print_links();
  void purge_links();


};

//==========================================================================
// Route Table:
// A single data struct naming a node, used for storing node addresses
// in various tables.
//==========================================================================
struct rtable_node {
  nsaddr_t id_;
  int numhops_; // number of hops from me
  rtable_node* next_;
  rtable_node() : id_(0), next_(NULL) { }
  rtable_node(nsaddr_t newid) : id_(newid), next_(NULL) { }
};

//==========================================================================
// Route Table:
// This struct is a head pointer for a route entry, has expiry timestamp.
// Route is at next_ .
//==========================================================================
struct rtable_entry {
  nsaddr_t id_;
  Time expiry_;
  int numhops_; // number of hops from me
  rtable_node* next_; // route is here
  rtable_entry() : id_(0), next_(NULL), expiry_(0), numhops_(0)  { }
  rtable_entry(nsaddr_t newid) : id_(newid), next_(NULL), expiry_(0),
    numhops_(0) { }
};

//==========================================================================
```

```
// Route Table:
// Class includes a tag (agent_) to agent it resides in to access friendly
// data and methods.
//=====================================================================
class RoutingTable {
  public:

  RoutingTable(ZRPAgent* a)  { // init
    num_entries_=(0);
    agent_ = a;
  }

  ZRPAgent* agent_;

  nsaddr_t  my_address_;

  int num_entries_;

  // Tables
  rtable_node* route[MAX_NODES_PER_ZONE+1]; // Last ptr is always null
  rtable_entry* outer_route[MAX_OUTER_ROUTES]; // Last ptr is always null
  rtable_node* periphnodes_;
  LinkStateTable *linkstatetable;

  // Methods
  void RoutingTable::compute_routes(nsaddr_t myadd, int radius);
  void erase();
  void print();
  // returns pointer to entry to node's route
  int get_entry(nsaddr_t node_id);
  int route_exists(nsaddr_t id);

  // drop from link table, add node to RouteTable
  void add_drop(link_struct* ls, int i);

  // Finds Next Hop On route to given,
  //returns FALSE if doesn't exist
  int next_hop_to(nsaddr_t node) { }

  // returns route entry index for given node
  int node_exists(nsaddr_t nodei);

  void empty_links(link_struct *lst);

  //  int rmax; // the number of hops to the farthest node in my LRZ
};

#endif
```